A HANDS-ON FIRST COURSE

GODOT 4

for Absolute Beginners

From your first node to your first finished games



AN UNOFFICIAL GUIDE

Y2KMALLRAT

Godot 4 for Absolute Beginners

A hands-on first course in game development, from your first node to your first finished games

y2kmallrat

An unofficial guide
Edition 1, 2026
Written for Godot 4.7 (stable)

Contents

Welcome

Part I: Meet Godot

Chapter 1: What Is a Game Engine? Meet Godot

Chapter 2: Nodes and Scenes: The Building Blocks

Chapter 3: Your First Script: Variables, Types, and the Output Panel

Chapter 4: Making Decisions and Repeating

Chapter 5: Scripts That Talk to the Game

Part II: Making 2D games

Chapter 6: 2D Space and Sprites

Chapter 7: Input and Movement

Chapter 8: Physics and Collisions

Chapter 9: Mini-Game 1, Coin Dash

Chapter 10: Animation

Chapter 11: Sound and Music

Chapter 12: Tilemaps and Level Building

Chapter 13: The Platformer Project

Part III: Interfaces and game structure

Chapter 14: User Interface

Chapter 15: Signals in Depth and Changing Scenes

Chapter 16: Autoloads, Game State, and Custom Resources

Chapter 17: Saving and Loading

Chapter 18: Project: Pip's Coin Run, a Complete Game

Part IV: Going further

Chapter 19: Your First Steps in 3D

Chapter 20: Particles, Shaders, and Game Feel

Chapter 21: Game AI and Navigation

Chapter 22: The Debugger, the Profiler and Good Habits

Part V: Finishing and sharing

Chapter 23: Exporting Your Game

Chapter 24: Where to Go Next

Appendices

Appendix A: GDScript Quick Reference

Appendix B: Editor Shortcuts

Appendix C: Troubleshooting Guide

Appendix D: Answers to the Review Questions

Answers: Chapter 2 exercises (hints)

Answers: Chapter 3 exercises (hints)

Exercise hints (selected)

Answers: Chapter 16 exercises (hints)

Answers: Chapter 17 exercises (hints)
Answers: Chapter 18 exercises (hints)
Appendix E: Glossary
Appendix F: Free Assets and Tools
Appendix G: Math for Games, Just Enough

© 2026 y2kmallrat

The text, diagrams, screenshots and cover art of this book are licensed under the Creative Commons Attribution-NonCommercial 4.0 International license (CC BY-NC 4.0, https://creativecommons.org/licenses/by-nc/4.0/). You may share and adapt them with credit to the author, but not for commercial use. The example projects and code listings are released under the MIT License, so you can open, change, and learn from them freely.

An unofficial guide. This book is an independent, unofficial guide to the free Godot Engine. All text in it was written for this book, and the diagrams were drawn for it. No passages were copied from the official Godot documentation; where the book points you to the official docs, it only links to them. The screenshots show the Godot editor, which is free software.

Godot Engine is Copyright (c) 2014-present Godot Engine contributors and Copyright (c) 2007-2014 Juan Linietsky and Ariel Manzur, and is released under the MIT License (full text in the editor under *Help > About Godot > License*). Thank you to the Godot contributors for making it.

The example projects use no outside art, fonts, or sounds: everything is drawn or generated in code, and the Godot logo is not used on the cover or in the projects.

Godot and the Godot logo are trademarks of the Godot Foundation. This book is an independent educational work. It is not produced, endorsed, or sponsored by the Godot Foundation or by the Godot Engine contributors.

Welcome

You do not need to know anything to start this book. You do not need to have written a line of code. You do not need to be good at drawing, music, or math. You need a computer, some curiosity, and a little patience with yourself.

By the last chapter you will have installed a professional game engine, learned the basics of programming, built several small games from scratch, and exported one so that a friend can play it. Along the way you will make many mistakes. That is not a sign that something is wrong. Making a mistake, reading the message, and fixing it is what programming actually is, and this book teaches that skill on purpose.

Who this book is for

This book is for you if you are:

- a college or university student taking a first course in game development or programming,
- a self-taught learner who wants a structured path instead of a pile of scattered videos,
- a teacher or mentor looking for a course that can be followed from the first page without prior knowledge,
- anyone who has said "I would love to make a game, but I do not know where to start."

It is not necessary to have used any game engine before. It is not necessary to know any programming language. Every programming idea (a variable, a function, a loop) is explained from zero, with everyday comparisons, the first time it appears.

What you will need

- A computer running Windows, macOS, or Linux that can run Godot 4. Almost any computer from the last several years will do for the 2D chapters. Chapter 19 (3D) benefits from a computer with a reasonably modern graphics chip, but the examples are small.
- An internet connection to download Godot (it is free, and roughly 100 to 200 MB).
- A mouse is strongly recommended. A keyboard is required. A gamepad is optional.
- About 3 to 6 hours per chapter if you read carefully and do the projects. Short chapters take less.

You do **not** need to buy anything. Every project in the book can be built with shapes, colors, and sounds that Godot itself provides. Appendix F lists free art, music, and sound if you would like to decorate your games.

How the book is organized

The book has 24 chapters in five parts, followed by seven appendices.

Part	Chapters	What happens
I. Meet Godot	1 to 5	Install Godot, learn the editor, learn nodes and scenes, learn programming from zero.
II. Making 2D games	6 to 13	Sprites, input, movement, physics, your first complete game, animation, sound, tilemaps, a platformer.
III. Interfaces and game structure	14 to 18	Menus and user interface, signals, scene changes, autoloads, saving and loading, a complete game with title and pause screens.
IV. Going further	19 to 22	A first look at 3D, visual effects, enemy behavior, debugging, and good habits.

Part	Chapters	What happens
V. Finishing and sharing	23 to 24	Exporting your game, publishing it, and deciding what to build next.

The appendices are for looking things up:

Appendix	Contents
A	GDScript quick reference
B	Editor shortcuts
C	Troubleshooting guide: symptom, cause, fix
D	Answers to the review questions (one file per chapter)
E	Glossary
F	Free assets and tools
G	Math for games, just enough, with pictures

Read in order the first time

The chapters build on one another. Chapter 4 uses ideas from Chapter 3, and the projects in Part II reuse characters and code from earlier chapters. If you are new, read in order. Later, once you know the basics, you can jump around freely.

If you already know some programming, you may skim Chapters 3 and 4, but do their projects anyway. Godot has its own way of organizing a game (nodes and scenes), and Chapter 2 is worth reading closely even for experienced programmers.

How each chapter is structured

Every chapter has the same shape, so you always know where you are.

1. **Learning objectives.** What you will be able to do by the end. Read these first. They are the goal.
2. **Key terms.** The new words in the chapter, each with a one-line definition. You can skim them before you start and use them as a checklist afterward.
3. **Numbered sections** (for example 4.1, 4.2, 4.3). The lesson itself. Concepts come first, then a worked example, then a "Try it" so you can practice at once.
4. **The project.** A hands-on build with numbered steps, usually 30 to 90 minutes. It ends with an **Expected result** so you can check your work.
5. **Chapter summary.** The main points in a short list.
6. **Review questions.** Eight to ten questions to test your understanding. Answers are in Appendix D, in a separate file for each chapter. Try the questions before you look.
7. **Exercises.** Three to five tasks: *Practice* (easy), *Apply* (medium), and *Challenge* (hard). Do at least the Practice and Apply ones.
8. **Common mistakes.** The problems most beginners run into, each written as symptom, cause, fix.
9. **Further reading.** Links to the official Godot documentation for going deeper.

Conventions used in this book

Code

Code appears in shaded blocks. The block is labeled with the language:

```gdscript
extends Node2D

func _ready() -> void:
    print("Hello, Godot!")
```

Before each script, the text tells you three things: the **file name**, the **node type the script attaches to**, and the **scene tree** it lives in. Scene trees are drawn like this:

```
Main (Node2D)
├── Player (CharacterBody2D)
│   ├── Sprite2D
│   └── CollisionShape2D
└── Label
```

Terminal commands are labeled `bash`:

```bash
godot --version
```

GDScript uses the Tab key for indentation. The code blocks in this book use tabs. If you type the code yourself, the Godot script editor will insert tabs for you. If you copy and paste, keep the indentation exactly as shown. Indentation is part of the language, not decoration.

A piece of code that is not complete on its own is called a **fragment**, and the text says where it goes.

Names

- `Monospace text` is for code, file names, node names, property names, and anything you type or see literally in the editor.
- **Bold text** marks a key term the first time it is defined.
- *Italic text* is for emphasis, titles, and menu paths written like *Project > Project Settings*.
- Keyboard keys are written like **Ctrl+S**. On macOS, **Ctrl** usually becomes **Cmd**. Appendix B lists the differences.

Callouts

Short boxes break up the text:

> **Note:** Extra information that is useful but not essential.
>
> **Warning:** Something that commonly goes wrong, or that can lose your work.
>
> **Why it works:** A short explanation of what is happening underneath.
>
> **Try it:** A small activity. Stop reading and do it. This is where learning happens.

Figures

Figures are numbered by chapter: Figure 7.2 is the second figure in Chapter 7. Diagrams explain ideas. Screenshots show the real Godot editor or a running game. The editor's default dark theme is used throughout. Your editor may look slightly different if you have changed its theme, language, or window layout.

Versions

The book is written for **Godot 4.7**. Godot 4 releases are designed to stay compatible with one another, so everything here works on 4.4 and later. If a feature exists only in newer versions, the text says so. This book uses **GDScript** only. It does not use the C# edition of Godot.

> **Warning:** Many tutorials online were written for Godot 3. Godot 3 and Godot 4 are different. If you copy code from an old tutorial and it fails, check whether it uses names such as `KinematicBody2D`, `Sprite`, `yield`, `onready`, or `export var`. Those are Godot 3. Appendix C explains how to translate them.

Optional: the finished projects

You do not need any files to follow this book. Every chapter gives every step, and you build each project yourself.

For each chapter there is also a finished, ready-to-open Godot project, one folder per chapter, named with the chapter number and a short label (for example `ch09-coin-dash`). If you have these optional files, you can use them to check your work when something does not behave as described, or to see the finished result before you start.

To open one:

1. Open Godot. In the Project Manager, choose **Import**.
2. Browse to the folder of the project you want and select its `project.godot` file.
3. Click **Import & Edit**.

Each project opens on its own and does not depend on any other.

> **Try it:** Do your own build first. When you are stuck, compare your project to the finished one. Looking at a working project beside a broken one is one of the best ways to learn what each part does.

A note on Godot's license and name

Godot Engine is free and open-source software, released under the **MIT License**. That is a very permissive license. In plain words: you may use Godot for any purpose, including commercial games, you do not pay royalties or fees, and the games you make belong to you. The only requirement is that the license text and copyright notice for Godot are included with distributed copies of the engine. Godot's editor provides these in its *Help > About Godot* window, and when you export a game you can include them in your credits or a licenses file. Chapter 23 shows how.

Godot also includes a number of third-party libraries, each with its own license. Those are all listed in the same *About* window.

"Godot" and the Godot logo are trademarks of the Godot Foundation. The MIT License covers the engine's code, not the name and logo, so this book uses them only to refer to the engine, and does not imply any endorsement. If you publish your own tools, courses, or products, check the Godot Foundation's trademark policy on the official website before using the name or logo in a way that could suggest official approval.

This is general information, not legal advice.

A word before you begin

Some chapters will feel easy. Some will feel hard. Both are normal. If a section does not make sense, read it again later, or do the "Try it" first and read the explanation afterward. If your code does not work, do not worry: Appendix C exists for exactly that moment, and Chapter 22 teaches you to find problems yourself.

Take breaks. Save often. Keep every project you make, because many of them become the starting point for the next.

Now turn the page. Chapter 1 begins with a simple question: what is a game engine?

Part I: Meet Godot

Chapter 1: What Is a Game Engine? Meet Godot

Learning objectives

By the end of this chapter you will be able to:

- Explain in your own words what a game engine is and which jobs it does for you.
- Describe the game loop and why a game is a picture that is redrawn many times every second.
- Download and start the standard (non-.NET) version of Godot 4 on Windows, macOS, or Linux.
- Use the Project Manager to create, open, and organize projects.
- Name the main parts of the Godot editor (viewport, Scene dock, FileSystem dock, Inspector, bottom panels) and say what each is for.
- Build a small scene, save it, set it as the main scene, and run it as a real program.

Key terms

- **Game engine**: a program (or toolkit) that handles the jobs almost every game needs, such as drawing, sound, input, and physics, so you can focus on what makes your game special.
- **Godot**: a free, open-source game engine and editor that can make 2D and 3D games.
- **Open source**: software whose source code is public and whose license lets anyone use, study, and change it. Godot uses the MIT license.
- **Editor**: the application window where you build your game: place things, change settings, write scripts, and press Play.
- **Project**: a folder that holds everything for one game: scenes, scripts, images, sounds, and a settings file.
- **Project Manager**: the first window Godot shows. It lists your projects and lets you create, import, and open them.
- **Scene**: a saved, reusable piece of your game, such as a character, a level, or a menu. (Chapter 2 covers scenes in depth.)
- **Node**: the basic building block of Godot. Every object in a scene is a node.
- **Inspector**: the editor panel that shows the properties (settings) of the selected node.
- **Game loop**: the cycle every game repeats many times per second: read input, update the world, draw the picture.
- **Frame**: one single picture drawn by the game loop. A game that runs at 60 frames per second (FPS) draws 60 pictures every second.
- **Main scene**: the scene Godot starts when you run the whole project.

1.1 What is a game engine?

Imagine you want to make a small game in which a round blue character walks around a room and picks up coins. Sounds simple. But if you started from a completely empty computer program, you would have to solve all of these problems yourself before the first coin ever appeared:

1. **Open a window** on the screen, on every kind of computer you care about.
2. **Draw** shapes and pictures into that window, quickly, using the graphics chip.
3. **Listen** to the keyboard, the mouse, and game controllers.
4. **Keep time**, so the character moves at the same speed on a slow laptop and a fast desktop.

5. **Play sound** through the speakers.
6. **Detect collisions**, so the character knows when it touches a coin or a wall.
7. **Load and save files**: pictures, sounds, and the player's progress.
8. **Package** everything into a program that other people can run.

Each of these is a serious piece of engineering. Professional programmers spend years on just one of them. A **game engine** is a collection of ready-made solutions to these jobs, wrapped together in one tool. Instead of building the road, you get a car and drive.

> **Why it works:** Engines exist because games have so much in common. Nearly every game draws pictures, reads input, plays sound, and checks for collisions. Solving those problems once, carefully, and sharing the solution saves everyone time.

What an engine usually contains

Look at Figure 1.1. It shows the typical parts of an engine. You do not need to memorize them; you only need to know that when you press a key or hear a sound in a Godot game, one of these parts is quietly doing the work.

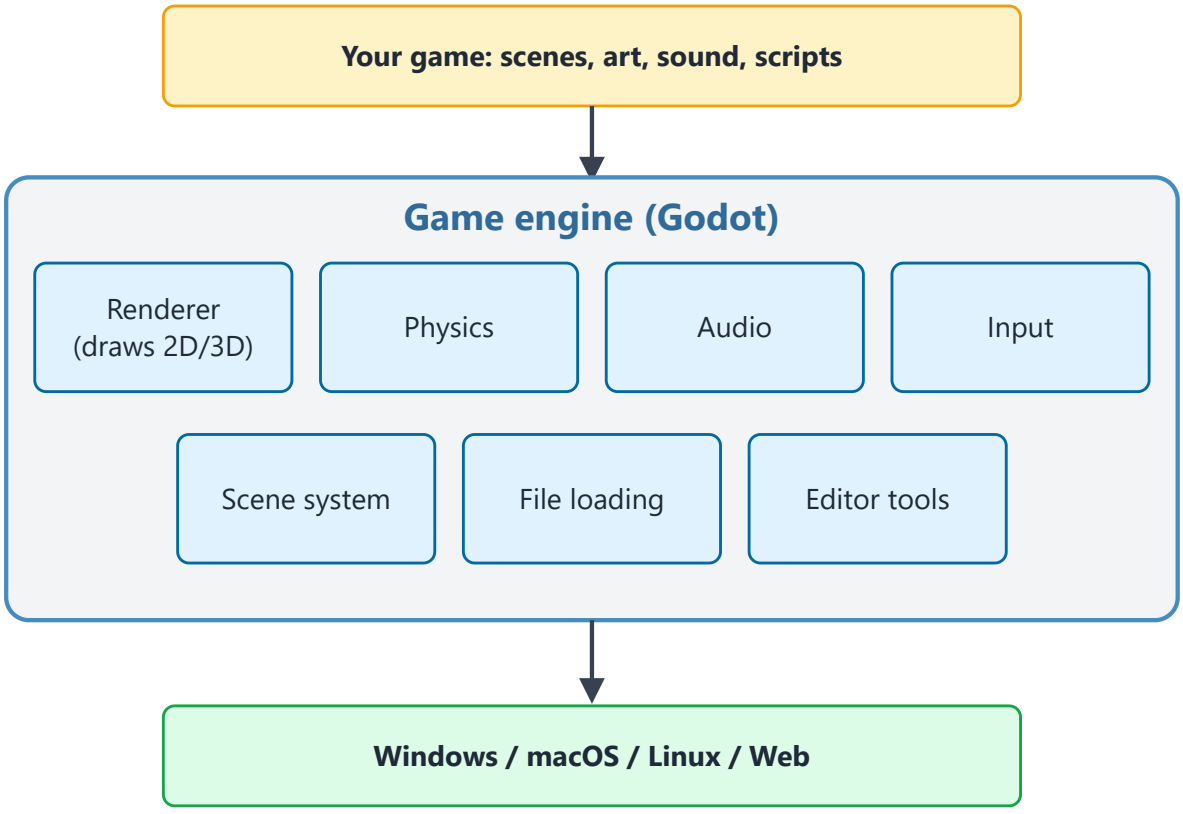


Figure 1.1: A game engine bundles many systems. Your game's content (art, sound, levels, and scripts) sits on top and tells the engine what to do.

- The **renderer** draws everything you see, in 2D or 3D.
- The **physics system** moves objects and detects when they bump into each other.
- The **audio system** plays and mixes sounds and music.
- The **input system** turns key presses, mouse clicks, touches, and button presses into information your game can use.

- The **scene system** organizes your game into pieces you can build, save, and reuse. In Godot this is the heart of everything, and Chapter 2 is about it.
- **Tools** let you build the game visually: place objects, preview animation, edit tile maps, and so on.

The engine is the same for every game. Your **content** is what differs: the art, the sounds, the levels, and the **scripts**, which are small programs you write to say "when the player touches a coin, add one to the score."

Engine versus library versus game

A word of caution about vocabulary:

- A **library** is code you call from your own program. You are in charge, and the library helps when asked.
- An **engine** usually works the other way around. The engine runs the main program and *calls your code* at the right moments, for example "a frame is about to be drawn; do you want to move anything?"
- A **game** is the final result: your content plus the engine, packaged for players.

When you export a Godot game (Chapter 23) the engine's runtime goes into the package alongside your content. Players never see the editor.

The game loop

Here is the most important idea in this whole chapter. A game looks like a world that moves, but really it is a stack of still pictures shown one after another, just like a flip book. Each picture is called a **frame**. The engine repeats the same short cycle over and over, usually 60 times every second, and that cycle is called the **game loop**.

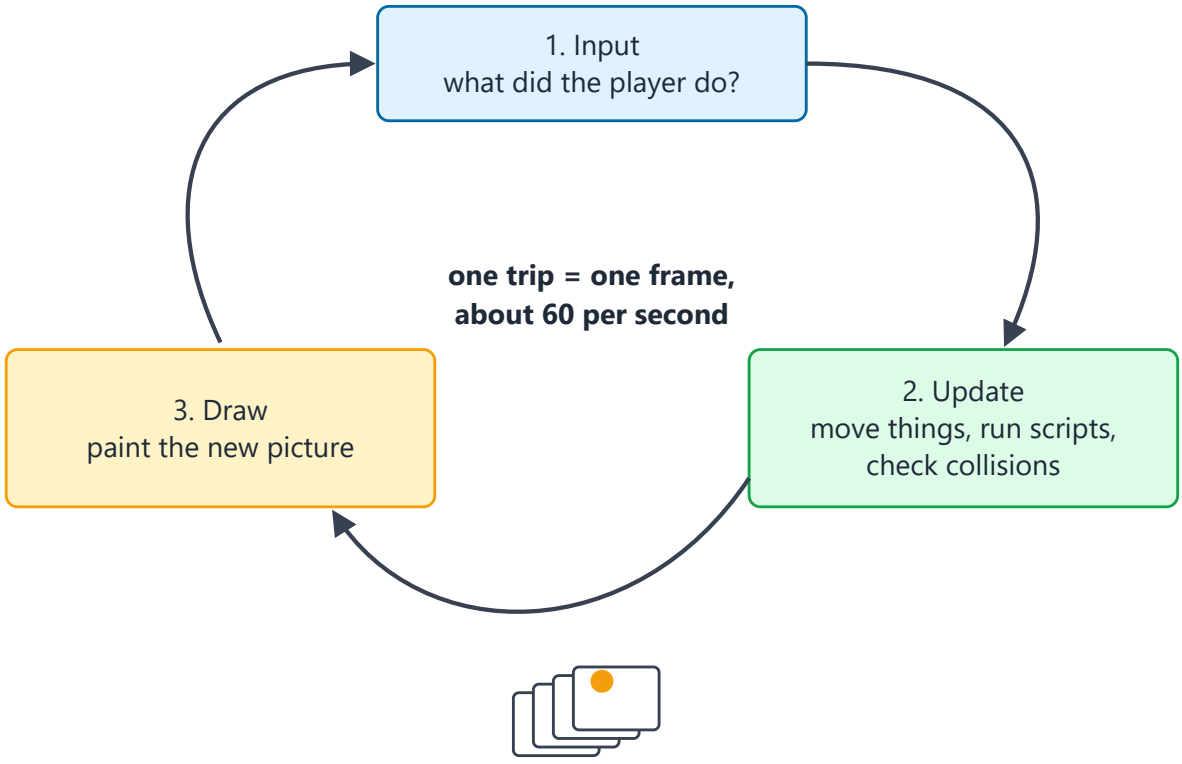


Figure 1.2: The game loop. One trip around the circle produces one frame.

In every trip around the loop the engine does three things:

1. **Input**: find out what the player did since the last frame. Was a key pressed? Did the mouse move?
2. **Update**: change the world. Move the character, run your scripts, check collisions, advance animations.
3. **Draw**: paint the new picture on the screen.

If the loop runs 60 times a second, each trip has about one sixtieth of a second, roughly 16 milliseconds, to finish. If it takes longer, the game feels choppy. You will meet this idea again and again. When you write a script that moves the hero a tiny bit each frame, you are writing code that runs inside the update step of the loop.

> **Note:** You do not have to build the loop yourself. Godot owns it. You simply say what should happen during it.
>
> **Try it:** Take a pencil and a stack of sticky notes, or the corner of a notebook. Draw a stick figure at the left edge of each page, moving it slightly to the right on every page. Flip the pages quickly. You just did what the game loop does: 20 or 30 frames, played fast, look like motion.

1.2 Meet Godot

Godot (pronounced "GOD-oh", like the character in the play *Waiting for Godot*) is a game engine that is:

- **Free.** There is no price, no subscription, and no fee on the games you make and sell.
- **Open source.** It is released under the MIT license, a very permissive license. Its code is public, and a large community of volunteers and a non-profit foundation maintain it.
- **Small and fast to start.** You download one file, unzip it, and run it. There is no long installer.
- **Both 2D and 3D.** Godot has a dedicated 2D engine, not just a 3D engine pretending to be flat. That makes it a particularly good first engine for 2D games, which is where this book spends most of its time.
- **Friendly to learn.** Its scripting language, **GDScript**, looks a lot like Python and was designed for making games.

Godot's big idea: everything is a node

Godot builds games out of **nodes**. A node is one small object that does one job. There is a node that draws a picture (`Sprite2D`), a node that plays a sound (`AudioStreamPlayer`), a node that shows text (`Label`), a node that detects collisions, and hundreds more. You combine nodes into trees, save a tree as a **scene**, and reuse scenes inside other scenes. Chapter 2 is entirely about this. For now, remember only: *a game in Godot is made of nodes, and nodes are grouped into scenes.*

The editor is also a Godot game

A fun fact: the Godot editor is itself built with Godot. The buttons, panels, and menus you click are nodes. That is one reason the editor feels consistent, and why the skills you learn using it carry over to building your own tools later.

Languages

Godot has several ways to write scripts:

- **GDScript**, the built-in language. This book uses it exclusively.
- **C#**, available in a separate ".NET" download of Godot.
- **GDExtension**, for advanced users who want to write fast native code (C++ and others).

You do not need any of the last two. For a first course, GDScript is the best choice: it is built into the editor, needs no setup, and has the shortest road from idea to working game.

Versions

Godot is currently in its **4.x** series. This book is written for **Godot 4.7**, and the lessons also work on recent earlier 4.x versions such as 4.4 and later. A few menu items may move slightly between versions. If something on your screen looks a little different from a figure in this book, look for the same name nearby. Godot 3 is an older, different version; tutorials for it use different code, so check that any tutorial you find says "Godot 4".

1.3 Installing Godot

Godot does not install in the usual sense. You download a compressed file (a `.zip`), extract the program inside, and run it. There is no registry entry, no background service, and removing it later means deleting the file.

Step 1: Download the right version

1. Open your web browser and go to **https://godotengine.org/download**.
2. Choose your operating system: **Windows**, **macOS**, or **Linux**. The site normally detects yours and highlights it.
3. Choose the **standard** version, usually labeled just "Godot Engine" (not ".NET"). The .NET version is for C# programmers and is not needed here.
4. Download the **stable** release. Avoid the "dev," "beta," and "release candidate" builds for now.

> **Warning:** Make sure you pick the standard version and not the one labeled ".NET" or "Mono." The .NET build needs extra software installed and is not what this book uses. Also make sure the page says **4.x**, not 3.x.
>
> **Note:** You should download Godot only from the official site, godotengine.org (or from the official Godot page on Steam, itch.io, or your operating system's package manager). Avoid unofficial download sites.

Step 2a: Windows

1. Find the downloaded `.zip` file, usually in your **Downloads** folder.
2. Right-click it and choose **Extract All...**, then choose a folder you will remember. A good choice is something like `C:\Godot`. Avoid running Godot from inside the zip.
3. Open the folder. You will see two programs with similar names. One ends in `_win64.exe` and the other in `_win64_console.exe`. Double-click the one **without** "console" in the name. (The console version opens an extra black text window; it is useful for advanced work and for automated testing, but you do not need it.)
4. If Windows shows a blue "Windows protected your PC" box, it is a common message for downloaded programs. Click **More info**, then **Run anyway**, if you downloaded it from the official site.
5. Right-click the program on the taskbar and choose **Pin to taskbar**, or make a shortcut on your desktop, so you can find it again.

Step 2b: macOS

1. Open the downloaded `.zip`. Your Mac extracts it to a file named **Godot** (the app).
2. Drag **Godot** into your **Applications** folder.

3. Open it from Applications. The first time, macOS may warn that the app was downloaded from the internet. If it refuses, hold **Control** while clicking the app and choose **Open**, or allow it in **System Settings > Privacy & Security**.

Step 2c: Linux

1. Extract the `.zip` to a folder, for example `~/Godot`.
2. The file you want is named something like `Godot_v4.x-stable_linux.x86_64`. Make it executable and run it. In a terminal:

```
cd ~/Godot
chmod +x Godot_v4.7-stable_linux.x86_64
./Godot_v4.7-stable_linux.x86_64
```

Use the exact file name that you downloaded. On many desktops you can also right-click the file, open **Properties**, and tick "Allow executing file as program," then double-click it. Many Linux distributions also offer Godot through their package manager or Flathub; those versions are fine as long as they are Godot 4.

Step 3: First launch

When Godot opens for the first time it may ask you to choose a language and a theme, or it may jump straight into the **Project Manager**. If it asks, pick whatever feels comfortable. All of these can be changed later in the editor settings.

> **Try it:** Start Godot now. If you see a window with a list of projects (probably empty) and buttons on the side, you are in the Project Manager. Congratulations: the installation is finished.

If nothing starts, check the "Common mistakes" section at the end of this chapter.

What do I need to run it?

Godot runs on modest computers. A reasonably recent computer with a graphics card or integrated graphics that supports modern graphics drivers is enough for everything in this book. If your computer is quite old, you will want to use the **Compatibility** renderer, which we meet in the next section.

1.4 The Project Manager

Every time you start Godot, the first window is the **Project Manager**. It is the front door. Its job is to keep track of your projects and to start the editor for the one you choose.

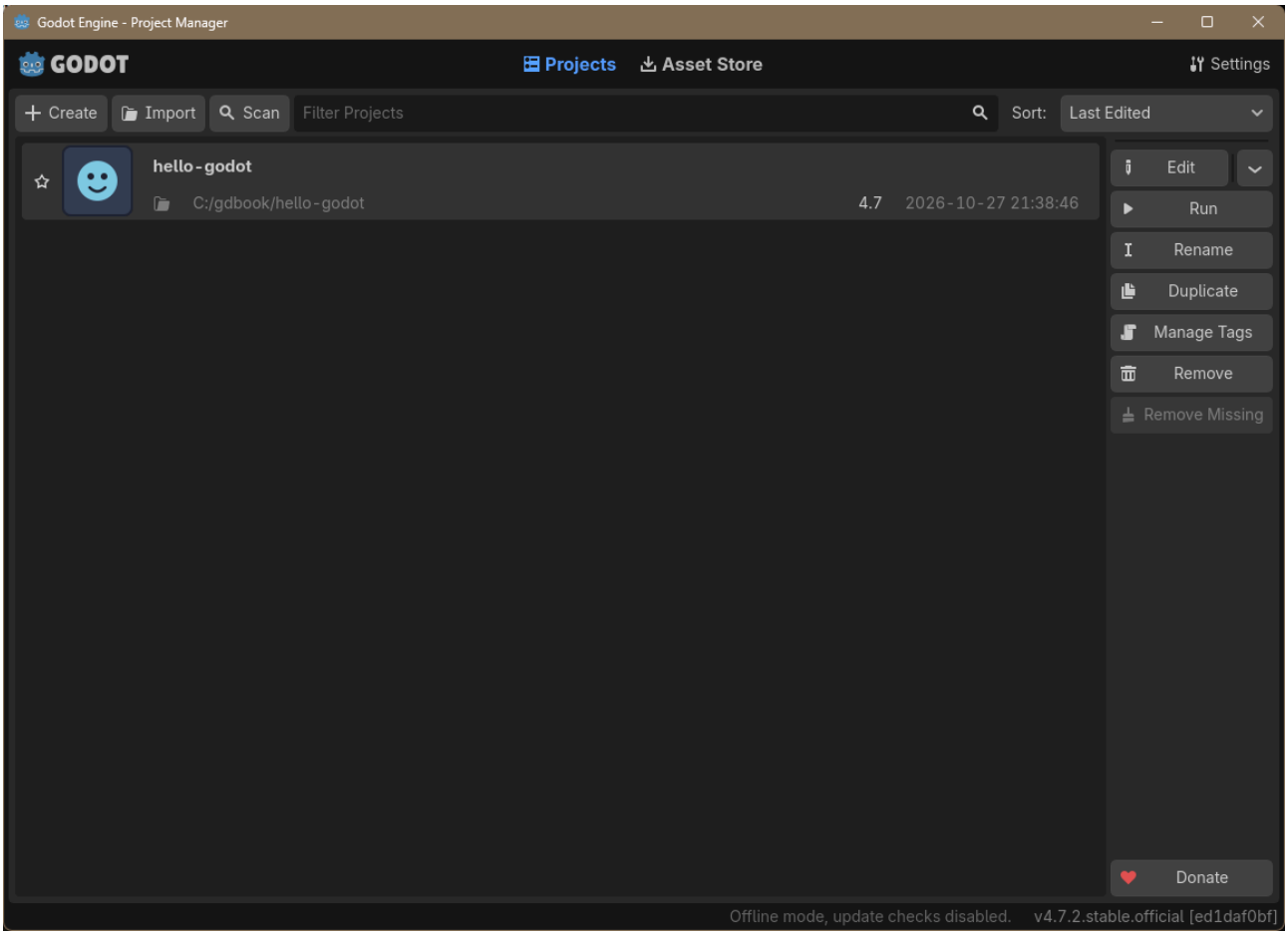


Figure 1.3: The Project Manager. Your projects appear in the list. Action buttons are along the side.

What is a project?

A **project** is simply a folder on your computer. Inside is a special text file called `project.godot` that marks the folder as a Godot project and stores its settings, plus all the things that make up your game. A typical small project looks like this:

```
my-first-game/
├── project.godot      <- settings; this file makes the folder a Godot project
├── icon.svg           <- the default project icon
├── main.tscn          <- a scene you saved
├── player.gd          <- a script
└── .godot/            <- cache; Godot creates and manages it for you
```

Two details are worth remembering now:

- Files that you create (scenes, scripts, pictures) live **inside the project folder**. Godot refers to this folder as `res://`, short for "resources." A path like `res://main.tscn` means "the file `main.tscn` at the top of the current project folder." You will see `res://` constantly.
- The hidden `.godot` folder is a cache. You can delete it safely; Godot rebuilds it. Never put your own files in it.

The buttons and tabs

The exact layout can differ a little between versions, but you will find these functions:

Function	What it does
Create (or "New")	Makes a new project in a folder you choose.
Import	Adds an existing project folder to the list (choose its `project.godot`).
Scan	Looks inside a folder for projects and adds all it finds.
Edit	Opens the selected project in the editor. Double-clicking a project does the same.
Run	Runs the selected project's game directly, without opening the editor.
Rename, **Duplicate**, **Manage Tags**	Housekeeping for the project list entry.
Remove	Takes the project off the list. It does **not** delete the files.

Near the top there are also tabs. **Projects** is your list. **Asset Store** lets you browse free community-made assets and demo projects; you can use it later. A **Settings** button sits at the top right.

Creating a project

1. In the Project Manager, click **Create**. A dialog appears.
2. **Project Name:** type `hello-godot`.
3. **Project Path:** choose where it will live. Click **Browse** and pick a folder such as your Documents folder. Godot makes a subfolder named after your project. Use a normal folder, not one that is inside another Godot project, and avoid folders that sync in real time with cloud services while Godot is writing files.
4. **Renderer:** this chooses how Godot draws. Leave the default selected unless your computer is old. The three options are:
    - **Forward+**: the most capable renderer, for modern computers. This is the default.
    - **Mobile**: tuned for phones and tablets.
    - **Compatibility**: works on the widest range of hardware, including old computers and the web. If Godot gives a graphics error or your computer is old, create the project again with this one.
5. **Version control metadata:** a setting that prepares the project for a tool called Git. If you see it, you can leave it at its default. We discuss Git in Chapter 22.
6. Click **Create** (in some versions, **Create & Edit**). The editor opens.

> **Note:** For the 2D games in this book the choice of renderer hardly matters. You can switch it later in **Project > Project Settings** (the renderer lives under **Rendering > Renderer**). If your computer is old, pick Compatibility from the start and you will avoid trouble.
>
> **Why it works:** A project is just a folder, which means you can copy it, zip it, back it up, or share it, the same way you handle any other folder. That is also how Git and file-sync tools work with it later.

1.5 A tour of the editor

When the editor opens, you will see a lot of panels. This looks like a busy cockpit, but only a few parts matter for now. Take a moment with Figure 1.4, and then we visit each area in turn.

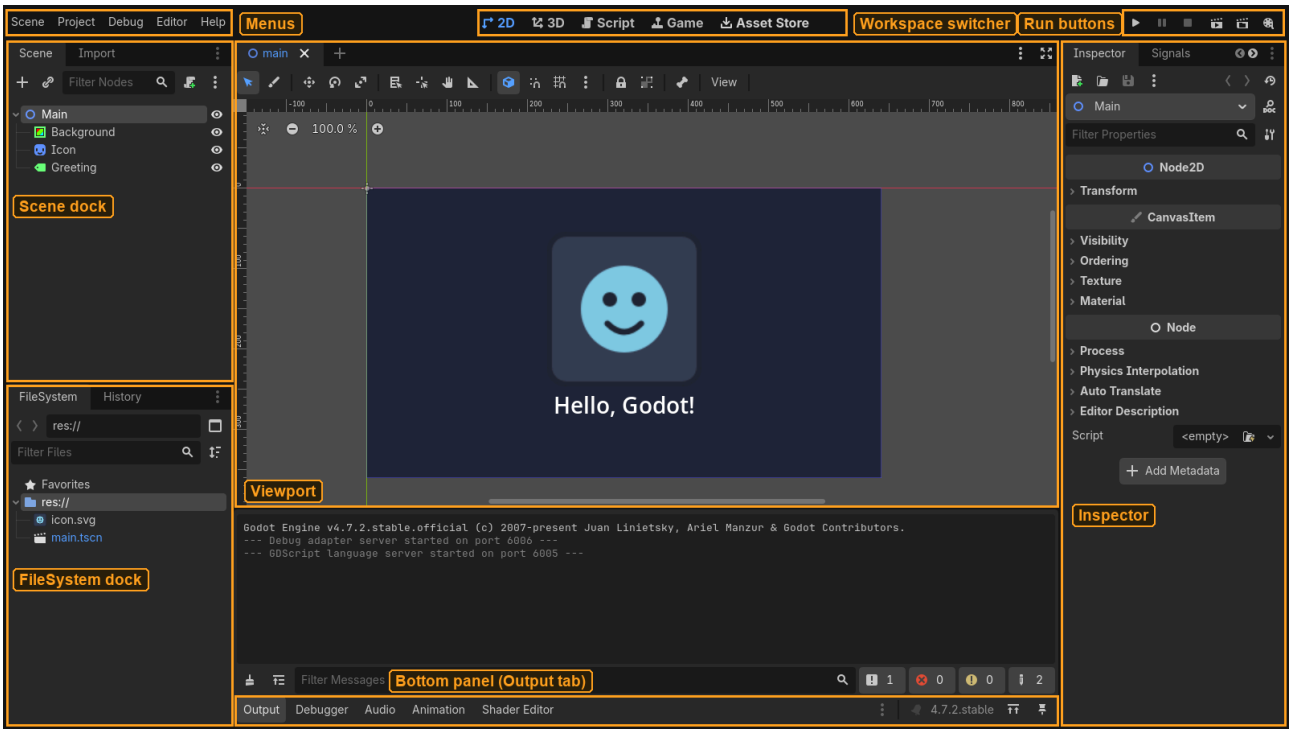


Figure 1.4: The editor. Left: Scene dock above the FileSystem dock. Center: the viewport. Right: the Inspector. Bottom: panels such as Output. Top right: the run buttons.

The top bar

- **Menus** on the left: **Scene**, **Project**, **Debug**, **Editor**, and **Help**. You will use **Scene** to save and **Project > Project Settings** to change project-wide options.
- **Workspaces** in the center. These are big tabs that change the central view: **2D**, **3D**, **Script**, and **Game** (in recent versions), and **Asset Store**. Click **2D** to work on flat scenes. Click **Script** to write code. Click **Game** to watch your running game inside the editor window in versions that provide it.
- **Run buttons** on the right. We cover these in section 1.7.

The Scene dock (top left)

This panel shows the **tree of nodes** in the scene you are editing. It starts out empty with a few buttons that offer to create a root node. Everything you add to your game will appear here as a list of nodes, indented to show which node belongs to which. Use the + button (Add/Create a New Node) to add a node and the chain-link button to instance a saved scene (Chapter 2).

The FileSystem dock (bottom left)

This panel shows the **files in your project folder**. Initially you will see `icon.svg`. When you save scenes and scripts, they appear here. You can drag files from here into the Inspector or the viewport. This panel is like the file explorer of your project; it shows files under `res://`.

The viewport (center)

The big center area is where you see and arrange your scene. In the 2D workspace it is a flat canvas with a grid. You can zoom with the mouse wheel and pan by holding the middle mouse button (or holding the space bar while dragging). A toolbar along the top of the viewport has tools to **select**, **move**, **rotate**, and **scale** things. The game window is a small blue rectangle in the middle of the canvas, showing what the player will see.

The Inspector (right)

The **Inspector** shows the **properties** of whatever node you have selected. A property is a setting: a position, a color, a piece of text, a size. If the Scene dock answers "what nodes do I have?", the Inspector answers "how is this node set up?" You will spend a lot of time here. Next to the Inspector tab there is a **Node** tab that shows a node's *signals* and *groups*. These come up in Chapter 5 and Chapter 15.

The bottom panel

Along the bottom you will find tabs such as **Output**, **Debugger**, **Audio**, **Animation**, and **Shader Editor**. Some appear only when relevant. Two matter now:

- **Output** displays text messages from your game, including anything you tell it to `print`. Chapter 3 depends on it.
- **Debugger** shows errors and warnings when your game runs, and it lets you inspect a running game. We use it heavily in Chapter 22.

> **Note:** If you accidentally close or rearrange a panel, do not worry. The panels can be moved and resized by dragging their edges, and **Editor > Editor Layout** offers ways to save or restore layouts. In an emergency, closing and reopening the editor usually resets what is broken.

Where do I get help?

- Press **F1** or use the **Help** menu to open the built-in **offline documentation** for every class. Select a word in a script and press Ctrl+click (Cmd+click on macOS) to jump to its documentation. The docs ship with the editor, so you can read them without internet.
- The official online manual is at https://docs.godotengine.org/en/stable/ and is the first place to look when you are stuck.

> **Try it:** Click each of the four workspace buttons (2D, 3D, Script, Game or Asset Store) and see what happens to the center of the screen, then go back to 2D. The editor does not change your project when you just look around.

1.6 Building your first scene

We now build a tiny but real scene: a dark background, the Godot icon, and a greeting. You will create nodes, change properties in the Inspector, and save. Do not worry about understanding every detail; Chapter 2 revisits them slowly. Right now we want the satisfaction of making something run.

Step 1: Create a root node

In the **Scene** dock you see buttons for creating a root node. A **root node** is the top node of a scene, the one everything else hangs from.

1. Click **2D Scene**. Godot creates a node named `Node2D` at the top of the tree. A `Node2D` is the simplest node that has a position in 2D space, so it is a good container for flat things.
2. Click the node's name once, then press **F2** (or double-click the name) to rename it. Type `Main` and press Enter.

Step 2: Add a background

1. With `Main` selected, click the + button at the top of the Scene dock (Add/Create a New Node).
2. A dialog opens with a long list of node types and a search box. Type `ColorRect`, choose **ColorRect**, and click **Create**.
3. `ColorRect` fills a rectangle with one color. Rename it `Background`.
4. In the **Inspector**, find the **Color** property and click the colored box. A color picker opens; pick a dark blue-gray.
5. Set the size. In the Inspector, expand the **Layout** section, then **Transform**, and set **Size** to `640` by `360`. (We will make the window the same size in Step 6.)

Step 3: Add the icon

1. Select `Main` again, then add another child node. Search for `Sprite2D` and create it. A **Sprite2D** displays a picture. Rename it `Icon`.
2. Look in the **FileSystem** dock for `icon.svg`. **Drag** it onto the empty **Texture** property in the Inspector. The Godot logo appears in the viewport.
3. Drag the icon in the viewport so it sits near the upper middle of the dark rectangle, or set **Position** in the **Transform** section of the Inspector to `320, 150`.
4. Make it larger: set **Scale** to `1.5, 1.5`.

Step 4: Add the greeting

1. Select `Main`, add a child node of type **Label**, and rename it `Greeting`. A **Label** shows text.
2. In the Inspector, type `Hello, Godot!` into the **Text** property.
3. Move the label under the icon and, if you like, enlarge the text: open **Theme Overrides > Font Sizes** and set **Font Size** to `28`.

Your Scene dock should now look like this:

```
Main (Node2D)
├── Background (ColorRect)
├── Icon (Sprite2D)
└── Greeting (Label)
```

> **Why it works:** Nodes are drawn in tree order: nodes lower in the list are drawn on top of nodes higher in the list. That is why `Background` comes first. If you put it last, the dark rectangle would cover the icon and the text.

Step 5: Save the scene

Press **Ctrl+S** (Cmd+S on macOS) or choose **Scene > Save Scene**. A file dialog appears. Name the file `main.tscn` and save it in the project folder (`res://`). It now shows up in the FileSystem dock.

The `.tscn` extension means "text scene." It is a plain text file. You may open it in any text editor if you are curious. Here is an excerpt of what ours looks like. You never have to write this by hand, but it is reassuring that nothing is hidden:

```
[node name="Main" type="Node2D"]

[node name="Background" type="ColorRect" parent="."]
color = Color(0.13, 0.15, 0.22, 1)
```

```
[node name="Icon" type="Sprite2D" parent="."]
position = Vector2(320, 150)
scale = Vector2(1.5, 1.5)
```

(This is a simplified excerpt, which is why it omits some lines that Godot would write.)

Step 6: Make the window the right size

Our background is 640 by 360 pixels, but the default game window is larger.

1. Open **Project > Project Settings**.
2. In the left-hand list, expand **Display** and click **Window**.
3. Set **Viewport Width** to `640` and **Viewport Height** to `360`. If you cannot see these options, turn on **Advanced Settings** with the toggle at the top right of the dialog.
4. Close the dialog.

1.7 Running your game

At the top right of the editor are the run buttons. Figure 1.5 shows them.



Figure 1.5: The run controls. Play (F5) runs the whole project. Run Current Scene (F6) runs just the scene you are editing.

Button	Shortcut	What it does
Run Project	F5	Starts the game from its main scene.
Pause	F7	Pauses the running game.
Stop	F8	Stops the running game.
Run Current Scene	F6	Runs only the scene you are editing, ignoring the main scene.
Run Specific Scene	Ctrl+Shift+F5	Lets you choose a scene to run.

(Shortcuts can change between versions or systems; the labels on the buttons and the **Debug** menu always show the current ones.)

Setting the main scene

Press **F5**. Because you have not yet told Godot which scene is the starting scene, it shows a **Please Confirm...** dialog saying that no main scene has ever been defined and asking whether to select one. Click **Select Current** (it is only available when a saved scene is open). This stores `res://main.tscn` as the **main scene**. You can change it later in **Project > Project Settings > Application > Run > Main Scene**.

A new window opens showing your dark background, the icon, and the text **Hello, Godot!** This is a real, separate program running your project. You have just made your first Godot game, even if all it does is greet the player.



Figure 1.6: The finished Chapter 1 project running. A plain window with a background, an icon, and a label.

Close the game window, or click **Stop** in the editor, to end it.

Run versus edit

It is worth pausing on this. There are two different modes:

- **Editing**: you arrange scenes and change properties. Nothing moves.
- **Running**: the engine starts the game loop on a copy of your project. Changes you make in the editor while the game runs do not (generally) change the running copy, and changes you make to the running game are lost when it stops.

Whenever you want to see the result of an edit, run the project again. Running is cheap and fast; get into the habit of running after every small change. That is how professionals work, too: change a little, test a little.

> **Try it:** Change the text in the Label to your own name, press **F5**, and see the change. Then try **F6** (Run Current Scene) and notice that it does the same thing here, because the scene you are editing is also the main scene.

What happens when you press Play?

Behind the scenes, Godot starts a second copy of the engine, loads your `project.godot`, finds the main scene, builds the nodes in memory, and starts the game loop from Figure 1.2. The editor and the running game are separate programs, which is why a crash in your game does not take down the editor.

1.8 Habits that save you later

A few habits cost nothing now and prevent grief later.

- **Save often.** Press Ctrl+S after each step. Godot warns you before closing with unsaved changes, but a power cut does not.
- **One project per folder.** Never nest one project inside another.

- **Use clear names.** `main.tscn`, `player.gd`, `coin.tscn`. Lowercase letters, numbers, and underscores or hyphens make file names friendly across all operating systems.
- **Move and rename files inside Godot.** If you rename or move a file in your computer's file explorer, Godot may lose track of references to it. Use the FileSystem dock instead; it updates everything that points to the file.
- **Keep backups.** Copy your project folder to another place (or use Git, Chapter 22) from time to time.
- **Read the red text.** When Godot shows an error, read the message. It is almost always trying to tell you exactly what is wrong.

Project 1: Hello, Godot

Goal: create a new project, build a one-scene "game" with a background, an icon, and a greeting, and run it. Time: about 30 minutes.

What you need: Godot 4 installed (section 1.3).

Final file list (also provided in the book's companion folder `projects/ch01-what-is-godot/`):

```
ch01-what-is-godot/
├── project.godot
├── icon.svg
└── main.tscn
```

Steps

1. Start Godot to open the Project Manager.
2. Click **Create**. Name the project `hello-godot`, choose a folder, keep the default renderer (or Compatibility on an old computer), and create it.
3. In the Scene dock, click **2D Scene**. Rename the root node to `Main`.
4. Add a **ColorRect** child named `Background`. Set its color to a dark blue-gray and its size to 640 by 360.
5. Add a **Sprite2D** child named `Icon`. Drag `icon.svg` from the FileSystem dock to its Texture property. Set Position to `320, 150` and Scale to `1.5, 1.5`.
6. Add a **Label** child named `Greeting`. Set its Text to `Hello, Godot!`, enlarge the font size to 28, and position it below the icon.
7. Check that the Scene dock matches the tree in section 1.6, with `Background` first.
8. Save the scene as `main.tscn` (Ctrl+S).
9. Open **Project > Project Settings > Display > Window** and set the viewport to 640 by 360.
10. Press **F5**. When asked, choose **Select Current** to make `main.tscn` the main scene.
11. Admire your running game. Close it.
12. **Extension:** change the text, the colors, or the scale of the icon. Run after every change.

Expected result: a 640 by 360 window with a dark background, the icon in the upper middle, and the words "Hello, Godot!" underneath. No errors appear in the Debugger. When you close the window, you are back in the editor.

If something is wrong: if you see only a gray window, check that `Background` is *above* the other nodes in the tree. If the text does not appear, check that the Label is a child of `Main` and has text.

Chapter summary

- A **game engine** supplies the common parts of a game (drawing, input, sound, physics, file handling) so you only build what is unique to your game.
- Games run a **game loop**: input, update, draw, repeated many times a second; each trip produces one **frame**.
- **Godot** is a free, open-source engine with strong 2D and 3D support. This book uses GDScript and Godot 4.7.
- Install Godot by downloading the **standard** version from godotengine.org, extracting it, and running it. There is no installer.
- The **Project Manager** creates and opens projects. A project is a folder containing `project.godot`, and Godot calls its root `res://`.
- The editor's main areas are the **Scene dock**, **FileSystem dock**, **viewport**, **Inspector**, and **bottom panels** such as Output and Debugger.
- A game is built from **nodes** arranged in a tree and saved as a **scene** (`.tscn`). Nodes lower in the tree draw on top.
- Press **F5** to run the project and **F6** to run the current scene. The first time you will pick a **main scene**.

Review questions

1. In your own words, what is a game engine? Name three jobs it does for you.
2. What are the three steps of the game loop, in order?
3. If a game runs at 60 frames per second, roughly how much time does the engine have to finish one frame?
4. True or false: you must pay Godot a fee for each game you sell.
5. Which of these should you download for this book? (a) The .NET version, (b) the standard version, (c) a version labeled 3.x, (d) an unofficial copy from a file-sharing site.
6. What does `res://` mean, and what file marks a folder as a Godot project?
7. What is the difference between the Scene dock and the Inspector?
8. Multiple choice: pressing **Remove** in the Project Manager (a) deletes the project's files, (b) takes the project off the list but keeps the files, (c) exports the game, (d) closes the editor.
9. Why must the `Background` node be listed above the `Icon` node in the scene tree for the project in this chapter?
10. What is the difference between **Run Project** (F5) and **Run Current Scene** (F6)?

Exercises

1. **Practice (easy):** Change the Label text to your own name and the background to a different color. Run the project and take a screenshot.
2. **Practice (easy):** Open the Help menu and find the built-in documentation page for `Sprite2D`. Write down two properties it lists.
3. **Apply (medium):** Add a second `Sprite2D` using the same `icon.svg` and place it in the lower right corner at a smaller scale (0.5). Run the project. Then move it so it sits *behind* the first icon by reordering it in the Scene dock (drag it up in the tree). What changes?
4. **Apply (medium):** Create a second project, named `scratch`, using the **Compatibility** renderer. Open the project settings, find where the renderer is shown, and compare with your first project. Then remove `scratch` from the Project Manager list without deleting its files, and import it again.

5. **Challenge (hard):** Open `main.tscn` in a plain text editor (Notepad, TextEdit, or gedit) while the editor is closed. Find the line that names your label's text and edit it by hand. Open the project again and confirm the change. What does this tell you about how Godot stores scenes?

Common mistakes

- **Nothing happens when I double-click the zip file's program (Windows).** The program is still inside the zip. *Cause:* running from the compressed folder. *Fix:* extract everything first, then run the `.exe`.
- **A black terminal window opens beside the editor.** *Cause:* you started the `_console` version. *Fix:* use the executable without "console" in the name for everyday work.
- **"No main scene has been defined" appears when I press F5.** *Cause:* a new project has no starting scene yet. *Fix:* click Select Current (or set it under Project Settings > Application > Run > Main Scene).
- **I see only a gray window when I run.** *Cause:* the background node is drawn on top of other nodes, or nothing is added to the scene. *Fix:* reorder nodes so the background is first, and confirm the nodes are children of `Main`.
- **I renamed a file in my file explorer and the scene broke.** *Cause:* Godot references files by path. *Fix:* rename and move files inside the FileSystem dock; if it already happened, move the file back to its old name.
- **The editor shows a graphics or renderer error at startup.** *Cause:* old hardware or drivers that do not support the Forward+ renderer. *Fix:* create the project again with the Compatibility renderer, and update your graphics drivers.

Further reading

- Introduction to Godot: https://docs.godotengine.org/en/stable/getting_started/introduction/introduction_to_godot.html
- Overview of Godot's key concepts: https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html
- Project Manager: https://docs.godotengine.org/en/stable/tutorials/editor/project_manager.html
- Command line tutorial: https://docs.godotengine.org/en/stable/tutorials/editor/command_line_tutorial.html

Chapter 2: Nodes and Scenes: The Building Blocks

Learning objectives

By the end of this chapter you will be able to:

- Explain what a node is and why Godot builds every game out of nodes.
- Read and build a node tree, using the words parent, child, sibling, and root correctly.
- Predict how a parent's position, rotation, and scale affect its children.
- Save a tree of nodes as a scene and create several copies of it by instancing.
- Use the Inspector to change properties, reset them, and understand where they come from.
- Organize a project with sensible node names, scene names, and file names.

Key terms

- **Node**: the smallest building block in Godot. Each node does one job, such as drawing a picture, showing text, or playing a sound.
- **Class**: the type of a node (for example `Sprite2D` or `Label`). A class defines what properties and abilities a node has.
- **Inheritance**: when one class builds on another and receives everything the other has. A `Sprite2D` is a kind of `Node2D`, which is a kind of `Node`.
- **Property**: a named setting on a node, such as `position`, `color`, or `text`.
- **Node tree**: nodes arranged in a hierarchy, where each node has one parent (except the top one) and can have many children.
- **Parent / child / sibling**: a parent holds children; children that share the same parent are siblings.
- **Root node**: the single top node of a scene or tree.
- **Scene**: a node tree saved to a file (`.tscn`) so it can be opened, run, and reused.
- **Instance**: one live copy of a scene placed inside another scene.
- **Transform**: a node's position, rotation, and scale together.
- **Scene tree**: the tree of every node in the running game, with the main scene attached under a built-in top node.

2.1 Everything is a node

In Chapter 1 you made a scene with a `ColorRect`, a `Sprite2D`, and a `Label`. Each of these is a **node**. A node is a small object with one job, and Godot ships with hundreds of them. Here are a few you will meet in this book:

Node	Its job
`Node`	Does nothing visible. The plain base for everything else. Useful as a container or to hold a script.
`Node2D`	Has a position, rotation, and scale in 2D space.
`Sprite2D`	Draws a picture (a texture) in 2D.
`Label`	Draws text.
`ColorRect`	Fills a rectangle with one color.
`Polygon2D`	Fills a shape you define with a color.

Node	Its job
`Timer`	Waits and then announces that time is up.
`AudioStreamPlayer`	Plays a sound.
`Camera2D`	Decides which part of the world the player sees.

Think of nodes like the different kinds of building blocks in a toy set. One brick is a window, one is a wheel, one is a flat plate. None of them is a toy by itself. You combine them.

What a node has

Every node has three things:

1. **Properties**: settings you can read and change, like `position` or `text`. You change them in the Inspector or from code.
2. **Abilities (functions)**: things the node can do, such as `play()` for an audio player. In code these are called *methods* or *functions*; Chapter 4 explains them.
3. **Signals**: announcements the node can make, like "I was clicked" or "time is up." Chapter 5 introduces signals and Chapter 15 goes deeper.

You do not need to remember individual node types. You need a habit: when you want something to happen in your game, ask "which node does that job?" Use the **Create New Node** dialog's search box, or the built-in help, to find out.

Types and inheritance

Node types are organized in families. Each type builds on a simpler one, receiving all of its properties and then adding its own. This is called **inheritance**.

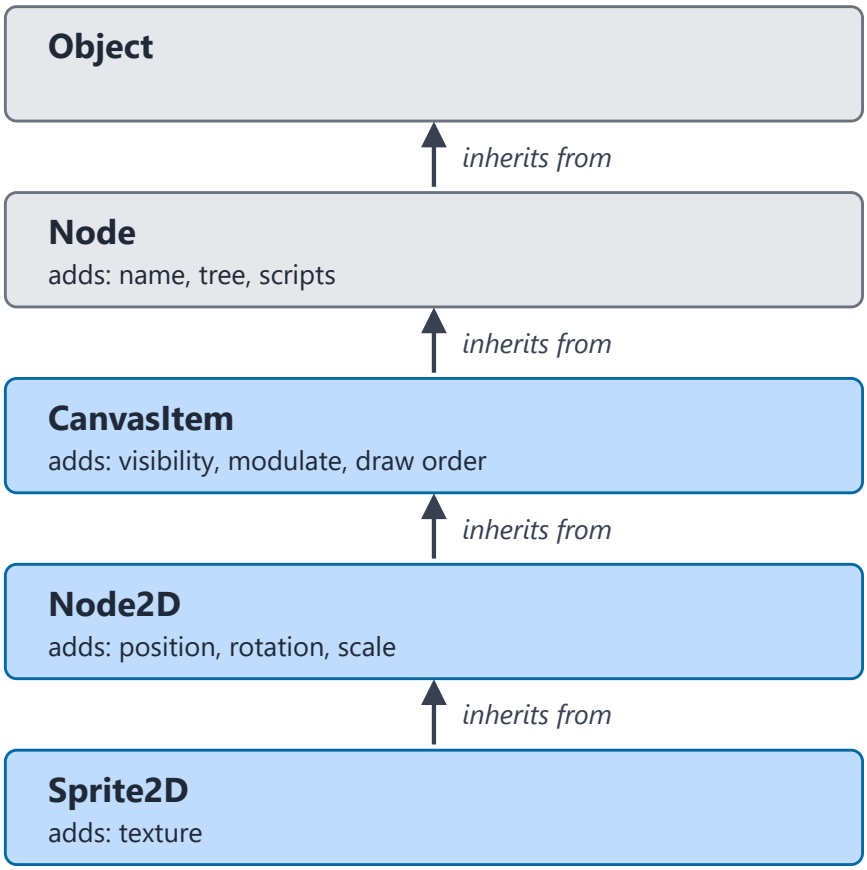


Figure 2.1: A `Sprite2D` inherits everything from `Node2D`, `CanvasItem`, `Node`, and `Object`.

Read the chain from top to bottom:

- `Object` is the root of almost all Godot types.
- `Node` adds a name, the ability to live in a tree, and the ability to run scripts every frame.
- `CanvasItem` adds everything related to *being drawn* in 2D (or in an interface): visibility, color tint, draw order.
- `Node2D` adds a **position**, **rotation**, and **scale** in 2D space.
- `Sprite2D` adds a **texture** and the logic to draw it.

When you select a `Sprite2D` in the editor, the Inspector shows properties from every level of that chain, and the sections are arranged from the most specific (Sprite2D) at the top to the most general below. That is why a `Sprite2D` has a Transform section (from `Node2D`) and a Visibility section (from `CanvasItem`).

> **Why it works:** Inheritance saves enormous effort. Anything you learn about `Node2D` positioning applies to `Sprite2D`, `Polygon2D`, `Camera2D`, and every other 2D node, since they all inherit from it.

Color-coded families

Look at the icons in the Scene dock. Godot colors them by family:

- **Blue** icons: 2D nodes (those under `Node2D`).
- **Red** icons: 3D nodes (those under `Node3D`). These are the subject of Chapter 19.
- **Green** icons: `Control` nodes, used for user interfaces such as buttons, labels, and menus (Chapter 14).
- **White or gray** icons: plain nodes and other helpers.

Notice that `Label` and `ColorRect` have green icons: they are **Control** nodes, a separate family built for interface layout. In Chapter 1 we placed them inside a `Node2D` and positioned them by hand, which works fine for a simple scene. Chapter 14 shows when to use Control nodes the way they were designed to be used.

> **Try it:** Create a new scene, add a `Node2D` root, and open the Create New Node dialog (the + button). Type `Sprite` and look at the results. Click a result once (without creating it) and read the description at the bottom of the dialog. Do the same for `Timer`.

2.2 The node tree

Nodes alone do little. The power comes from connecting them in a **tree**.

A tree in computer science is a family-tree-like diagram turned upside down: one node at the top and branches below it. In Godot:

- Every node has **at most one parent**.
- A node can have **any number of children**.
- Children of the same parent are **siblings**.
- The one node at the top, which has no parent inside the scene, is the **root**.

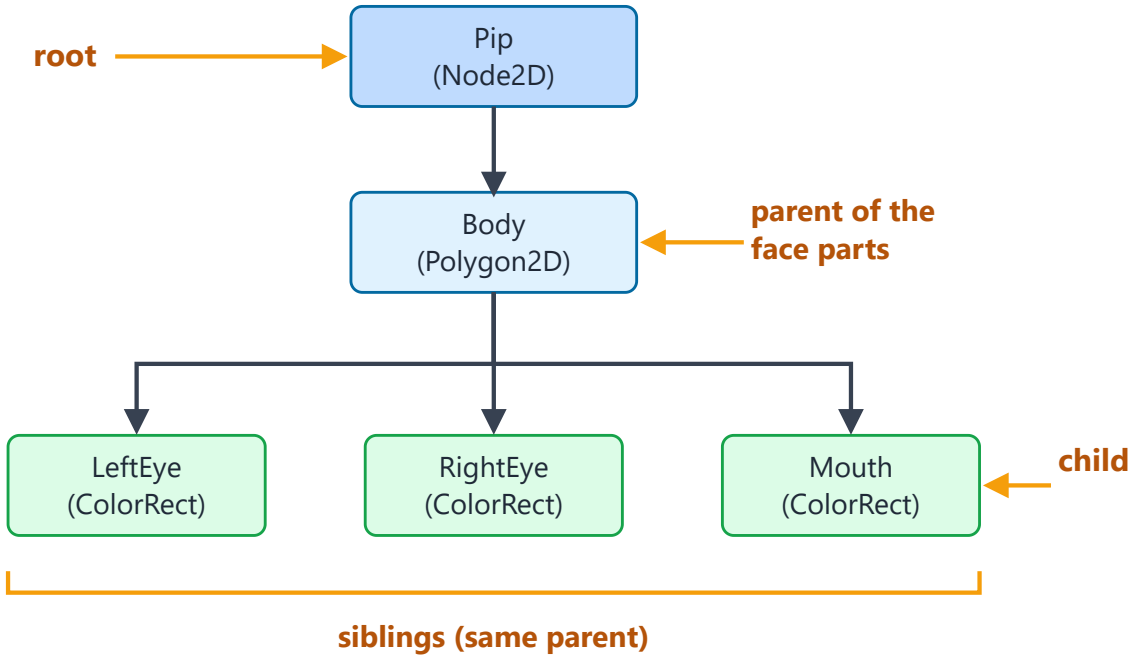


Figure 2.2: The tree for Pip, the hero of this book. `Body` is a child of `Pip` and the parent of the three face parts. `LeftEye`, `RightEye`, and `Mouth` are siblings.

In the Scene dock the same tree is drawn as an indented list:

```
Pip (Node2D)
└── Body (Polygon2D)
    ├── LeftEye (ColorRect)
    ├── RightEye (ColorRect)
    └── Mouth (ColorRect)
```

Each level of indentation means "is a child of the node above."

Why a tree?

Why not a flat list of objects? Because things in games belong together. Pip's eyes belong to Pip. When Pip moves, his eyes must move too; when Pip is deleted, so should his eyes. The tree captures "belongs to," and Godot uses it to do three valuable things.

1. Children move with their parent (transform inheritance). A node's position is measured *relative to its parent*. If `Body` is at `(0, 0)` relative to `Pip`, and `LeftEye` is at `(-14, -12)` relative to `Body`, then moving `Pip` to the right moves everything. Rotating `Pip` rotates everything. Scaling `Pip` to 2 doubles everything.

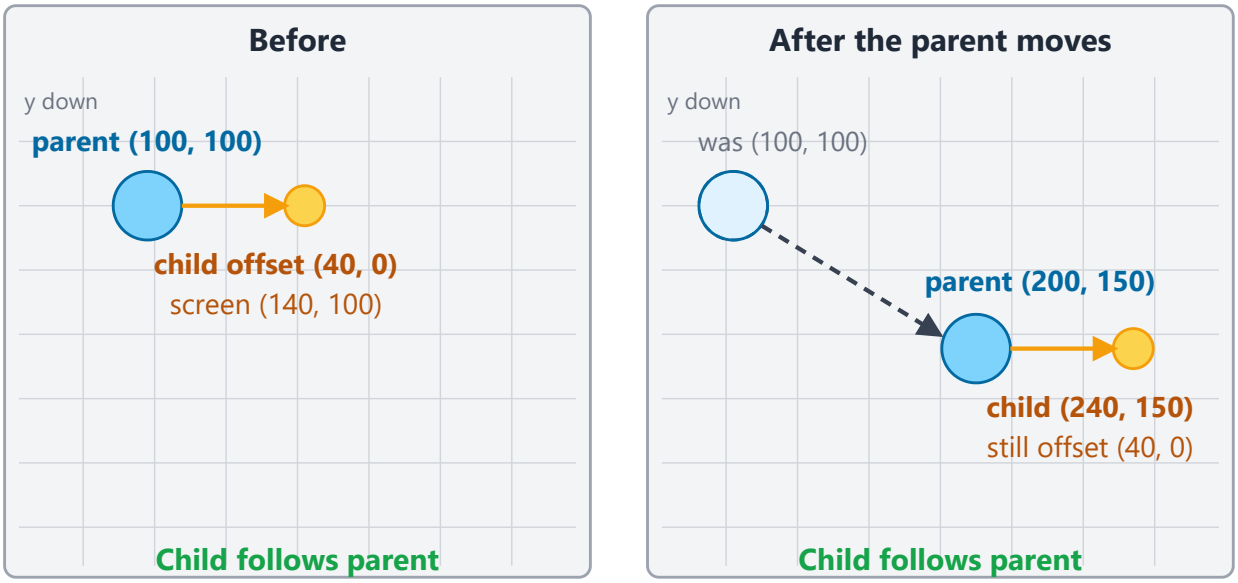


Figure 2.3: A child's position is an offset from its parent. When the parent moves, the child comes along.

This is why you almost never place every sprite in a game at an absolute screen position. You build a character as a small tree, and then you only ever move the top node.

2. Deleting a parent deletes its children. When a node leaves the game, its whole branch goes with it.

3. Draw order follows the tree. As you saw in Chapter 1, siblings are drawn from top of the list to bottom of the list, so lower nodes appear in front. A child draws after (on top of) its parent.

Names and paths

Each node has a **name**, shown in the Scene dock. Rules:

- Siblings must have **different names**. If you add a second `Sprite2D` next to the first, Godot names it `Sprite2D2` automatically.
- Names are case sensitive: `Body` and `body` are different.
- Avoid dots, slashes, colons, and at-signs in names, as these have special meanings.

Because names are unique among siblings, every node in a scene can be reached through a **path**, written like a folder path:

- `Body` from `Pip` means the child named Body.

- `Body/LeftEye` from `Pip` means the left eye, inside Body.
- `..` means "go up to the parent." From `LeftEye`, the path `../Mouth` points to its sibling `Mouth`.

Scripts use paths to find other nodes (`$Body/LeftEye`, Chapter 5), so good, short, meaningful names pay off later. Name nodes for *what they are in the game*, not what type they are: `Player`, `ScoreLabel`, `CoinTimer`.

The root node's special role

Whatever type you give the root node tells the world what the scene *is*. A scene for a character that moves around in 2D usually has a 2D-type root. A scene for a menu usually has a Control-type root. Pip's root is a plain `Node2D` because Pip only needs a position.

You can change a node's type later (right-click the node and choose **Change Type...**), but it is easier to choose well at the start.

2.3 Scenes

A **scene** is a node tree that has been **saved to a file**. That is all. The file ends in `.tscn` (text scene). You made one in Chapter 1.

Scenes in Godot play several roles at once:

- A scene can be a **whole level**.
- A scene can be a **character**, an **enemy**, a **coin**, a **door**, or a **bullet**.
- A scene can be a **menu** or a **single button**.
- A scene can be your **entire game**, built from smaller scenes.

There is no special "level" or "prefab" or "object" type: it is always a scene. A scene can contain other scenes. This uniform design is one of the nicest things about Godot, because once you understand scenes you understand how the whole engine is organized.

Opening, saving, and tabs

- **Scene > New Scene** (or the + tab at the top of the viewport) starts a new tree.
- **Scene > Save Scene** (Ctrl+S) writes the current tree to its `.tscn` file. The first time, it asks for a name and location.
- **Scene > Open Scene...** opens a saved file. You can also **double-click a `.tscn` file in the FileSystem dock**.
- Each open scene gets a **tab** above the viewport. A star or dot on the tab means unsaved changes.

> **Note:** Saving the scene does not save the project's settings separately; Project Settings are written when you close that dialog. Saving scripts is separate too (Ctrl+S in the Script workspace, or Scene > Save All).

The scene tree at run time

When the game runs, Godot builds one big tree for the whole running game. At the very top is a built-in node called `root` (a `Window`). Your main scene is attached beneath it as a child. When your game later loads other scenes (a new level, a menu), they are attached into this same tree. Godot calls this tree the **scene tree**, and it is what the game loop of Chapter 1 visits each frame.

If you press F5 and look at the Scene dock while the game runs, you will see two buttons, **Local** and **Remote**. Local is your editable scene; **Remote** shows the live tree of the running game. It is a good way to see what exists right now. Chapter 22 uses it for debugging.

2.4 Instancing: one scene, many copies

Here is the idea that makes scenes powerful. Suppose you want a room with three characters like Pip. You could build Pip three times, but then if you change Pip's color you have to change it three times. Instead you build Pip **once**, save him as `pip.tscn`, and put *instances* of that scene into the room.

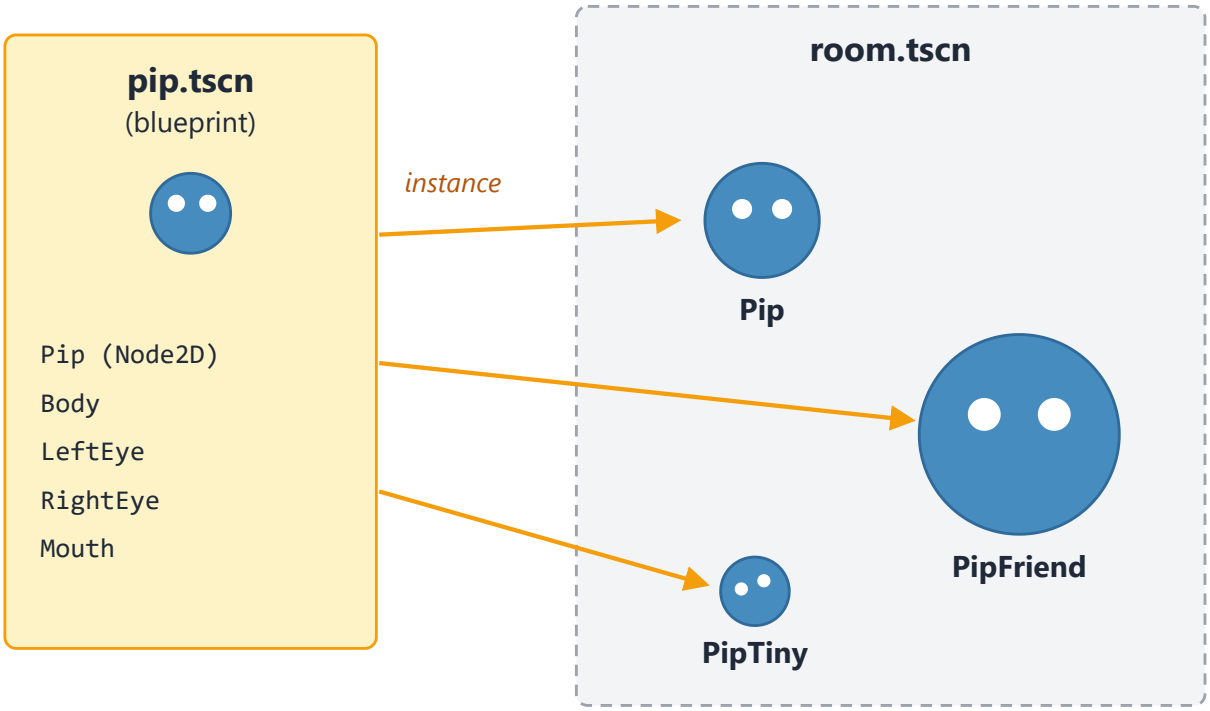


Figure 2.4: One scene file, three instances. Each instance has its own position, scale, and rotation, but all share the same design.

Think of a scene file as a **blueprint** or a cookie cutter, and an **instance** as a cookie. Press the cutter three times and you have three cookies. Change the cutter, and every future cookie, including the ones you already placed in a Godot scene, changes shape. (Real cookies would not, but Godot's do. When you edit `pip.tscn` and save, every room that uses Pip shows the new Pip.)

How to instance a scene

1. Open (or create) the scene that will **contain** the copies, for example `room.tscn`.
2. Select the node that should be the parent.
3. Click the **chain-link** button at the top of the Scene dock (**Instantiate Child Scene**), choose `pip.tscn`, and confirm. Or **drag `pip.tscn` from the FileSystem dock** into the Scene dock or the viewport.
4. Repeat for every copy you want.

An instance shows up in the Scene dock with a small **scene icon** (a clapperboard) to its right. Its children are hidden by default, because they belong to `pip.tscn`, and you should edit them there. To open the original, click that icon, or right-click the instance and choose **Open in Editor**.

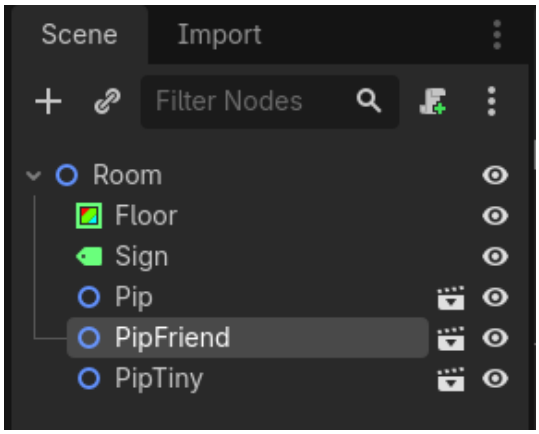


Figure 2.5: Instances in the Scene dock. The clapperboard icon marks a node that is an instance of another scene.

What can differ between instances?

Each instance is independent in the properties that belong to *this placement*. In our room, the three Pips differ in:

- **position** (where they stand),
- **scale** (size),
- **rotation**.

The *inside* of each Pip (its eyes, its color) comes from the blueprint. If you want to change the inside of just one instance, right-click it and choose **Editable Children**. That lets you override properties of one copy, and Godot remembers the override. Use this sparingly; usually it is cleaner to change the blueprint, or to give the scene a script setting.

> **Warning:** Don't build a scene by instancing a scene inside *itself*. A scene cannot contain a copy of itself because it would repeat forever. Godot stops you.

Why this matters for real games

Instancing is how almost every game is built:

- One `coin.tscn`, instanced thirty times across a level.
- One `slime.tscn`, instanced wherever an enemy should be.
- One `hud.tscn` (the score display), instanced into every level.
- In Chapter 9 your scripts will instance scenes while the game runs, to spawn coins out of thin air.

Each scene is small, focused, and can be tested alone with **Run Current Scene (F6)**. That last point matters most for beginners: when something breaks, you can open that one small scene, run it by itself, and find the cause quickly.

> **Why it works:** Small scenes keep your game understandable. Instead of one giant tree with 500 nodes, you have a few dozen scenes, each with 3 to 20 nodes. You always know where to look.

2.5 The Inspector in depth

You used the Inspector in Chapter 1. Now we look closely, since you will spend a large share of your Godot life there.

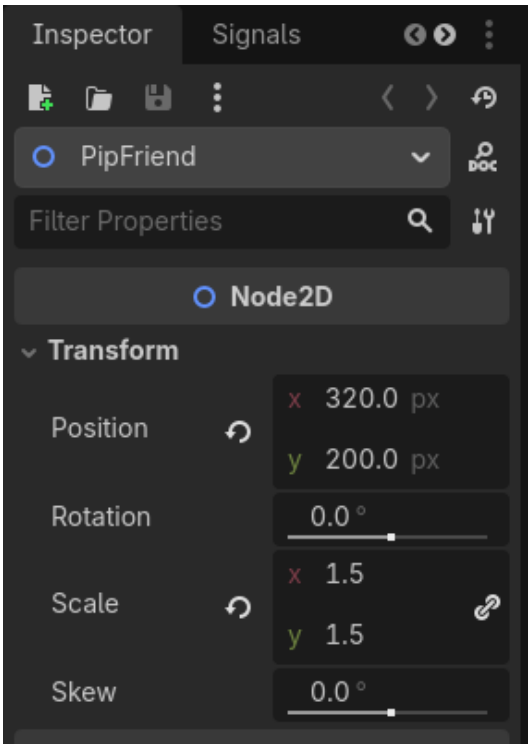


Figure 2.6: The Inspector showing a node's Transform section. A small circular arrow appears beside a property you have changed from its default.

Reading a property

Each row has a **name** on the left and an **editor** on the right. The editor depends on the type of data:

- A **number** appears as a box you can type into or drag left and right.
- A **text** value appears as a text box.
- A **color** appears as a colored swatch; click it to open a picker.
- A **checkbox** appears for true/false values.
- A **vector** (like a 2D position) appears as two boxes, `x` and `y`.
- A **resource** (like a texture) appears as a slot where you can drag a file or click to create a new one.

When you have changed a value away from its default, a small **revert arrow** appears next to it. Click it to put the default back. This is a quick way to undo a change you do not remember making.

Sections and groups

Properties are grouped into collapsible sections such as **Transform**, **Visibility**, **Ordering**, and **Texture**. Click a section title to open or close it. You can also type in the search box at the top of the Inspector to filter properties by name, which is faster than scrolling.

Transform: position, rotation, scale

Every `Node2D` has these three properties, which together form its **transform**:

- **Position** is a pair of numbers `(x, y)` measuring where the node is, relative to its parent. In Godot's 2D world, `x` grows toward the **right** and `y` grows **downward**. This surprises many beginners; Chapter 6 explains it fully. The point `(0, 0)` is the top-left of the screen for a root-level node with no camera.

- **Rotation** turns the node around its own origin. The Inspector shows **degrees** (a full turn is 360). Inside scripts, rotation is stored in **radians** (a full turn is about 6.28). The editor converts for you, which is a feature worth knowing about when you read code later.
- **Scale** stretches the node: `(1, 1)` is normal, `(2, 2)` is double size, `(0.5, 0.5)` is half size, and a negative number flips the node.

The node's own origin point, `(0, 0)` inside the node, is where rotation and scale pivot. For Pip, we draw the body around `(0, 0)` so that he spins around his center.

> **Try it:** Open the Chapter 2 project's `pip.tscn` (or any Node2D) and change `Rotation` on the root to 45. Now select `Body` and change its Rotation to 45 as well. The face turns twice as far. Why? (Hint: transforms combine from parent to child.)

Visibility and draw order

Inside the **CanvasItem** section you will find:

- **Visible**: a checkbox. Unchecking hides the node and *all* its children.
- **Modulate**: a color tint multiplied over the node's colors. White means no change. Use a half-transparent white to fade something.
- **Z Index** (under **Ordering**): a number that overrides tree order. Higher numbers draw on top. Use it rarely; the tree order is usually enough.

You can also hide a node in the editor only, by clicking the **eye** icon beside it in the Scene dock. For a quick test this is more convenient than the Visible checkbox, but be careful: the eye icon does not change the running game.

Resources inside properties

Some properties hold more than a number: a texture, a font, a shape. These are called **resources**. A resource is a piece of data that can be saved in a file and shared. Dragging `icon.svg` onto the Texture slot of a `Sprite2D` put a texture resource there. Chapter 16 covers resources in depth. For now, know that a resource slot works by dragging a file in, or by clicking the slot's dropdown to create a new one, such as `New PlaceholderTexture2D` (a colored rectangle, handy as stand-in art).

2.6 Working with the Scene dock

A handful of commands make building scenes quick. Most are available by **right-clicking** a node in the Scene dock.

Action	How
Add a child	Select the parent, click + (Add/Create a New Node)
Instance a scene as a child	Click the chain-link button, or drag a `.tscn` in
Rename	Select, press **F2**, or double-click the name
Duplicate	Select, press **Ctrl+D** (Cmd+D on macOS)
Delete	Select, press **Delete**
Reorder or re-parent	Drag a node up, down, or onto another node

Action	How
Change a node's type	Right-click, **Change Type...**
Save a branch as its own scene	Right-click, **Save Branch as Scene**
Open an instance's original	Right-click, **Open in Editor**
Allow editing inside an instance	Right-click, **Editable Children**

Save Branch as Scene is especially useful. You build a few nodes inside a bigger scene, realize that group should be reusable, right-click its top node, and save it as its own file. Godot replaces the branch with an instance of the new scene. Many professional projects grow this way: start big and messy, split out the reusable pieces.

Press **Ctrl+Z** to undo almost anything you do in the editor, including deleting nodes. Use it freely.

2.7 Keeping a project organized

Even small projects become messy unless you adopt a few rules from the start.

Name nodes by purpose. `Player`, `Door`, `ScoreLabel` tell you more than `Sprite2D3`. Use PascalCase (each word capitalized, no spaces) for node names, which is Godot's convention.

Name files in lowercase. `pip.tscn`, `room.tscn`, `main_menu.tscn`. Lowercase with underscores works on every operating system. (Windows is case-insensitive about file names but Linux and the web are not, and a mismatch is a classic bug to be found after export.)

Group files in folders. Once a project has more than a handful of files, make folders in the FileSystem dock (right-click, **New Folder...**), for example `scenes/`, `scripts/`, `art/`. Always move files inside the FileSystem dock so references update.

One scene, one purpose. If a scene has two unrelated jobs, split it.

Pick the right root. The root should be the most general type the scene needs. Do not use a `Sprite2D` as a root and then try to add a collision shape and a hundred children; use `Node2D` or a body type.

The projects in this book are small enough to keep files at the top level, but each chapter's project shows the naming habits above.

2.8 Looking inside a scene file

Scenes are plain text, and reading one is the best way to see that the tree is not magic. Here is `room.tscn` from this chapter's project, slightly trimmed:

```
[gd_scene load_steps=2 format=3]

[ext_resource type="PackedScene" path="res://pip.tscn" id="1_pip"]

[node name="Room" type="Node2D"]

[node name="Floor" type="ColorRect" parent="."]
offset_right = 640.0
offset_bottom = 360.0

[node name="Pip" parent="." instance=ExtResource("1_pip")]
position = Vector2(160, 200)

[node name="PipFriend" parent="." instance=ExtResource("1_pip")]
```

```
position = Vector2(320, 200)
scale = Vector2(1.5, 1.5)
```

Read it line by line:

- The first line says this is a Godot scene file. You can ignore the numbers.
- `[ext_resource ...]` says "this scene uses another file," here `pip.tscn`. It gets a short label (`1_pip`) so the rest of the file can refer to it.
- `[node name="Room" type="Node2D"]` is the root. It has no `parent=` because nothing is above it.
- `parent="."` means "my parent is the root." A deeper node would say `parent="Body"` or `parent="Body/LeftEye"`; these are the same paths you learned in section 2.2.
- An instance has `instance=ExtResource("1_pip")` instead of a `type`. It does not repeat Pip's insides. It only says "put a copy of that file here," followed by the properties that differ: `position` and `scale`.

Notice what is *missing*: the file lists only values that differ from the defaults. A `Node2D` with position `(0, 0)` has no `position` line. This is also why the revert arrow in the Inspector exists: a property with no line in the file is at its default.

> **Why it works:** Because instances store just a reference plus differences, editing `pip.tscn` instantly affects every scene that points at it. There is exactly one copy of Pip's design on disk.
>
> **Warning:** Edit scene files by hand only when the editor is closed, and only if you are curious. A typo can make a scene fail to load. The editor never needs you to do this. If you ever damage a file, your version-control history (Chapter 22) or a backup is the cure.

Putting the vocabulary together

Here is the same story told with this chapter's words. `Pip` is a **scene** made of four **nodes** arranged in a **tree**, with `Pip` as the **root**. `room.tscn` is another scene whose tree contains three **instances** of the first. When the game runs, Godot assembles everything into one **scene tree**, draws it in tree order, and every child follows its parent's **transform**. If you can explain that paragraph to a friend, you understand the heart of Godot.

> **Try it:** Open `room.tscn` from the companion project in a text editor with Godot closed, find the line with `PipTiny`, and change its `scale` to `Vector2(1, 1)`. Open the project and check that the little Pip grew. Then change it back using the Inspector and see the same file change.

Project 2: Pip's Room

Goal: build Pip, a small hero made from basic shapes, as a reusable scene, then place three instances of him in a room. Time: about 45 minutes.

What you need: Godot 4 and the knowledge from Chapter 1.

Final file list (companion folder `projects/ch02-nodes-and-scenes/`):

```
ch02-nodes-and-scenes/
├── project.godot
├── icon.svg
├── pip.tscn
└── room.tscn
```

Part A: build Pip

1. Create a new project named `pips-room`. In **Project > Project Settings > Display > Window**, set the viewport to 640 by 360 (as in Chapter 1).
2. Create a new scene with a **2D Scene** root. Rename the root `Pip`.

3. Add a child of type **Polygon2D** and name it `Body`. In the Inspector, set **Color** to a light blue. Expand the **Polygon** property (click its arrow). Set the array **Size** to `8`, and fill in the eight points as `(x, y)` pairs, which make a round-ish octagon around the origin:

Point	x	y
0	32	0
1	22.6	22.6
2	0	32
3	-22.6	22.6
4	-32	0
5	-22.6	-22.6
6	0	-32
7	22.6	-22.6

   (Polygon2D shows a hint of its shape in the viewport as you enter points. If the editor shows the array as a single list of numbers instead of pairs, enter them in order `x, y, x, y, ...`.)

4. Add three **ColorRect** children **to** `Body` (select `Body` first). Name them `LeftEye`, `RightEye`, and `Mouth`, and give each a dark color. Use these Layout > Transform values:

Node	Position	Size
`LeftEye`	-14, -12	8, 8
`RightEye`	6, -12	8, 8
`Mouth`	-8, 8	16, 3

5. Your tree should now match Figure 2.2. Save the scene as `pip.tscn`.

6. Press **F6** (Run Current Scene) to see Pip alone. He appears at the top-left corner, partly cut off, because his origin is at `(0, 0)`. That is fine; we position him in the room.

Part B: build the room

7. Create a new scene with a **2D Scene** root and rename it `Room`.
8. Add a **ColorRect** child named `Floor`, size 640 by 360, with a dark blue-gray color.
9. Add a **Label** child named `Sign` at position `20, 16` with the text `Pip's Room (Chapter 2)`.
10. Drag `pip.tscn` from the FileSystem dock into the Scene dock (onto `Room`). Rename the instance `Pip` and set its position to `160, 200`.
11. Add a second instance, name it `PipFriend`, position `320, 200`, scale `1.5, 1.5`.
12. Add a third instance, name it `PipTiny`, position `480, 200`, scale `0.6, 0.6`, rotation `20` (degrees).
13. Save as `room.tscn` and press **F5**; choose **Select Current** to make it the main scene. You should see three Pips of different sizes.

Part C: prove that instances share a blueprint

14. Open `pip.tscn` (click the clapperboard icon on any Pip). Select `Body` and change its **Color** to a light green. Save (Ctrl+S).
15. Switch back to the `room.tscn` tab. All three Pips are now green. Press **F5** to confirm. Undo your color change if you prefer blue.
16. **Extension:** inside `pip.tscn`, add a fourth child of `Body` named `Cheek` (a small pink `ColorRect`). Save. All three instances gain the cheek instantly.

Expected result: a 640 by 360 window with a dark floor, a caption in the top-left, and three Pips in a row at different sizes (the third slightly tilted). No errors in the Debugger. Changing the blueprint changes every instance.

Chapter summary

- A **node** is a building block with one job. Nodes have **properties**, **functions**, and **signals**.
- Node types are related by **inheritance**: `Sprite2D` is a `Node2D`, which is a `CanvasItem`, which is a `Node`.
- Nodes form a **tree** of parents and children. Children inherit their parent's transform and are deleted with it.
- Siblings must have unique names; scripts and the editor find nodes through **paths** such as `Body/LeftEye`.
- A **scene** is a saved node tree (`.tscn`). Scenes can contain other scenes.
- **Instancing** creates copies of one scene. Editing the blueprint updates every instance.
- The **Inspector** shows a node's properties. A revert arrow appears on changed values. Transform is position, rotation (degrees in the editor), and scale.
- Good habits: name by purpose, lowercase file names, small focused scenes, and move files in the FileSystem dock.

Review questions

1. What three things does every node have?
2. In the inheritance chain `Node` > `CanvasItem` > `Node2D` > `Sprite2D`, which class adds a position, rotation, and scale?
3. True or false: a node can have two parents at the same time.
4. In the tree below, name Mouth's parent, its siblings, and the root.

```
Pip (Node2D)
└── Body (Polygon2D)
    ├── LeftEye (ColorRect)
    ├── RightEye (ColorRect)
    └── Mouth (ColorRect)
```

5. A parent is at position `(100, 100)` and its child has position `(40, 0)`. Where does the child appear on screen? What happens to the child's on-screen position if the parent moves to `(200, 150)`?
6. Why are two siblings not allowed to have the same name?
7. What is the difference between a scene file and an instance of that scene?
8. Multiple choice: after you edit and save `pip.tscn`, what happens to the three Pips in `room.tscn`? (a) Nothing, (b) only the first changes, (c) all three change, (d) the room scene is deleted.
9. In the editor, rotation is shown in ____ , while scripts store it in ____ . (Fill in the blanks.)
10. In Godot 2D, which direction does the `y` coordinate grow: up or down?

Exercises

1. **Practice (easy):** In `room.tscn`, change the scale of `PipFriend` to `(-1.5, 1.5)`. What happens to Pip? Why might negative scale be useful in a game?
2. **Practice (easy):** Draw by hand, on paper, the node tree of a scene for a front door with a door body, a window in the door, and a doorknob. Which node would you choose as the root?
3. **Apply (medium):** Add a fourth Pip to the room by duplicating an existing instance with **Ctrl+D**. Give it a new name and position. Then use **Visible** on its parent to hide it. Does it vanish from the running game?
4. **Apply (medium):** Create `coin.tscn` out of a yellow `Polygon2D` diamond or a `ColorRect`, then instance it five times along a line in the room. Change the color of the blueprint and check that all five follow.
5. **Challenge (hard):** In `room.tscn`, right-click one Pip, choose **Editable Children**, and change the color of that instance's `Body` only. Save and run. Then open `pip.tscn`, change the color again, and see which instances follow and which do not. Write two sentences explaining the result.

Common mistakes

- **My instance is huge/tiny/missing.** *Cause:* the instance's position, scale, or the blueprint's drawing is not where you expected, since the blueprint is drawn around its origin at `(0, 0)`. *Fix:* check Position and Scale on the instance; draw the blueprint around its origin.
- **I changed the blueprint but one copy did not change.** *Cause:* that instance has a local override (for example through Editable Children). *Fix:* revert the property with the revert arrow on the instance.
- **The child ended up in the wrong place in the tree.** *Cause:* you had the wrong node selected when you clicked Add Child. *Fix:* drag the node onto the right parent in the Scene dock, or undo with Ctrl+Z and select the parent first.
- **A node vanished from the game but is still in the tree.** *Cause:* its Visible checkbox is off, or its parent's is, or it is covered by a node drawn after it. *Fix:* check Visible on the node and its parents; check the tree order.
- **Moving a node moved everything inside it.** *Cause:* that is how parents work; children follow. *Fix:* if you only want to move one part, select the child, not the parent.
- **Godot says a file or scene could not be found after I moved it.** *Cause:* you moved or renamed files outside the editor. *Fix:* use the FileSystem dock; if it already happened, move the file back or choose a new path when Godot asks.

Further reading

- Nodes and Scenes: https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html
- Introduction to Godot, Nodes and Scenes: https://docs.godotengine.org/en/stable/getting_started/step_by_step/nodes_and_scenes.html
- Creating instances: https://docs.godotengine.org/en/stable/getting_started/step_by_step/instancing.html
- Scene organization: https://docs.godotengine.org/en/stable/tutorials/best_practices/scene_organization.html

Chapter 3: Your First Script: Variables, Types, and the Output Panel

Learning objectives

By the end of this chapter you will be able to:

- Explain what a program and a script are, and why a computer needs exact instructions.
- Attach a new GDScript file to a node and run it.
- Print messages to the Output panel and read them.
- Create variables and constants, give them good names, and change their values.
- Choose between the basic types `int`, `float`, `String`, and `bool`, and convert between them.
- Combine values with arithmetic, comparison, and string operators, and read simple error messages.

Key terms

- **Program**: a list of instructions that a computer follows, in order.
- **Script**: a text file of code attached to a node. In Godot, scripts are written in GDScript and end in `.gd`.
- **GDScript**: Godot's built-in programming language, designed for games.
- **Statement**: one instruction, usually one line of code.
- **Function**: a named group of statements that runs when called. `_ready()` is a function Godot calls for you.
- **Variable**: a named box in the computer's memory that holds a value that can change.
- **Constant**: a named value that is set once and never changes (written with `const`).
- **Type**: the kind of value something is, such as whole number, decimal number, text, or true/false.
- **Operator**: a symbol that does something with values, such as `+`, `-`, `*`, `/`, or `==`.
- **Assignment**: storing a value into a variable with a single equals sign (`=`).
- **Comment**: a note in the code for humans, beginning with `#`, which the computer ignores.
- **Output panel**: the editor panel where text from `print()` appears.

3.1 What is programming?

So far you have built games by placing nodes and changing settings. That is powerful but also fixed: the scene you built looks the same every time you run it. A real game must *do* things: respond to the player, count points, decide what happens next. To make things happen you write a **program**.

A program is a list of instructions. If you have ever followed a recipe, you have followed a program: "Crack two eggs into a bowl. Add one cup of flour. Stir for one minute." A computer follows instructions the same way, with two big differences.

1. **A computer does exactly what you say, not what you mean.** If a recipe forgot to say "remove the shells," a person would work it out. A computer would put the shells in the bowl. Programming is mostly the skill of saying things precisely.
2. **A computer is extremely fast and never tires.** It can follow millions of instructions in a second and repeat them forever without a mistake.

> **Why it works:** Programming feels hard at first because we are so used to being understood loosely. Slow down and be exact. The reward is that the computer will do precisely what you wrote, every time, so a mistake is always findable.

Scripts in Godot

In Godot you do not write a whole program from scratch. The engine already runs the game loop (Chapter 1). Instead, you write a **script** and attach it to a node. The script adds new behavior to that node: it lets you say "when this node starts, do this," or "every frame, do that."

Scripts are plain text files whose names end in `.gd`. They are written in **GDScript**, a language made for Godot. GDScript was designed to be readable: it uses plain words and indentation rather than lots of punctuation. If you have seen Python, you will notice it looks similar. If you have never programmed, that is fine. You will learn everything you need as it comes up.

What you will write in this chapter

By the end you will write a script that stores facts about Pip (name, health, coins, speed), does a little math, and prints a short report in two places: the Output panel and a label on the screen. Along the way you learn the tools that every future chapter relies on.

3.2 Attaching your first script

Let us make a script and see it run. Create a new project named `first-script`, or open the companion project for this chapter. Set the viewport size to 640 by 360 if you like (Chapter 1).

1. Create a new scene. In the Scene dock choose **Other Node** and pick a plain **Node**. Rename it `Main`.
2. Select `Main`. At the top of the Scene dock there is a button that looks like a scroll with a small plus sign: **Attach a new or existing script to the selected node**. Click it. (You can also right-click the node and choose **Attach Script**.)
3. A dialog appears. The important fields are:
    - **Language:** `GDScript`.
    - **Inherits:** `Node`. This says which node type the script extends. It is filled in for you.
    - **Template:** leave it as the default node template. Templates are starting code. The default for a node includes a short skeleton with comments.
    - **Path:** where the file will be saved. Name it `main.gd` and keep it at the top of the project (`res://main.gd`).
4. Click **Create**. The editor switches to the **Script** workspace and shows your new file.

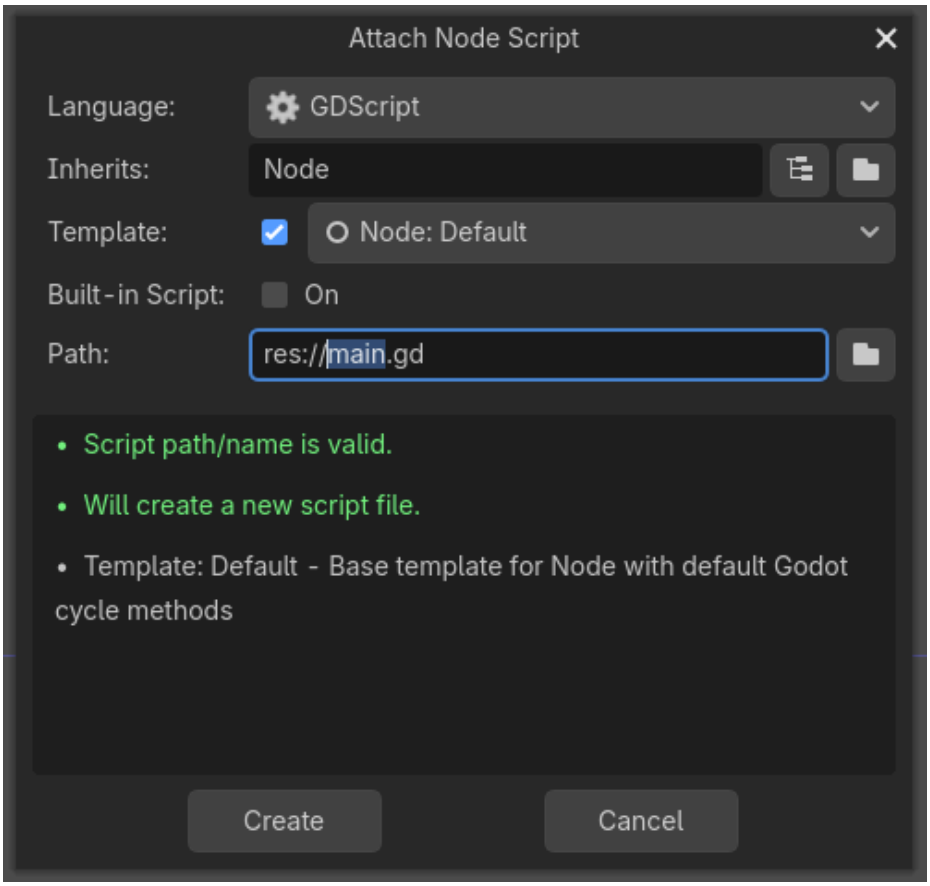


Figure 3.1: The Attach Node Script dialog. Pick GDScript, keep the defaults, and name the file `main.gd`.

You will see something like this (the exact comments differ between versions):

```gdscript
extends Node


# Called when the node enters the scene tree for the first time.
func _ready() -> void:
    pass


# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta: float) -> void:
    pass
```

Let us read it slowly, because every script you write follows this pattern.

- `extends Node` says: "this script adds to a `Node`." It must be the first real line. The script inherits all the node's abilities and adds its own.
- `func _ready() -> void:` begins a **function** named `_ready`. A function is a named group of instructions. Godot calls `_ready` once, when the node has entered the game and is ready. The `-> void` part means "this function gives nothing back." The colon `:` at the end opens the function's body.
- The lines **indented** underneath are the function's body. GDScript uses indentation (the space at the start of a line) to show what belongs inside what. Godot's editor uses a **tab** for each level. Mixing tabs and spaces causes errors, so let the editor do the indenting.
- `pass` means "do nothing." It is a placeholder, because a function body cannot be empty.
- `# Called when ...` is a **comment**: any text after `#` is ignored by the computer. It is for people.
- `func _process(delta: float) -> void:` is another function, which Godot calls every frame. We will use it in Chapter 5 and beyond. You can delete it for now, because we do not need it. Deleting it is safe.

Two names beginning with an underscore like `_ready` are **built-in functions**: Godot looks for them by name and calls them at the right moment. Spelling matters. `_Ready` or `ready` would be ignored.

Your first line of code

Delete `_process` and replace `pass` so your script reads:

```
extends Node


func _ready() -> void:
    print("Hello, world!")
```

Press **Ctrl+S** to save the script. Go back to the **2D** workspace (or any), save the scene as `main.tscn`, and press **F5**. If asked for a main scene, choose **Select Current**.

A window opens with a blank gray screen. That is expected, since the scene has no visible nodes. Now look at the bottom of the editor for the **Output** panel. It says:

```
Hello, world!
```

You have written and run your first program. The traditional first message, "Hello, world!", has been the first output of generations of programmers.



Figure 3.2: The Output panel. Each `print()` call adds a line of text here while the game runs.

> **Note:** If the Output panel is not visible, click the **Output** tab at the bottom of the editor, or the small arrow that opens the bottom panel. The game window can be closed at any time with its close button or the **Stop** button in the editor.

What just happened?

1. You pressed F5. Godot started the game and built the scene tree.
2. The `Main` node entered the tree and became ready. Godot called its `_ready` function.
3. The function ran its one statement: `print("Hello, world!")`.
4. `print` is a built-in function that writes text to the Output panel. The text goes between **quotation marks**, which mark it as a piece of text rather than code.

> **Try it:** Change the message to your own name and run again. Add a second `print` line below the first and run again. Notice that statements run **in order, from top to bottom**. Then try removing one of the closing quotation marks and look at the red error that Godot reports. Undo it by typing the quotation mark again.

3.3 Variables: named boxes

A program that only prints fixed messages is not very useful. Real programs *remember* things. A game must remember the player's health, the score, the name of the current level. Programs remember with **variables**.

A **variable** is a named box in the computer's memory. It has three things (Figure 3.3):

1. A **name**, so you can refer to it. (`health`)
2. A **value** inside it. (`100`)
3. A **type**, which says what kind of thing the box may hold. (`int`, a whole number)

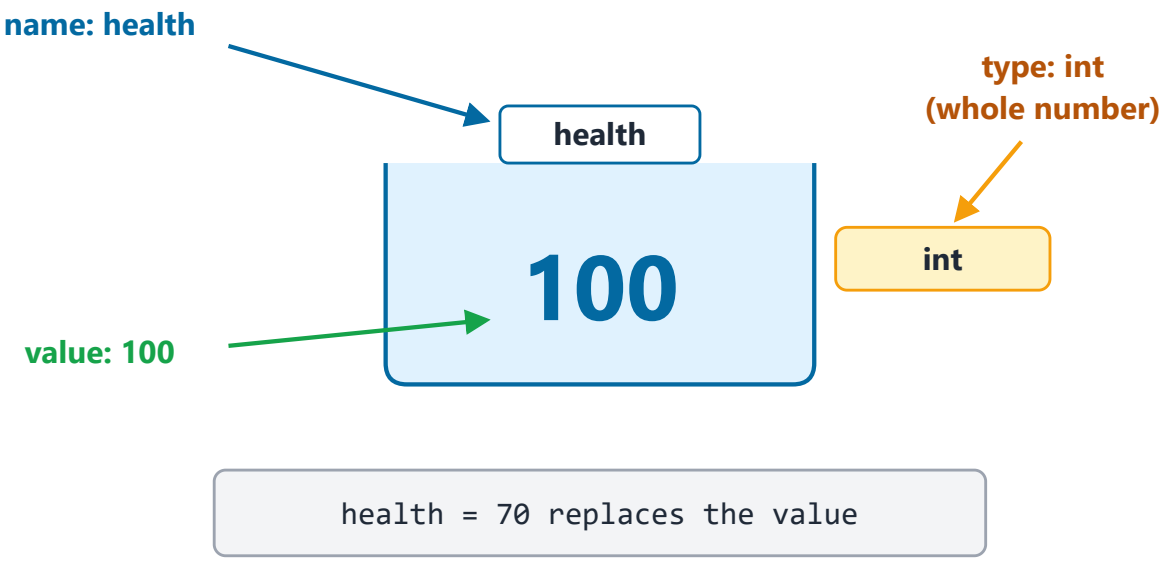


Figure 3.3: A variable is a named box that holds a value of a certain type. The value can change; the name stays.

Creating a variable

You create a variable with the keyword `var`:

```
extends Node


func _ready() -> void:
    var health: int = 100
    print(health)
    health = 70
    print(health)
```

This prints:

```
100
70
```

Break the line `var health: int = 100` into pieces:

Part	Meaning
`var`	"I am creating a variable."
`health`	Its name.
`: int`	Its type: a whole number.
`= 100`	Store the value 100 into the box.

The later line `health = 70` has no `var`, because the box already exists. It **assigns** a new value: the 100 is thrown out and replaced by 70. The single `=` sign does **not** mean "equals" as in mathematics. It means "put the value on the right into the box on the left." Read `health = 70` aloud as "health becomes 70."

> **Why it works:** Computers read a program from top to bottom. When `print(health)` runs the second time, the box holds 70, so that is what it prints. A variable always holds only the *latest* value stored in it.

Using a variable's value

Anywhere you can write a value, you can write the name of a variable that holds one. When the code runs, the name is replaced by what is currently in the box.

```
extends Node


func _ready() -> void:
    var health: int = 100
    var damage: int = 30
    health = health - damage
    print(health)
```

Look at `health = health - damage`. In mathematics this would be nonsense. In programming it is perfectly sensible and extremely common. The computer first works out the *right-hand side*: `100 - 30` is `70`. Only then does it store that into `health`. The output is `70`.

Naming variables

Good names make code readable. Follow these rules:

- Use **lowercase words joined by underscores**: `health`, `max_health`, `hero_name`, `coin_count`. This style is called **snake_case**, and it is the GDScript convention.
- Names can contain letters, digits, and underscores, but **cannot begin with a digit** and **cannot contain spaces**. `2fast` and `my health` are errors.
- Names are **case sensitive**: `Health` and `health` are two different variables.
- **Do not reuse words the language already owns**, like `var`, `func`, `if`, `extends`, `class`, or `signal`.
- Choose names that say what the value *means*: `coin_count`, not `x` or `thing2`.

Where a variable lives: local and member variables

All the variables above were created *inside* a function. These are **local variables**. They exist only while that function runs and are forgotten when it finishes. A variable created at the top of the script, outside any function, is a **member variable**. It belongs to the node, lasts as long as the node does, and every function in the script can use it.

```
extends Node

var coins: int = 0


func _ready() -> void:
    var bonus: int = 5
    coins += bonus
    print(coins)
```

Here `coins` is a member variable and `bonus` is a local one. `coins += bonus` is shorthand that we meet in section 3.5. The output is `5`. If another function in this script tried to use `bonus`, Godot would report an error: there is no `bonus` in that function. But it could use `coins`.

Use member variables for facts that belong to the thing the script describes (a hero's health), and local variables for temporary working values.

3.4 Types: what kind of value is it?

Computers keep different kinds of values in different ways, so every value has a **type**. You will use four basic types constantly.

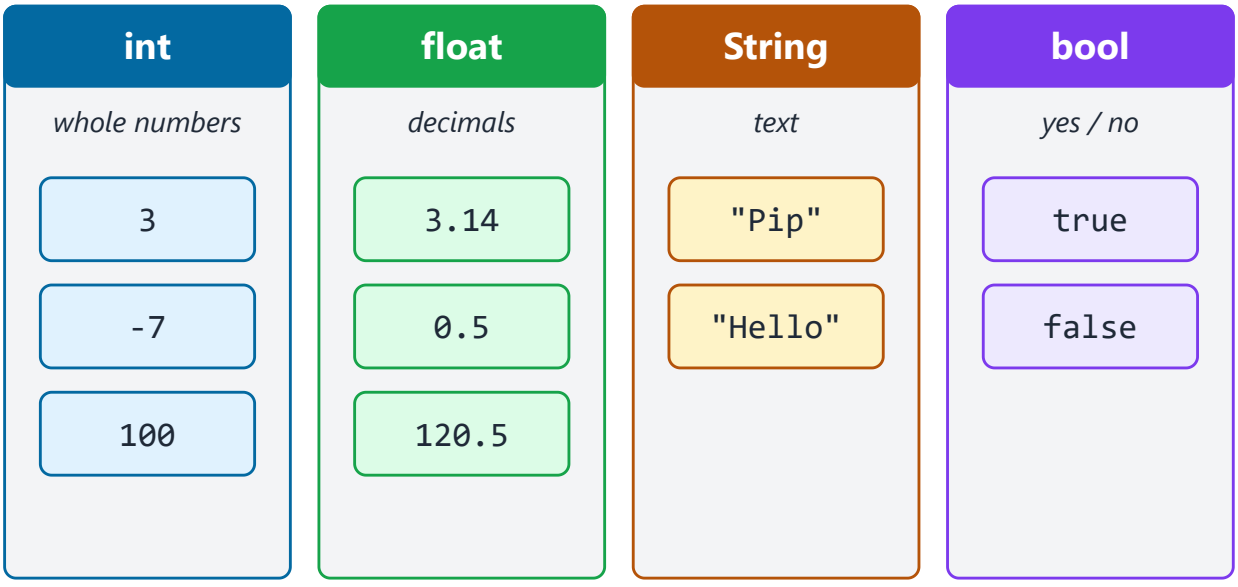


Figure 3.4: The four basic types and examples of each.

Type	Holds	Examples
`int`	a **whole number** (integer), positive, negative, or zero	`0`, `3`, `-7`, `100`
`float`	a number **with a decimal point** (floating-point number)	`3.14`, `0.5`, `-2.0`, `120.5`
`String`	**text**, written between quotation marks	`"Pip"`, `"Hello, world!"`, `""`

Type	Holds	Examples
`bool`	a **Boolean**: either `true` or `false` (no other value)	`true`, `false`

(A *bool* is named after the mathematician George Boole. It answers yes/no questions, like "is the hero alive?")

```
extends Node


func _ready() -> void:
    var lives: int = 3
    var speed: float = 120.5
    var title: String = "Pip"
    var is_alive: bool = true
    print(lives, " ", speed, " ", title, " ", is_alive)
```

This prints `3 120.5 Pip true`. Notice a few things:

- `print` can take several values separated by commas, and it joins them into one line with **nothing** added between them. That is why we added `" "` (a space in quotation marks) between them. Also note that `print("a", "b")` prints `ab`. If you want spaces added automatically, use `prints("a", "b", 3)`, which prints `a b 3`.
- The number `120.5` has a decimal point, so it is a `float`. The number `3` has none, so it is an `int`. Even `2.0` is a `float`, because of the `.0`.
- `true` and `false` are written in lowercase and without quotation marks. `"true"` in quotation marks would be text, not a Boolean.

Why types matter

Types protect you. If you try to store text into a box that was declared as a whole number, Godot notices and reports a problem *before* the game even runs. For example, this is wrong:

```
var lives: int = "three"   # error: a String cannot be stored in an int
```

(This one-line example is a fragment that is intentionally broken. Do not add it to your project, or Godot will refuse to run the script.) The red underline and the message tell you what you tried. Catching the mistake early is the whole point of writing the type down. This habit is called **static typing**, and this book uses it from the beginning.

Types also change what operators mean. `3 + 4` is `7`, but `"3" + "4"` is the text `"34"`. Same symbol, different type, different result.

Letting Godot work out the type

If you give a variable a value when you create it, Godot can figure out the type itself. Write `:=` instead of `: type =`:

```
extends Node


func _ready() -> void:
    var coins := 0
    var title := "Pip's Report"
    print(coins, title)
```

Here `coins` becomes an `int` because `0` is an integer, and `title` becomes a `String`. After that, they are just as strictly typed as before. The `:=` form is shorter, but writing the type out is clearer when you are learning, so the project in this chapter writes types explicitly.

> **Note:** `:=` needs a value whose type is certain. Writing `var y := x + 1` where `x` was created with plain `var x = 5` (no type) produces the error *"Cannot infer the type of 'y' variable because the value doesn't have a set type."* Fix it by typing both variables.

You may also write `var x = 5` with no type at all, which creates a flexible variable that can hold any type. That is allowed in GDScript, but it removes Godot's safety net, so we avoid it in this book.

Checking and converting types

Sometimes you have a value of one type and need another. Godot provides **conversion functions** named after the types:

```
extends Node


func _ready() -> void:
    print(int("42") + 1)
    print(int(3.9), float(2), str(true))
```

This prints:

```
43
32.0true
```

(The second line runs three values together: `int(3.9)` is `3`, `float(2)` is `2.0`, and `str(true)` is the text `true`.) Notice that `int(3.9)` **cuts off** the decimal part, giving `3`, rather than rounding. To round, use `round()`, `floor()`, or `ceil()`:

Function	Meaning	Example
`round(x)`	nearest whole number (halves round away from zero)	`round(2.5)` is `3.0`
`floor(x)`	round down	`floor(2.7)` is `2.0`
`ceil(x)`	round up	`ceil(2.1)` is `3.0`

These return a `float` with a whole value (`3.0`). Wrap them in `int(...)` if you need a whole number type.

You can also ask a value for its type with `typeof()`, and compare against names such as `TYPE_INT`, `TYPE_FLOAT`, `TYPE_STRING`, and `TYPE_BOOL`:

```
extends Node


func _ready() -> void:
    var lives: int = 3
    print(typeof(lives) == TYPE_INT)
```

This prints `true`. You will rarely need `typeof` in everyday work; it helps when you are curious or debugging.

Other types you will meet soon

Godot has many more types. Two will appear in the next chapters:

- `Vector2`: a pair of numbers `(x, y)`, such as a 2D position. `var spot := Vector2(3, 4)`, then `spot.x` is `3.0`.
- `Color`: a color, such as `Color.RED`.

Mentioning them now is only so that you are not startled later. You already used them without writing code: the Position property in the Inspector is a `Vector2`, and the color pickers set `Color` values.

3.5 Operators: doing things with values

An **operator** is a symbol that combines or compares values. You know most of them from math class.

Arithmetic

```
extends Node


func _ready() -> void:
    var a: int = 7
    var b: int = 2
    print(a + b)
    print(a - b)
    print(a * b)
    @warning_ignore("integer_division")
    print(a / b)
    print(a % b)
    print(float(a) / b)
    print(a / 2.0)
```

Output, line by line:

```
9
5
14
3
1
3.5
3.5
```

Operator	Name	Example	Result
`+`	add	`7 + 2`	`9`
`-`	subtract	`7 - 2`	`5`
`*`	multiply (the star, not an x)	`7 * 2`	`14`
`/`	divide	`7 / 2`	`3` (!)
`%`	remainder ("modulo")	`7 % 2`	`1`

Two surprises deserve attention.

Surprise 1: integer division. When *both* numbers are `int`, `/` also gives an `int`, and the remainder is thrown away. So `7 / 2` is `3`, not `3.5`. To get the decimal answer, make at least one side a `float`: `7 / 2.0` or `float(a) / b`. Godot warns you with a yellow **warning** ("integer division, decimal part will be discarded") because this is a

common slip. In the example above the line `@warning_ignore("integer_division")` tells Godot "I know, I did this on purpose" for the next line. You will not need that line unless you intentionally divide integers.

Surprise 2: the remainder operator. `%` gives what is left after dividing as many whole times as possible. `7 % 2` is `1` because 2 fits into 7 three times (6) with 1 left over. Remainders are surprisingly useful: `n % 2` is `0` for even numbers, and `1` for odd ones.

Order of operations

Multiplication and division happen before addition and subtraction, just as in math. Use parentheses to control the order:

```
extends Node


func _ready() -> void:
    print(2 + 3 * 4)
    print((2 + 3) * 4)
```

This prints `14` and then `20`.

Shortcuts for changing a variable

Changing a variable using its own current value is so common that there are shortcuts:

Long form	Shortcut
`score = score + 10`	`score += 10`
`score = score - 3`	`score -= 3`
`score = score * 2`	`score *= 2`
`score = score / 2`	`score /= 2`

```
extends Node


func _ready() -> void:
    var score: int = 0
    score += 10
    score -= 3
    score *= 2
    print(score)
```

The score goes 0, then 10, then 7, then 14, and the program prints `14`.

Comparison operators

Comparison operators ask a question and answer with a `bool`:

Operator	Question
`==`	Are these equal?
`!=`	Are these different?
`<`	Is the left smaller?
`>`	Is the left bigger?

Operator	Question
`<=`	Smaller or equal?
`>=`	Bigger or equal?

```
extends Node


func _ready() -> void:
    print(7 == 7, " ", 7 != 7, " ", 5 > 3, " ", 5 <= 3)
```

This prints `true false true false`.

> **Warning:** One equals sign (`=`) **stores**. Two equals signs (`==`) **ask**. Writing `=` where you meant `==` is one of the most common beginner errors. Godot usually flags it, but keep it in mind.

Comparisons are not very useful on their own yet. In Chapter 4, `if` statements use them to let the game make decisions.

Working with text

Operators work on text too. Joining two pieces of text is called **concatenation**, and it uses `+`:

```
extends Node

var hero_name: String = "Pip"


func _ready() -> void:
    print("Hello, " + hero_name + "!")
```

This prints `Hello, Pip!`.

You cannot add text to a number directly. `"Health: " + 70` is an error. First convert the number to text with `str()`:

```
extends Node


func _ready() -> void:
    var health: int = 70
    print("Health: " + str(health))
```

A tidier way to mix text and values is the **format operator**, `%`. Put a placeholder in the text and the values after a `%`:

```
extends Node


func _ready() -> void:
    var health: int = 70
    var player: String = "Pip"
    print("Health: %d" % health)
    print("%.2f" % 3.14159)
    print("%s has %d coins" % [player, 15])
```

This prints:

```
Health: 70
3.14
Pip has 15 coins
```

The placeholders are `%d` for a whole number, `%s` for text, `%f` for a decimal number, and `%.2f` for a decimal number rounded to 2 places. When there is more than one value, list them in square brackets, in the same order as the placeholders. Do not worry if it feels fussy; you can always fall back on `+` with `str()`.

Special characters inside text begin with a backslash. The two you need most are `\n` (start a new line) and `\t` (a tab):

```
extends Node


func _ready() -> void:
    print("Line one\nLine two\tTabbed")
```

This prints two lines, with the second line containing a tab after "Line two".

> **Try it:** Write a script that stores a `hero_name` and a `coins` count, then prints a sentence like `Pip has 15 coins.` using the format operator. Then print the same sentence using `+` and `str()`. Compare how they feel.

3.6 Constants, comments, and exports

Constants

A **constant** is a named value that never changes while the game runs. Declare it with `const`, conventionally in `UPPER_SNAKE_CASE`:

```
extends Node

const MAX_HEALTH: int = 100


func _ready() -> void:
    print("The most health a hero can have is ", MAX_HEALTH)
```

Why not just write `100` wherever you need it? Two reasons. First, a name explains the number: `MAX_HEALTH` is clearer than a bare `100`. Second, if you later decide the maximum should be 120, you change one line instead of hunting for every `100` (some of which might be a different 100). Programmers call a bare unexplained number in code a **magic number**, and constants are the cure. If you try to assign a new value to a constant, Godot reports an error.

Comments

A comment starts with `#` and runs to the end of the line:

```
extends Node


func _ready() -> void:
    # Pip starts the level with full health.
```

```
    var health: int = 100  # a comment can also follow code
    print(health)
```

Use comments to explain *why* you did something, not to repeat what the code obviously says. `# set health to 100` above `health = 100` adds nothing. You can also temporarily **comment out** a line, turning it into a comment so it does not run. In the Godot script editor, select lines and press **Ctrl+K** to toggle comments on or off. A comment starting with two hash marks (`##`) just above a function or variable is a *documentation comment*, which the editor shows in tooltips; it is optional.

Exported variables: change them in the Inspector

Here is a little preview of how scripts and the Inspector work together. Add `@export` in front of a member variable:

```
extends Node

@export var hero_name: String = "Pip"


func _ready() -> void:
    print("Hello, ", hero_name)
```

Save the script, select the `Main` node in the Scene dock, and look at the Inspector. A new section shows a property named **Hero Name** with the value `Pip`. Type a different name in the Inspector, run the game, and the Output panel prints the new name. The variable can now be adjusted without touching the code. This is how designers tweak speeds, health, and colors while the game is open. Exports come back often in Parts II and III.

> **Note:** The label in the Inspector is the variable name with underscores turned into spaces and capital letters added: `hero_name` shows as "Hero Name". The value you set in the Inspector is saved in the scene file and wins over the default written in the script.

3.7 Showing results on the screen

`print` writes to the Output panel, which only you, the developer, can see. Players need to see things in the game window. For that we use a `Label` node, which you met in Chapter 1.

Add a `Label` as a child of `Main` and name it `Report`. In the script, you can reach it using a **node path** with the dollar sign. We study this properly in Chapter 5; for now, copy the pattern:

```
extends Node

@onready var report: Label = $Report


func _ready() -> void:
    report.text = "Hello from the screen!"
```

Two new things appear:

- `$Report` means "the child node named `Report`." The `$` is a shortcut for finding a node by its name or path (the paths from Chapter 2).

- `@onready` says "fill in this variable at the moment the node is ready, not before." That is needed because the child nodes do not exist yet while the script is first loaded. Without `@onready`, `$Report` would not find anything.

Then `report.text = ...` sets the Label's `text` property, exactly what you did in the Inspector in Chapter 1, now from code. The dot (`.`) means "the thing that belongs to": `report.text` is "the text property of report."

So a script can do everything you do in the Inspector, and do it while the game runs. This is the key idea behind the next ten chapters.

3.8 Reading error messages

You will make mistakes. Every programmer does, every day. The skill is not avoiding errors; it is reading them calmly. Godot reports problems in three places.

1. **In the script editor, as you type.** A red underline or a red circle in the margin marks an error. A yellow triangle marks a warning. Hover the mouse over it to read the message.
2. **When you press F5.** If a script has a **parse error** (code Godot cannot understand), the game refuses to start and the **Debugger** or **Output** panel lists the problem with a file name and a line number.
3. **While the game runs.** A **runtime error** happens when the code is valid but something goes wrong during execution. The Debugger panel shows it, with the line.

Here are the errors you are most likely to meet in this chapter.

Message (approximately)	Likely cause	Fix
Parse Error: Expected ... ")" or *Unterminated string*	A missing closing parenthesis or quotation mark	Check the quotation marks and parentheses on that line
Identifier "heatlh" not declared in the current scope	A typo in a variable name, or using a local variable outside its function	Fix the spelling; declare the variable where you can use it
Cannot assign a value of type String as int style message	You stored the wrong type	Convert the value or choose the right type
Invalid operands 'String' and 'int' in operator '+'	You added text and a number	Use `str(number)`
Mixed use of tabs and spaces for indentation	Inconsistent indentation	Use the editor's Tab key throughout the file
Yellow warning: *integer division*	Dividing two `int` values	Use a `float` on one side if you want decimals

Always read **the line number first** and then the **message**. Very often the real mistake is on the line just before the one Godot underlines (for example a missing quotation mark). If the message does not make sense, copy it exactly into a search engine along with the word "Godot 4," and you will almost certainly find an explanation.

> **Try it:** Deliberately break your script in each of these ways, one at a time, read what Godot says, and fix it: (1) delete a closing quotation mark; (2) misspell a variable name in a `print`; (3) write `"Score: " + 5`. Seeing each error once, on purpose and calmly, makes it far less scary when it happens by accident.

Project 3: Pip's Report

Goal: write a script that keeps facts about Pip in variables, does a little arithmetic, and shows the report both in the Output panel and on the screen. Time: about 45 minutes.

Final file list (companion folder `projects/ch03-first-script/`):

```
ch03-first-script/
├── project.godot
├── icon.svg
├── main.tscn
└── main.gd
```

Scene tree (save as `main.tscn`):

```
Main (Node)         <- main.gd is attached here
└── Report (Label)
```

Steps

1. Create a new project named `pips-report`. In Project Settings, set the viewport to 640 by 360.
2. Create a scene with a plain **Node** as root, named `Main`.
3. Add a **Label** child named `Report`. In the Inspector, give it **Layout > Transform > Position** `24, 20` and **Size** `592, 320`, so it has room for several lines of text.
4. Select `Main` and attach a new script named `main.gd` (Inherits `Node`).
5. Replace the contents of `main.gd` with the code below.
6. Save the script (Ctrl+S) and the scene as `main.tscn`.
7. Press **F5** and choose **Select Current** if asked.
8. Compare the Output panel and the window to the Expected result.

`main.gd` (attach to `Main`, a `Node` whose only child is the `Label` named `Report`):

```gdscript
extends Node

const MAX_HEALTH: int = 100

var hero_name: String = "Pip"
var health: int = MAX_HEALTH
var speed: float = 120.5
var is_alive: bool = true
var coins: int = 0

@onready var report: Label = $Report


func _ready() -> void:
    print("Hello from ", hero_name, "!")
    print("Starting health: ", health, " / ", MAX_HEALTH)

    # Pip collects three coins worth 5 each, then hits a spike.
    var coin_value: int = 5
    coins = coins + coin_value
    coins += coin_value
    coins += coin_value
    health -= 30
    is_alive = health > 0

    var speed_text: String = "%.1f" % speed
    var summary: String = ""
    summary += "Hero: " + hero_name + "\n"
```

```
    summary += "Health: " + str(health) + " / " + str(MAX_HEALTH) + "\n"
    summary += "Coins: " + str(coins) + "\n"
    summary += "Speed: " + speed_text + " pixels per second\n"
    summary += "Alive? " + str(is_alive) + "\n"
    summary += "Health as a fraction: " + str(float(health) / MAX_HEALTH) + "\n"
    summary += "Coins split between 2 friends: " + str(coins / 2) + " each, " + str(coins % 2) + " left
over"

    print(summary)
    report.text = summary
    print("Type of health: ", typeof(health) == TYPE_INT)
```

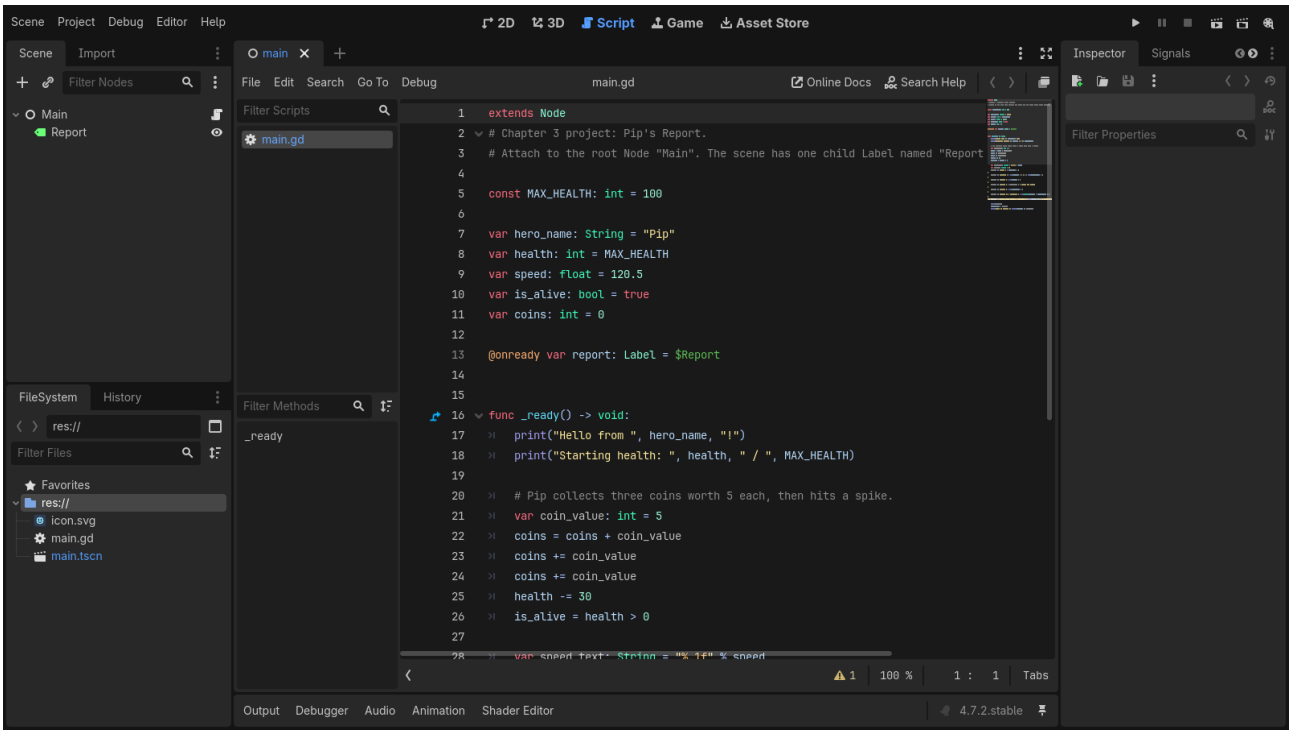


Figure 3.5: `main.gd` in the Script workspace.

How it works, step by step

- The constant `MAX_HEALTH` is the top health; `health` starts at that value.
- In `_ready`, the coins are added in two ways: `coins = coins + coin_value` and the shortcut `+=`. Both do the same thing.
- `health -= 30` removes the spike damage. `is_alive = health > 0` stores the *answer to a question* (a `bool`) in a variable.
- `"%.1f" % speed` formats the decimal with one digit.
- `summary` starts empty (`""`) and grows with `+=`, because `+=` also works on text. `\n` starts a new line each time.
- `float(health) / MAX_HEALTH` converts `health` first so the division keeps decimals: `70 / 100` would be `0`.
- The last line of the summary divides `coins` by 2 using integer division and uses `%` to find the remainder. Godot may show a yellow integer-division warning on that line in the editor; it is harmless here, and you can silence it by writing `@warning_ignore("integer_division")` on the line above.
- Finally the summary goes to both the Output panel and the label.

Expected result: the Output panel shows

```
Hello from Pip!
Starting health: 100 / 100
Hero: Pip
Health: 70 / 100
```

```
Coins: 15
Speed: 120.5 pixels per second
Alive? true
Health as a fraction: 0.7
Coins split between 2 friends: 7 each, 1 left over
Type of health: true
```

and the game window shows the seven-line report starting from "Hero: Pip" in the top-left. There are no red errors in the Debugger.

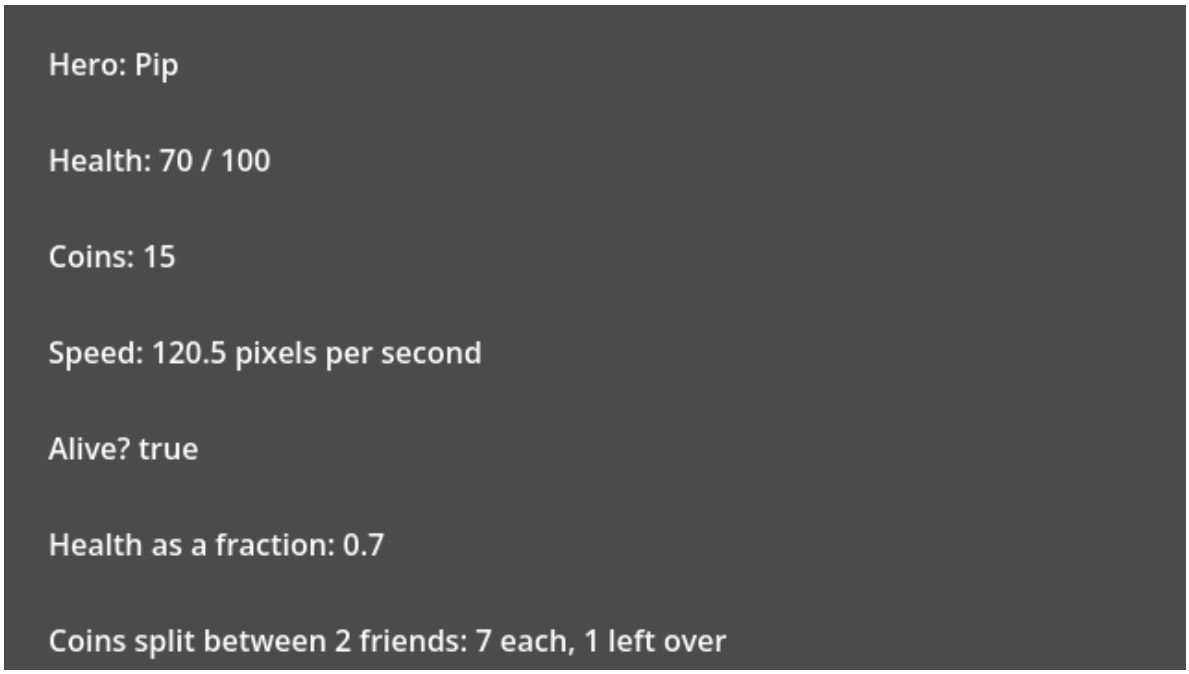


Figure 3.6: The finished project running.

Extensions

1. Add a `level` variable (an `int`) and show it in the report.
2. Make `hero_name` an `@export` variable and change it from the Inspector.
3. Add a variable `jump_height` as a `float` and print how many times higher it is than `speed / 10.0`.

Chapter summary

- A **program** is a list of exact instructions. In Godot, you write **scripts** (`.gd` files) and attach them to nodes.
- `extends Node` says which node type a script builds on. `func _ready() -> void:` runs once when the node is ready.
- `print()` shows text in the **Output** panel. Statements run top to bottom.
- A **variable** is a named box with a name, a value, and a type. `=` stores a value; it does not mean "equals."
- Basic types: `int` (whole numbers), `float` (decimals), `String` (text), `bool` (`true` or `false`). Types protect you from mixing things up.
- Local variables live inside a function; member variables belong to the whole script. `const` holds a value that never changes.
- Operators: `+ - * / %` for arithmetic, `== != < > <= >=` for comparisons, `+` to join text, and `str()`, `int()`, and `float()` to convert. Dividing two `int` values discards the decimal part.
- `#` starts a comment. `@export` shows a variable in the Inspector. `$Name` with `@onready` finds a child node so a script can change its properties.

- Read errors calmly: line number first, then the message.

Review questions

1. In your own words, what is the difference between a program and a script in Godot?
2. What does the function `_ready()` do, and who calls it?
3. What does the single equals sign `=` do in `health = 70`? How is it different from `==`?
4. Which type would you choose for each: (a) the number of lives, (b) the hero's speed in pixels per second (like 120.5), (c) the player's name, (d) whether a door is open?
5. What does this script print?

```gdscript
extends Node


func _ready() -> void:
    var n: int = 10
    n -= 4
    n *= 3
    print(n)
```

6. What do `7 / 2`, `7 / 2.0`, and `7 % 2` evaluate to?
7. True or false: `var 2fast: int = 5` is a valid variable declaration.
8. Which line has a problem, and why? (a) `print("Score: " + str(5))` (b) `print("Score: " + 5)`
9. Why is `const MAX_HEALTH: int = 100` better than writing `100` in several places?
10. What is the difference between a local variable and a member variable? Give one example of when each is a good choice.

Exercises

1. **Practice (easy):** Write a script that prints your name, your age as an `int`, and your height in meters as a `float` on three separate lines.
2. **Practice (easy):** Make a script that sets `var score: int = 0`, adds 50 three times using `+=`, subtracts 20, and prints the final score. Predict the number before you run it.
3. **Apply (medium):** Pip has 17 coins and wants to share them equally among 5 friends. Use `/` and `%` to print how many each friend gets and how many are left. Then print the exact decimal answer (3.4) using a `float`.
4. **Apply (medium):** Create a `Label` and an `@export var greeting: String` and show it in the label with `report.text = greeting`. Change the greeting from the Inspector without editing the code.
5. **Challenge (hard):** Write a "shop receipt" script. Store an item name, a price (`float`), and a quantity (`int`). Calculate the total and print a receipt of at least three lines using the format operator `%`, with the total displayed to exactly two decimal places. Then add a constant `TAX_RATE` of `0.08` and show the total with tax.

Common mistakes

- **The Output panel is empty.** *Cause:* the script is not attached to a node in the scene you ran, the function is misspelled (`_Ready`), or the Output tab is hidden. *Fix:* check the scroll icon beside the node in the Scene dock, check the spelling `_ready`, and open the Output tab.

- **"Mixed use of tabs and spaces."** *Cause:* indentation made with both tabs and spaces. *Fix:* select all the code and let the editor convert it (Edit > Convert Indent to Tabs in the script editor), then use only the Tab key.
- **"Identifier not declared in the current scope."** *Cause:* a misspelled variable, or a local variable used outside its function. *Fix:* check the spelling; move the variable to the top of the script if several functions need it.
- `"Score: " + 5` **gives an error.** *Cause:* text and numbers cannot be joined directly. *Fix:* `"Score: " + str(5)` or `"Score: %d" % 5`.
- `7 / 2` **gives** `3`**, not** `3.5`**.** *Cause:* integer division. *Fix:* make one side a float, such as `7 / 2.0` or `float(a) / b`.
- `$Report` **says "Node not found" or is null.** *Cause:* the child's name differs from the path, or the variable was not marked `@onready`. *Fix:* match the name exactly, including capitals, and use `@onready var report: Label = $Report`.

Further reading

- GDScript reference: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
- GDScript static typing: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/static_typing.html
- GDScript style guide: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_styleguide.html
- Creating your first script: https://docs.godotengine.org/en/stable/getting_started/step_by_step/scripting_first_script.html

Chapter 4: Making Decisions and Repeating

Learning objectives

By the end of this chapter you will be able to:

- Write comparisons and combine them with `and`, `or` and `not` to produce a true or false answer.
- Use `if`, `elif` and `else` to make a script choose between different actions.
- Use `match` to pick one action out of many possible values.
- Repeat work with `while` loops and `for` loops, and stop or skip a repeat with `break` and `continue`.
- Write your own functions with parameters and return values, and call them from other code.
- Store lists of values in arrays and name-to-value pairs in dictionaries, and loop over both.

Key terms

- **Boolean (`bool`):** a value that is either `true` or `false`. Nothing else.
- **Comparison operator:** a symbol such as `==`, `<` or `>=` that compares two values and gives back a boolean.
- **Condition:** any expression that produces a boolean, used to decide what the script does next.
- **Branch:** one of the possible paths a script can take after a decision.
- **Loop:** code that runs again and again until a condition says stop.
- **Iteration:** one single trip through the body of a loop.
- **Function:** a named, reusable block of code. You give it inputs (parameters) and it can hand back a result (a return value).
- **Parameter:** a named input of a function, written between the parentheses in its definition.
- **Return value:** the result a function hands back to the code that called it.
- **Array:** an ordered list of values, numbered from zero.
- **Dictionary:** a collection of key-value pairs, where you look things up by name (the key) instead of by position.
- **Scope:** the part of the script where a variable exists and can be used.

4.1 Programs that choose

In Chapter 3 you made variables and printed them. Every script you wrote ran straight from the first line to the last, like reading a shopping list. Real games do not work that way. A game constantly asks questions:

- Is the player touching a coin?
- Is the health at zero?
- Has the timer run out?
- Are there still enemies left on the screen?

And depending on the answer, it does different things. This chapter gives you the three tools for that: **decisions** (`if` and `match`), **repetition** (loops), and **reusable steps** (functions). It also gives you two containers that hold many values at once: **arrays** and **dictionaries**.

A small note before we start. The first few examples are short "fragments" you can test quickly. The simplest way to try any of them is to put the lines inside the `_ready()` function of a script attached to a node, exactly as you did in Chapter 3:

```
extends Node
# Attached to a Node. Anything inside _ready() runs once when the game starts.

func _ready() -> void:
    print("Hello from inside _ready!")
```

Whenever this chapter says "try this fragment", paste its lines inside `_ready()` (keep the tab indentation) and press **F5** (Run Project). The Output panel will show the result. Chapter 5 explains `_ready()` properly; for now, read it as "run these lines when the game starts".

> **Note:** In GDScript, indentation is not decoration. It is how the language knows which lines belong together. Use one **tab** for each level, as the editor does by default when you press the Tab key. We will see why this matters in a moment.

Booleans: the answer to a yes/no question

A **boolean** is the simplest kind of value. It can only be `true` or `false`. You can store one in a variable like any other value:

```
var has_key: bool = true
var door_is_open: bool = false
```

Booleans on their own are not very exciting. They become useful when they come out of a **comparison**.

Comparison operators

A comparison looks at two values and answers a yes/no question. The answer is always a boolean.

Operator	Meaning	Example	Result
`==`	is equal to	`5 == 5`	`true`
`!=`	is not equal to	`5 != 5`	`false`
`<`	is less than	`3 < 10`	`true`
`>`	is greater than	`3 > 10`	`false`
`<=`	is less than or equal to	`4 <= 4`	`true`
`>=`	is greater than or equal to	`3 >= 4`	`false`

You can test these right away. Try this fragment:

```
var health: int = 40
print(health > 0)        # true
print(health == 100)     # false
print(health != 100)     # true
print("Pip" == "Pip")    # true: text can be compared too
print("Pip" == "pip")    # false: capital letters matter
```

> **Warning:** The single equals sign `=` and the double equals sign `==` are completely different. `=` means "store this value in that variable" (assignment). `==` means "are these two values equal?" (comparison). Writing `if health = 0:` is an error. The double sign is the question; the single sign is the command.

Combining conditions: `and`, `or`, `not`

Often one question is not enough. "Can Pip open the door?" might mean "does Pip have the key **and** is the door unlocked?". GDScript gives you three words to combine booleans:

Word	Meaning	`true` when
`and`	both must be true	both sides are `true`
`or`	at least one must be true	one side or both are `true`
`not`	flips the answer	the value was `false`

```gdscript
var has_key: bool = true
var is_locked: bool = true
var has_axe: bool = false

print(has_key and is_locked)    # true: both are true
print(has_key and has_axe)      # false: has_axe is false
print(has_key or has_axe)       # true: at least one is true
print(not has_axe)              # true: flips false into true
print(has_key and not has_axe)  # true: key, and no axe
```

GDScript also accepts the symbols `&&`, `||` and `!` for the same three operations, because many other languages use them. The words are easier to read, so this book uses the words.

A truth table is a quick way to remember how `and` and `or` behave. Think of two switches, A and B:

A	B	`A and B`	`A or B`
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

A useful picture: `and` is two switches wired **in a row** (current only flows if both are on). `or` is two switches wired **side by side** (current flows if either is on).

You cannot chain comparisons the way you do in mathematics (`0 < x < 10`). Write each comparison separately and join them:

```gdscript
var x: int = 7
print(x > 0 and x < 10)   # true: x is between 0 and 10
```

A warning about decimal numbers

Decimal numbers (floats) are stored in a way that can produce tiny rounding errors. So this surprising result is normal:

```gdscript
print(0.1 + 0.2 == 0.3)   # false! The left side is 0.30000000000000004
```

When you need to compare floats, use the helper `is_equal_approx`, which says "equal, give or take a tiny error":

```
print(is_equal_approx(0.1 + 0.2, 0.3))   # true
```

You will use this later when comparing positions and speeds. For whole numbers (`int`) the plain `==` is always safe.

4.2 `if`, `elif` and `else`

The `if` statement runs a block of code only when a condition is `true`. It has this shape:

```
if condition:
    # this block runs only when the condition is true
```

Look at the pieces:

1. The word `if`.
2. A condition (anything that gives `true` or `false`).
3. A **colon** `:` at the end of the line. Forgetting the colon is the most common beginner error.
4. The lines to run, **indented** one more level.

```
var health: int = 0

if health <= 0:
    print("Pip has fainted.")
print("This line always runs, because it is not indented.")
```

The last `print` is lined up with `if`, so it does not belong to the `if` block. It runs no matter what. That is what indentation is doing: it draws the boundary of the block.

Adding an alternative with `else`

Sometimes you want one thing when the condition is true and a different thing when it is false. Add `else`:

```
var coins: int = 3

if coins >= 5:
    print("You can buy the potion.")
else:
    print("Not enough coins.")
```

Exactly one of the two blocks runs. Never both, never neither.

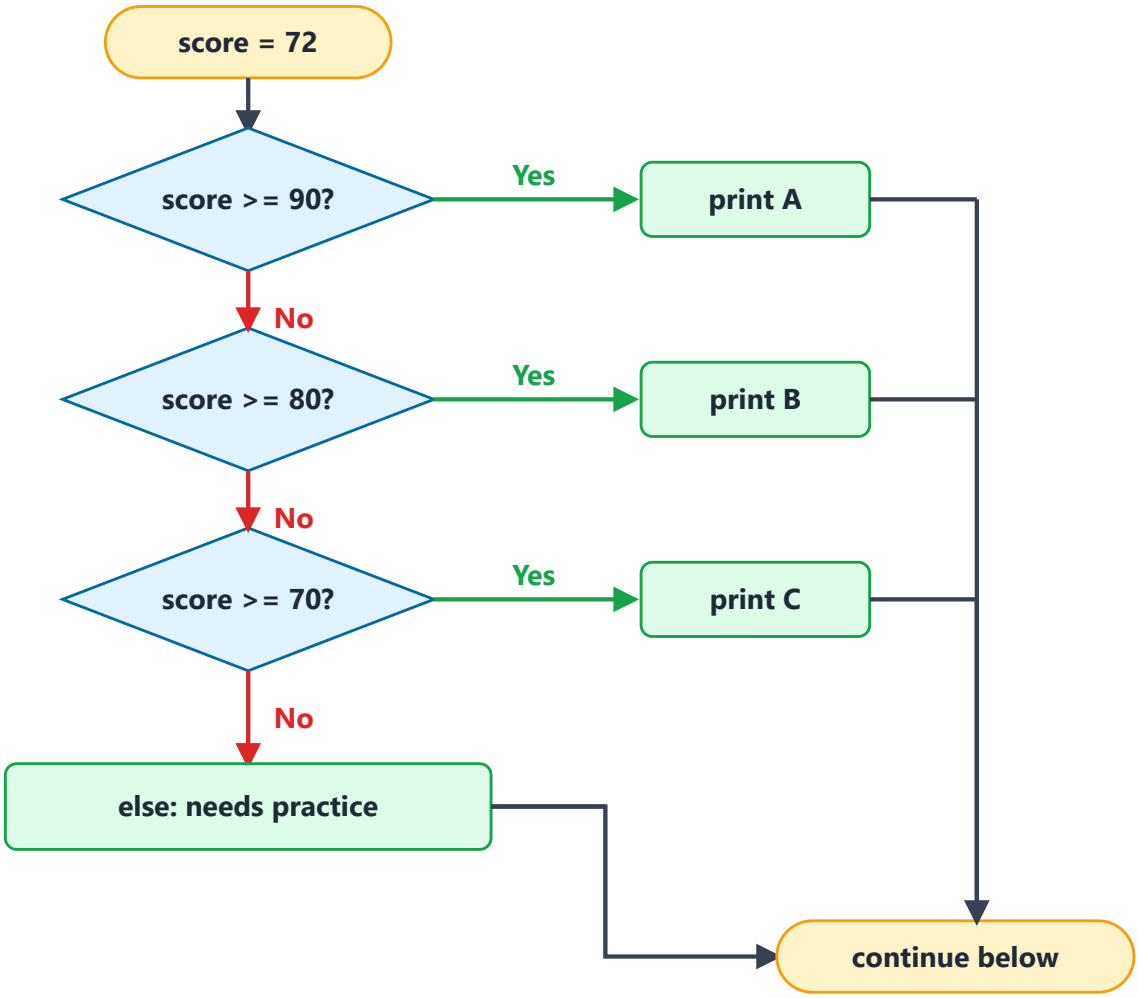


Figure 4.1: A decision is a fork in the road. The script checks each condition from the top and takes the first path whose condition is true. `else` is the road taken when no condition matched.

Many choices with `elif`

`elif` is short for "else if". It lets you check a second, third or fourth condition. GDScript tests them from the top and stops at the first one that is true.

```
var score: int = 72

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: needs practice")
```

With a score of 72, the first two conditions are false, the third is true, so it prints `Grade: C`. The `else` at the end is optional, but it is a good habit: it catches everything you did not think of.

> **Why it works:** The order matters. Suppose you wrote the `score >= 70` check *first*. A score of 95 is also at least 70, so it would print "C" and stop, and the A and B checks would never get a chance. Always put the most specific condition first.

Nested decisions

A decision can sit inside another decision. Indent one more level for each:

```
var has_key: bool = true
var is_night: bool = false

if has_key:
    if is_night:
        print("You unlock the door quietly.")
    else:
        print("You unlock the door.")
else:
    print("The door is locked.")
```

Nesting is allowed, but do not go too deep. Two levels is comfortable. Four levels is a sign that you should combine conditions with `and` or move part of the logic into a function (section 4.5).

A shortcut for simple choices

When you only need to pick one of two *values*, there is a one-line form:

```
var coins: int = 7
var label: String = "rich" if coins >= 5 else "poor"
print(label)   # rich
```

Read it aloud: "label is 'rich' if coins is at least 5, else 'poor'". Use it only when it stays short and clear. For anything longer, use a normal `if`.

Empty blocks: `pass`

Sometimes you want to write the shape of a decision now and fill it in later. A block cannot be empty, so use the word `pass`, which means "do nothing here":

```
var enemy_nearby: bool = false

if enemy_nearby:
    pass   # TODO: make Pip run away
```

> **Try it:** Create a new script with a variable `temperature: int = 25`. Print "Freezing" when it is below 0, "Cold" when it is below 15, "Warm" when it is below 30, and "Hot" otherwise. Change the number a few times and check each branch. This is the same shape as the grade example.

4.3 `match`: choosing by value

When you are comparing *one* value against many fixed possibilities, a long chain of `elif` gets tiresome:

```
var day: int = 3

if day == 1:
    print("Monday")
elif day == 2:
    print("Tuesday")
elif day == 3:
    print("Wednesday")
```

`match` says the same thing more neatly:

```
var day: int = 3

match day:
    1:
        print("Monday")
    2:
        print("Tuesday")
    3:
        print("Wednesday")
    _:
        print("Some other day")
```

How to read it:

- `match day:` means "look at the value in `day`".
- Each line like `1:` is a **pattern**. If `day` equals it, the indented block under it runs.
- `_:` is the **wildcard**. It matches anything, so it works like `else`. Put it last.
- Only the first matching pattern runs. You do not need `break`.

You can list several values for one pattern by separating them with commas, and `match` works on text as well as numbers:

```
var command: String = "run"

match command:
    "walk", "run":
        print("Pip moves.")
    "jump":
        print("Pip jumps.")
    _:
        print("Pip does not know that command.")
```

Use `if` when the condition is a *comparison* (`>=`, `<`, `and`...). Use `match` when you are asking "which one of these exact values is it?". Later chapters use `match` for game states such as "menu", "playing" and "game over".

4.4 Loops: doing something again

Imagine you want to print the numbers 1 to 5. You could write five `print` lines. Now imagine 1 to 1,000. A **loop** repeats a block of code so you do not have to copy it. Each trip through the block is called an **iteration**.

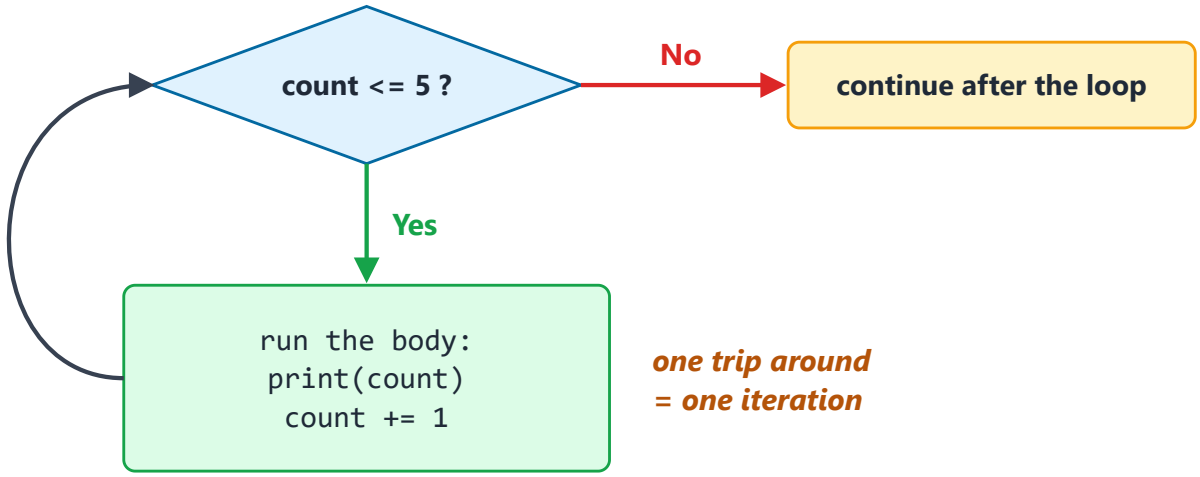


Figure 4.2: Every loop has the same rhythm. Check the condition, run the body once, return to the check. When the check fails, the loop ends and the script continues below it.

The `while` loop

A `while` loop repeats as long as its condition stays true:

```
var count: int = 1

while count <= 5:
    print(count)
    count += 1   # same as: count = count + 1
```

Walk through it slowly:

1. `count` starts at 1. Is `1 <= 5`? Yes, so run the body: print 1, then make `count` 2.
2. Is `2 <= 5`? Yes. Print 2. And so on.
3. After printing 5, `count` becomes 6. Is `6 <= 5`? No. The loop ends.

The shortcut `count += 1` means "add 1 to count". Similar shortcuts exist for the other operations: `-=`, `*=` and `/=`.

> **Warning:** If nothing inside a `while` loop ever makes the condition false, the loop never ends and your game freezes. Here, forgetting the line `count += 1` would repeat "print 1" forever. If a running game hangs, close its window (or press the Stop button in the editor) and look for a `while` whose condition cannot become false.

The `for` loop

Most of the time you know *how many times* you want to repeat, or you have a collection to walk through. A `for` loop is made for that:

```
for i in range(5):
    print(i)
```

This prints `0 1 2 3 4`. Two things surprise beginners:

- `range(5)` produces the numbers **0, 1, 2, 3, 4**: it starts at 0 and stops *before* 5. Computers usually count from zero.
- The name `i` is just a variable that holds the current number. You can choose any name: `for step in range(5):` works the same.

`range` can take more than one number:

```
for n in range(1, 6):        # start at 1, stop before 6
    print(n)                 # 1 2 3 4 5

for n in range(0, 10, 2):    # start at 0, stop before 10, go up by 2
    print(n)                 # 0 2 4 6 8

for n in range(3, 0, -1):    # count down: 3 2 1
    print(n)
```

A `for` loop can also walk straight through the items of an array (you will meet arrays in section 4.6):

```
var friends: Array[String] = ["Pip", "Slime", "Door"]

for friend in friends:
    print("Hello, " + friend + "!")
```

`friend` takes the value `"Pip"` on the first iteration, then `"Slime"`, then `"Door"`.

Stopping early: `break`, and skipping: `continue`

Two small words give you control inside a loop:

- `break` ends the whole loop immediately.
- `continue` skips the rest of the current iteration and jumps to the next one.

```
for n in range(1, 10):
    if n == 3:
        continue      # skip 3, go straight to 4
    if n == 6:
        break         # stop the whole loop at 6
    print(n)          # prints 1 2 4 5
```

A common use of `break` is "search until you find it":

```
var coins_per_room: Array[int] = [3, 7, 0, 12, 5]

for coins in coins_per_room:
    if coins == 0:
        print("Found an empty room!")
        break
```

Loops inside loops

A loop can contain another loop. The inner loop runs completely for every single iteration of the outer loop. This is how you would build a grid:

```
for row in range(3):
    for column in range(4):
```

```
        print("row ", row, " column ", column)
```

That prints 3 × 4 = 12 lines. You will use this pattern for tile grids in Chapter 12.

Choosing between `for` and `while`

- Use `for` when you know the count or you are walking through a collection. This is most of the time.
- Use `while` when you do not know how many repeats you need, only a stopping condition: "keep going until the health is zero".

> **Try it:** Write a `for` loop that prints the multiplication table for 7: "7 x 1 = 7", "7 x 2 = 14", up to 7 x 10. Hint: inside the loop you can print `"7 x ", i, " = ", 7 * i`, with `i` going from 1 to 10 using `range(1, 11)`.

4.5 Functions: naming a piece of work

So far every script lived in `_ready()`. A **function** is a named block of code you can run whenever you like, as many times as you like. You have already *used* functions: `print()` and `range()` are functions someone else wrote. Now you will write your own.

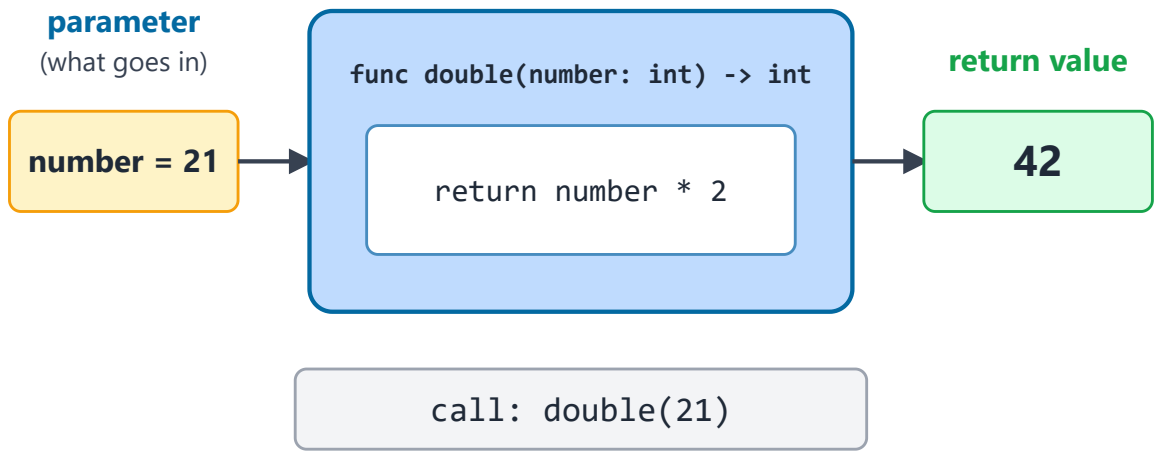


Figure 4.3: Think of a function as a small machine. Parameters are what you feed in, the body is the machine's work, and the return value is what comes out.

Defining and calling a function

```
func say_hello() -> void:
    print("Hello, adventurer!")
```

Piece by piece:

- `func` tells GDScript "a function definition starts here".
- `say_hello` is the name. Use lowercase words joined by underscores (this style is called snake_case).
- `()` is where the parameters go. This one has none.
- `-> void` says "this function does not hand anything back".
- The colon and the indented block form the body.

Defining a function does not run it. To run it, **call** it by writing its name with parentheses:

```
extends Node

func _ready() -> void:
    say_hello()
    say_hello()   # a second call runs the same code again

func say_hello() -> void:
    print("Hello, adventurer!")
```

Output:

```
Hello, adventurer!
Hello, adventurer!
```

Notice that `_ready()` is itself a function. Godot calls it for you when the game starts. The order in which you write functions in the file does not matter.

Parameters: giving the function input

A function that does the same thing every time is limited. Parameters let you pass different values in:

```
func greet(name: String) -> void:
    print("Hello, " + name + "!")
```

```
greet("Pip")      # Hello, Pip!
greet("Slime")    # Hello, Slime!
```

`name: String` means "this function expects one input, called `name`, which must be text". The value you pass in a call (`"Pip"`) is called an **argument**. The name inside the function body is the parameter. In practice people mix the two words, and that is fine.

A function can take several parameters, separated by commas:

```
func add_coins(current: int, found: int) -> void:
    print("Pip now has ", current + found, " coins.")
```

```
add_coins(10, 3)   # Pip now has 13 coins.
```

You can give a parameter a **default value**. If the caller leaves it out, the default is used:

```
func greet_with(name: String, greeting: String = "Hello") -> void:
    print(greeting + ", " + name + "!")
```

```
greet_with("Pip")             # Hello, Pip!
greet_with("Pip", "Welcome")  # Welcome, Pip!
```

Return values: getting a result back

Often you want the function to *compute* something and give it back. Use `return`, and write the result type after `->`:

```
func double(number: int) -> int:
    return number * 2
```

Now the call itself acts like a value. You can store it, print it, or use it in a bigger expression:

```
var result: int = double(21)
print(result)            # 42
print(double(5) + 1)     # 11
```

When Godot reaches `return`, the function stops right there and hands the value back. Any lines after `return` are skipped. That makes `return` useful inside an `if`:

```
func rank_for(total: int) -> String:
    if total >= 30:
        return "Treasure Hunter"
    elif total >= 15:
        return "Explorer"
    else:
        return "Beginner"
```

Every path through the function must return a value of the promised type. If you promise `-> String` and one branch forgets to return, GDScript reports an error.

A function that answers a yes/no question should return a `bool`. By convention its name often starts with `is_` or `can_` or `has_`:

```
func is_even(number: int) -> bool:
    return number % 2 == 0
```

The `%` symbol here is the **modulo** operator: it gives the remainder after division. `7 % 2` is `1` (7 divided by 2 is 3, remainder 1) and `8 % 2` is `0`. A number whose remainder after dividing by 2 is zero is even. Modulo shows up constantly in games: "every 5th hit", "every 3rd frame", "wrap around a list".

Scope: where a variable lives

A variable created inside a function exists only inside that function. This is called **scope**.

```
func make_pile() -> void:
    var pile: int = 5     # pile exists only inside make_pile
    print(pile)

func try_to_use_pile() -> void:
    print(pile)           # ERROR: "pile" does not exist here
```

The same rule applies to blocks: a variable created inside an `if` or a loop body disappears when that block ends. If you need a value to survive, declare it *before* the block, or return it.

Variables declared at the top of the script, outside any function, belong to the whole script. Every function in it can read and change them. These are called **member variables**, and they are what you used in Chapter 3:

```
extends Node

var total_coins: int = 0   # a member variable: every function can see it

func collect(amount: int) -> void:
    total_coins += amount
```

```
func show_total() -> void:
    print("Total: ", total_coins)
```

Why write functions?

Three reasons, in order of importance:

1. **Names explain intent.** `if can_open_door():` reads like English. The same logic written inline as three conditions joined by `and` does not.
2. **No copy and paste.** Fix a bug once, and it is fixed everywhere the function is called.
3. **Small pieces are easy to test.** You can call `rank_for(30)` alone and check the answer.

A good rule of thumb: if a function needs more than about 20 lines, or you need the word "and" to describe what it does, split it into two.

> **Note:** Writing `-> void`, `: int` and `-> String` is called **static typing**. Types are optional in GDScript, but they let the editor catch mistakes before you press Run, and they make code easier to read. We use them in most examples in this book.

4.6 Arrays: ordered lists

So far each variable has held one value. A game often needs a *list*: the coins in each room, the names of the enemies, the positions of the platforms. An **array** holds many values in order.

```
var names: Array[String] = ["Pip", "Slime", "Door"]
var coins_per_room: Array[int] = [3, 7, 0, 12, 5]
```

`Array[String]` means "an array that may only contain strings". This is a **typed array**. Typed arrays are safer because Godot refuses to put a number into a list of names.

You can also make an untyped array, `var stuff: Array = [1, "two", 3.0]`, which accepts anything. Use typed arrays unless you have a reason not to.

Indexes start at zero

Each item has a position number called an **index**. The first item is index **0**, not 1.

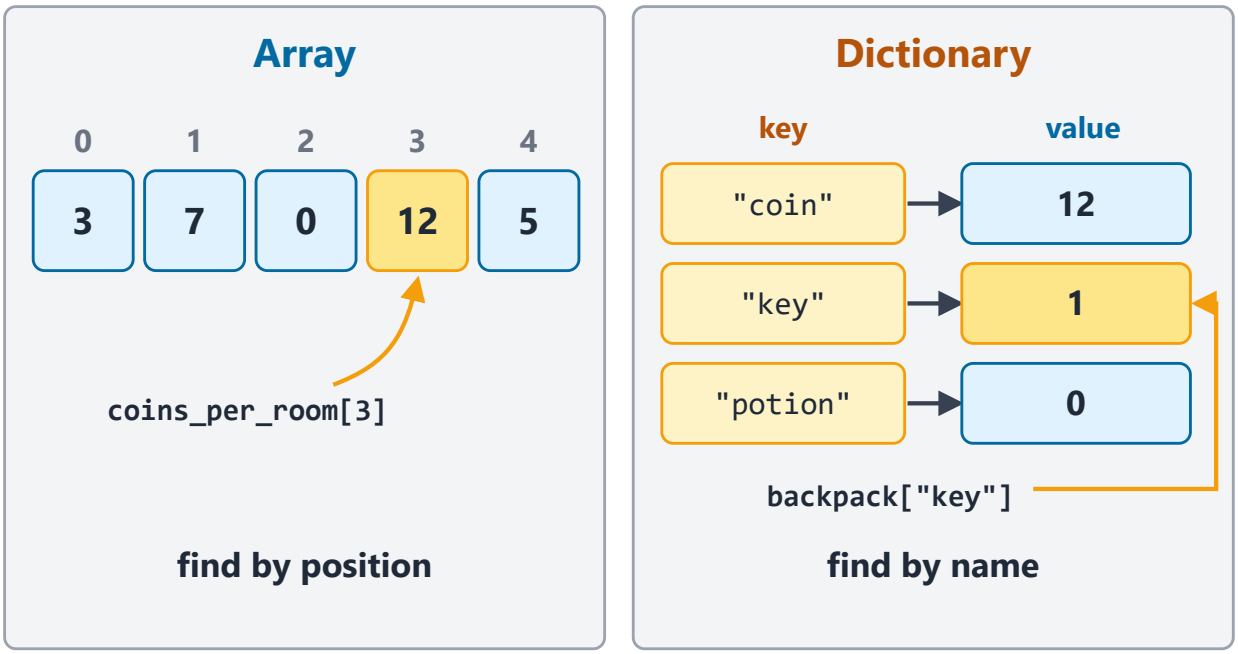


Figure 4.4: An array finds things by position (0, 1, 2...). A dictionary finds things by a name you choose, called a key.

```
var coins_per_room: Array[int] = [3, 7, 0, 12, 5]

print(coins_per_room[0])    # 3  (the first item)
print(coins_per_room[1])    # 7
print(coins_per_room[-1])   # 5  (a negative index counts from the end)
```

An array with 5 items has valid indexes 0 to 4. Asking for `coins_per_room[5]` is an error ("index out of bounds") and stops your game.

Change an item by assigning to its index:

```
coins_per_room[2] = 9      # room 3 is no longer empty
```

Useful array tools

Arrays come with built-in functions (called **methods** when they belong to a value). You call them with a dot:

```
var items: Array[String] = ["sword", "shield"]

items.append("potion")        # add to the end: [sword, shield, potion]
items.insert(0, "map")        # add at index 0: [map, sword, shield, potion]
print(items.size())           # 4: how many items
print(items.has("shield"))    # true: is it in the list?
print(items.find("sword"))    # 1: its index (-1 if not found)
items.erase("shield")         # remove the first match
items.remove_at(0)            # remove the item at index 0
print(items)                  # [sword, potion]
print(items.is_empty())       # false
items.clear()                 # remove everything
```

Other methods you will meet: `pop_back()` (remove and return the last item), `sort()` (put numbers or text in order), `reverse()`, `shuffle()` (random order) and `pick_random()` (return a random item).

You can also ask for `in`:

```
if "potion" in items:
    print("Pip has a potion.")
```

Looping over arrays

You saw this already. The simple form gives you each item:

```
var total: int = 0
for coins in coins_per_room:
    total += coins
print(total)   # 27
```

When you also need the position number, loop over the indexes with `range` and `size()`:

```
for i in range(coins_per_room.size()):
    print("Room ", i + 1, " has ", coins_per_room[i], " coins")
```

We print `i + 1` because people count rooms from 1 while the array counts from 0.

> **Warning:** Do not add to or remove from an array while you are looping over it. The loop gets confused about where it is. If you need to remove items, loop over a copy (`for x in items.duplicate():`) or collect the changes and apply them after the loop.

Putting an array in a function

Arrays are values like any other, so a function can take one and return one. Here is a function that adds up a list:

```
func sum_of(numbers: Array[int]) -> int:
    var total: int = 0
    for n in numbers:
        total += n
    return total
```

```
print(sum_of([1, 2, 3, 4]))   # 10
```

4.7 Dictionaries: looking things up by name

An array answers "what is item number 3?". A **dictionary** answers "how many coins does Pip have?". Each entry is a pair: a **key** (the name) and a **value**.

```
var backpack: Dictionary = {
    "coin": 12,
    "key": 1,
    "potion": 0,
}
```

The curly braces `{}` mark a dictionary. Each line reads `key: value`. A trailing comma after the last pair is allowed and makes it easy to add lines later.

Reading and changing values

Use square brackets with the key:

```
print(backpack["coin"])        # 12
backpack["coin"] = 15          # change an existing value
backpack["map"] = 1            # a new key creates a new entry
```

Reading a key that does not exist is an error. The safe way to read is `get`, which lets you provide a fallback:

```
print(backpack.get("sword", 0))    # 0: there is no sword, so use 0
```

Other everyday tools:

```
print(backpack.has("key"))     # true
print(backpack.size())         # how many pairs
backpack.erase("potion")       # remove a pair
print(backpack.keys())         # an array of all the keys
print(backpack.values())       # an array of all the values
```

When your keys are simple words, GDScript allows a shorter dot form: you can write `backpack.coin` instead of `backpack["coin"]`. The bracket form works with every key, so it is the one this book uses first.

Looping over a dictionary

A `for` loop over a dictionary gives you the **keys**. Use each key to look up its value:

```
for item in backpack:
    print(item, ": ", backpack[item])
```

This prints one line per pair. The order is the order in which the pairs were added.

Arrays or dictionaries?

Use an array when…	Use a dictionary when…
order matters	names matter
items are the "same kind of thing" (a list of coins)	each value means something different (health, name, speed)
you look up by position	you look up by name

You can also nest them. An array of dictionaries is a natural way to describe a list of enemies:

```
var enemies: Array[Dictionary] = [
    {"name": "Green slime", "health": 10},
    {"name": "Blue slime", "health": 14},
]

for enemy in enemies:
    print(enemy["name"], " has ", enemy["health"], " health")
```

In Chapter 16 you will learn a more powerful way to bundle data (custom resources), but dictionaries are perfect for quick, small jobs.

4.8 Reading errors and testing small pieces

You will see red error text in this chapter. That is normal; every programmer sees it daily. Three habits make it painless.

Read the first line of the error. The Debugger panel at the bottom of the editor shows a message such as `Invalid access to property or key 'sword' on a base object of type 'Dictionary'`. That tells you the problem (you asked a dictionary for a key it does not have), and the line number tells you where.

Test small. After writing one function, call it with a value where you already know the answer: `print(rank_for(30))` should print `Treasure Hunter`. Do this before you write the next function.

Print to see inside. If a loop misbehaves, put `print(i)` inside it and watch what happens. When it works, delete the print.

Here are the most frequent beginner errors from this chapter, and how they appear:

What you see	What it means
`Expected ":" after "if" condition`	You forgot the colon at the end of the line
`Unexpected indentation` or `Expected indented block`	The tabs do not line up, or the block is empty
`Index 5 out of bounds`	You used an array index past the end
`Invalid get index 'sword' (on base: 'Dictionary')`	You read a dictionary key that does not exist (use `get`)
Game freezes, no error	A `while` loop that never ends

Project 4: Pip's Treasure Report

Goal: build a small program that uses everything in this chapter. It loads a list of coin counts, a backpack dictionary, and a handful of functions, then builds a text report and shows it both on screen and in the Output panel.

Time: about 40 minutes.

Project folder: `projects/ch04-decisions-and-loops/` (a finished copy to compare with).

Files:

```
ch04-decisions-and-loops/
├─ project.godot
├─ icon.svg
├─ main.tscn
└─ main.gd
```

Steps

1. In the Godot Project Manager click **Create**, name the project `Ch04 Decisions and Loops`, pick a folder, and press **Create & Edit**. (The Compatibility or Forward+ renderer both work.)
2. Create the scene. In the Scene dock click **Other Node**, choose **Node**, and rename it **Main**. Add a child: right-click **Main**, **Add Child Node**, choose **Label**, and rename it **ReportLabel**. The tree must look like this (names are case-sensitive because the script refers to them):

   ```
   Main (Node)
   └─ ReportLabel (Label)
   ```

3. Select **ReportLabel**. In the Inspector, find **Layout** and set its position to about (24, 16) and its size to about (1100, 600) so the text has room. If the Inspector fields are greyed out, make sure ReportLabel is selected and not Main.

4. Save the scene as `main.tscn` (**Ctrl+S**).
5. Select **Main**, click the **Attach Script** button (the scroll icon with a green plus), keep the default settings, and name the file `main.gd`.

6. Replace the script contents step by step. First write the top of the file: the two data variables and the `_ready` function skeleton. Use tabs for indentation.

```
extends Node

@onready var report_label: Label = $ReportLabel

# Coins Pip found in each room, in order.
var coins_per_room: Array[int] = [3, 7, 0, 12, 5]

# What Pip carries: item name -> how many.
var backpack: Dictionary = {"coin": 12, "key": 1, "potion": 0}
```

`@onready var report_label ... = $ReportLabel` fetches the Label node when the scene is ready. Chapter 5 explains it fully; for now it simply gives you a variable named `report_label` that points to the label on screen.

7. Add the helper functions at the bottom of the file, one at a time. After each one, you can test it with a `print` in `_ready` if you like. First `sum_of` and `rank_for`:

```
func sum_of(numbers: Array[int]) -> int:
    var total: int = 0
    for n in numbers:
        total += n
    return total


func rank_for(total: int) -> String:
    if total >= 30:
        return "Treasure Hunter"
    elif total >= 15:
        return "Explorer"
    else:
        return "Beginner"
```

Then `describe_room`, `can_open_door` and `weather_message`:

```
func describe_room(coins: int) -> String:
    if coins == 0:
        return "empty room"
    elif coins < 5:
        return "%d coins (a few)" % coins
    elif coins < 10:
        return "%d coins (a nice pile)" % coins
    else:
        return "%d coins (jackpot!)" % coins


func can_open_door() -> bool:
    var has_key: bool = backpack.get("key", 0) > 0
    var has_potion: bool = backpack.get("potion", 0) > 0
    # The door opens with a key, or when we carry a potion to shrink through the gap.
    return has_key or has_potion
```

```
func weather_message(code: int) -> String:
    match code:
        0:
            return "sunny"
        1, 2:
            return "cloudy"
        3:
            return "rainy"
        _:
            return "unknown"
```

Here `"%d coins" % coins` is **string formatting**: the `%d` is a slot that is replaced by a whole number. When there are several slots, put the values in an array on the right: `"Room %d: %s" % [3, "empty"]`. `%s` is a slot for text.

8. Add the counting-game function `fizz_buzz`:

```
func fizz_buzz(limit: int) -> Array[String]:
    var results: Array[String] = []
    for n in range(1, limit + 1):
        if n % 15 == 0:
            results.append("FizzBuzz")
        elif n % 3 == 0:
            results.append("Fizz")
        elif n % 5 == 0:
            results.append("Buzz")
        else:
            results.append(str(n))
    return results
```

The 15 check comes first on purpose: a multiple of 15 is also a multiple of 3 and 5, so it must be tested before them. (This is the "most specific first" rule from section 4.2.)

9. Now write `_ready`, which collects lines of text in an array and then joins them. Put it right after the variables:

```
func _ready() -> void:
    var lines: Array[String] = []
    lines.append("PIP'S TREASURE REPORT")
    lines.append("")

    # 1. Functions and arrays: total and rank.
    var total: int = sum_of(coins_per_room)
    lines.append("Rooms visited: %d" % coins_per_room.size())
    lines.append("Total coins: %d" % total)
    lines.append("Rank: %s" % rank_for(total))
    lines.append("")

    # 2. A for loop with an index.
    lines.append("Room by room:")
    for i in range(coins_per_room.size()):
        var room_number: int = i + 1
        lines.append("  Room %d: %s" % [room_number, describe_room(coins_per_room[i])])
    lines.append("")

    # 3. Looping over a dictionary.
    lines.append("Backpack:")
    for item in backpack:
        var count: int = backpack[item]
        if count > 0:
            lines.append("  %s x%d" % [item, count])
        else:
            lines.append("  %s (empty)" % item)
    lines.append("Door can open: %s" % str(can_open_door()))
```

```
    lines.append("")

    # 4. A while loop: how many doublings until we pass 100 coins?
    var pile: int = 1
    var doublings: int = 0
    while pile <= 100:
        pile *= 2
        doublings += 1
    lines.append("A pile of 1 coin doubled %d times is %d coins." % [doublings, pile])
    lines.append("")

    # 5. match: pick a message by value.
    lines.append("Weather today: %s" % weather_message(2))
    lines.append("")

    # 6. Counting game.
    lines.append("Counting game: " + ", ".join(fizz_buzz(15)))

    report_label.text = "\n".join(lines)
    for line in lines:
        print(line)
```

Two new tools appear at the end. `", ".join(list)` glues an array of strings together with a separator between them. `"\n"` is the "new line" character, so joining with it puts each line on its own row.

10. Press **F5** (Run Project). If the editor asks for a main scene, choose `main.tscn` and **Select Current**. Check that the window shows the report and the Output panel shows the same lines.

11. Experiment: change a number in `coins_per_room`, set `"key"` to 0 in the backpack and run again. Does the door message change? Does the rank change?

Expected result

A window titled with your project name shows this text, and the same lines appear in the Output panel:

```
PIP'S TREASURE REPORT

Rooms visited: 5
Total coins: 27
Rank: Explorer

Room by room:
  Room 1: 3 coins (a few)
  Room 2: 7 coins (a nice pile)
  Room 3: empty room
  Room 4: 12 coins (jackpot!)
  Room 5: 5 coins (a nice pile)

Backpack:
  coin x12
  key x1
  potion (empty)
Door can open: true

A pile of 1 coin doubled 7 times is 128 coins.

Weather today: cloudy

Counting game: 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz
```

Figure 4.5: The finished Project 4 window: a plain Label showing the report built by the script.

```
PIP'S TREASURE REPORT

Rooms visited: 5
Total coins: 27
Rank: Explorer

Room by room:
  Room 1: 3 coins (a few)
  Room 2: 7 coins (a nice pile)
  Room 3: empty room
  Room 4: 12 coins (jackpot!)
  Room 5: 5 coins (a nice pile)

Backpack:
  coin x12
  key x1
  potion (empty)
Door can open: true

A pile of 1 coin doubled 7 times is 128 coins.

Weather today: cloudy

Counting game: 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz
```

If your output differs, compare it line by line with the finished project in `projects/ch04-decisions-and-loops/main.gd`.

Chapter summary

- A boolean is `true` or `false`. Comparisons (`==`, `!=`, `<`, `>`, `<=`, `>=`) produce booleans, and `and`, `or`, `not` combine them.
- `=` stores a value; `==` asks if two values are equal. Never mix them up.
- `if`, `elif` and `else` choose between paths. The first true condition wins, so put the most specific check first. Indentation marks the block.
- `match` compares one value against many patterns; `_` is the catch-all.
- `while` repeats while a condition holds; `for ... in range(...)` repeats a counted number of times or walks through a collection. `break` ends a loop and `continue` skips to the next iteration.
- A function (`func name(params) -> type:`) packages code under a name. Parameters bring data in, `return` sends a result out, and variables created inside live only there (scope).
- An array is an ordered list read by index starting at 0; a dictionary stores key-value pairs read by key. A `for` loop over a dictionary gives its keys.
- Small tests, `print` and reading the first line of an error solve most problems.

Review questions

1. What are the only two values a `bool` can hold?
2. What is the difference between `=` and `==`? Which one belongs inside an `if` condition?
3. What does `not (5 > 3)` evaluate to?
4. Given `var score: int = 85`, which line does this print?

```gdscript
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
elif score >= 70:
```

```
    print("C")
else:
    print("F")
```

(a) A (b) B (c) C (d) B and C

5. True or false: in a `match` statement, you must write `break` at the end of each pattern block to stop it from running into the next one.

6. How many times does the body of `for i in range(2, 8):` run, and what is the last value of `i`?
7. What is the difference between `break` and `continue`?
8. Write a function named `triple` that takes an `int` and returns three times its value, including the type annotations.
9. Given `var names: Array[String] = ["Pip", "Slime", "Door"]`, what does `names[1]` give, and what happens if you ask for `names[3]`?
10. A dictionary `var stats: Dictionary = {"hp": 10}` does not contain the key `"mana"`. What is a safe way to read mana so that you get 0 instead of an error?

Exercises

1. **Practice (easy):** Write a function `is_adult(age: int) -> bool` that returns `true` for age 18 and above. Call it with three different ages and print the results.
2. **Practice (easy):** Use a `for` loop and `range` to print the even numbers from 2 to 20. Do it once with `range(2, 21, 2)` and once with a normal `range(1, 21)` and an `if` using `%`.
3. **Apply (medium):** Make an array of five numbers. Write a function `largest(numbers: Array[int]) -> int` that returns the biggest one by looping and keeping a "best so far" variable. (Do not use the built-in `max`. The point is to practise the loop.)
4. **Apply (medium):** Extend Project 4. Add an `inventory_report()` function that takes a dictionary and returns an array of lines. Show a warning line such as "Low on potions!" when a count is 0. Add the lines to the report.
5. **Challenge (hard):** Write a "guess the number" program with no input: pick `var secret: int = 37` and write a `while` loop that tries guesses with a simple strategy (halving the range each time: 50, then 25 or 75, and so on) and prints how many guesses it needed. Hint: keep `low` and `high` variables and change one of them after each guess.

Common mistakes

- **Symptom:** the editor underlines an `if` line and says it expected a colon. **Cause:** you left off the `:` at the end of `if`, `elif`, `else`, `for`, `while` or `func`. **Fix:** add the colon.
- **Symptom:** a message about "unexpected indentation" or "expected indented block". **Cause:** a mix of tabs and spaces, or an empty block. **Fix:** use one tab per level, and put `pass` in a block you have not filled in.
- **Symptom:** `if x = 5:` is an error. **Cause:** `=` stores a value; you meant to compare. **Fix:** write `if x == 5:`.
- **Symptom:** the game freezes when a particular part runs. **Cause:** a `while` loop whose condition never becomes false. **Fix:** make sure something in the body changes the variable the condition checks.
- **Symptom:** "Index N out of bounds" when reading an array. **Cause:** arrays start at 0, so an array of 5 items has indexes 0 to 4. **Fix:** loop with `range(array.size())` or use `for item in array`.

- **Symptom:** the function returns `null` or the editor says "Not all code paths return a value". **Cause:** one branch of an `if` has no `return`. **Fix:** make sure every path ends with a `return`, such as an `else` at the bottom.

Further reading

- GDScript reference (basics, `if`, `match`, loops, functions): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
- GDScript format strings (`%` slots): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_format_string.html
- Class reference for `Array`: https://docs.godotengine.org/en/stable/classes/class_array.html
- Class reference for `Dictionary`: https://docs.godotengine.org/en/stable/classes/class_dictionary.html

Chapter 5: Scripts That Talk to the Game

Learning objectives

By the end of this chapter you will be able to:

- Explain when Godot calls `_ready()` and `_process(delta)` and choose the right one for a task.
- Find other nodes from a script using `$`, `get_node()`, `%UniqueName`, `get_parent()` and `get_tree()`.
- Read and change the properties of a node (position, rotation, color, text, visibility) from code.
- Use `@onready` and `@export` to prepare references and expose settings in the Inspector.
- Put nodes into groups and act on every member of a group at once.
- Connect built-in signals in the editor and in code, and declare and emit a custom signal.

Key terms

- **Script:** a text file of code attached to one node that gives that node extra behavior.
- **Callback:** a function that you write but that Godot calls for you at the right moment, such as `_ready()`.
- **Frame:** one picture drawn by the game. A game usually draws 60 or more frames each second.
- **`delta`:** the number of seconds that passed since the previous frame. It is passed to `_process()`.
- **Node path:** the "address" of a node in the scene tree, such as `Hud/StatusLabel` or `../Pip`.
- **`@onready`:** a marker that delays filling a variable until the node is ready, so the child nodes it points to exist.
- **Property:** a named value stored on a node, such as `position` or `text`. You can read and change it.
- **Method:** a function that belongs to a node or other object, called with a dot: `lamp.hide()`.
- **Group:** a tag you put on nodes so you can find or command all of them together.
- **Signal:** a message a node broadcasts when something happens, such as "button pressed" or "timer finished".
- **Emit:** to send a signal.
- **Connect:** to tell a signal which function to call when it is emitted.

5.1 From "a script" to "a game that reacts"

In Chapter 3 you attached a script to a node and printed values. In Chapter 4 you learned decisions, loops, functions, arrays and dictionaries. Those scripts all ran once and then finished. A game does something different: it **waits**, **reacts** and **keeps running**.

Think about a game of tag. Nobody gives the players a script of what to do second by second. Instead there are rules for events: when you are touched, you become "it"; when the whistle blows, the game ends. Godot works the same way. You write small pieces of code that Godot calls at the right moment:

- once, when a node is ready (`_ready`)
- over and over, once per frame (`_process`)
- when something happens, such as a button press (**signals**)

To make this work, scripts need to find and control other nodes. A script on the player might need the score label, a script on a button might need to change a door, and so on. This chapter is about those two skills: **being called at the right time** and **reaching other nodes**.

What a script attached to a node means

Recall that every script begins with a line saying which kind of node it can be attached to:

```
extends Sprite2D
```

That line means "this script *is* a Sprite2D, plus my own extras". Because of it, the script can use everything a Sprite2D has, such as `position`, `rotation` and `texture`, by name, without writing anything else. Inside the script, the node itself is called `self`, though you can almost always leave `self.` out.

```
extends Sprite2D
# Attached to a Sprite2D node.

func _ready() -> void:
    position = Vector2(100, 50)   # same as self.position = ...
    print("My name is ", name)    # 'name' is a property every node has
```

> **Note:** If you attach a script that says `extends Sprite2D` to a different kind of node, such as a Label, Godot reports an error. The `extends` line must match the node's type, or be one of its ancestors in the class hierarchy (for example, `extends Node2D` works on a Sprite2D because a Sprite2D is a Node2D).

5.2 Callbacks: `_ready` and `_process`

Godot calls certain functions automatically if you define them. They begin with an underscore. You do not call them yourself.

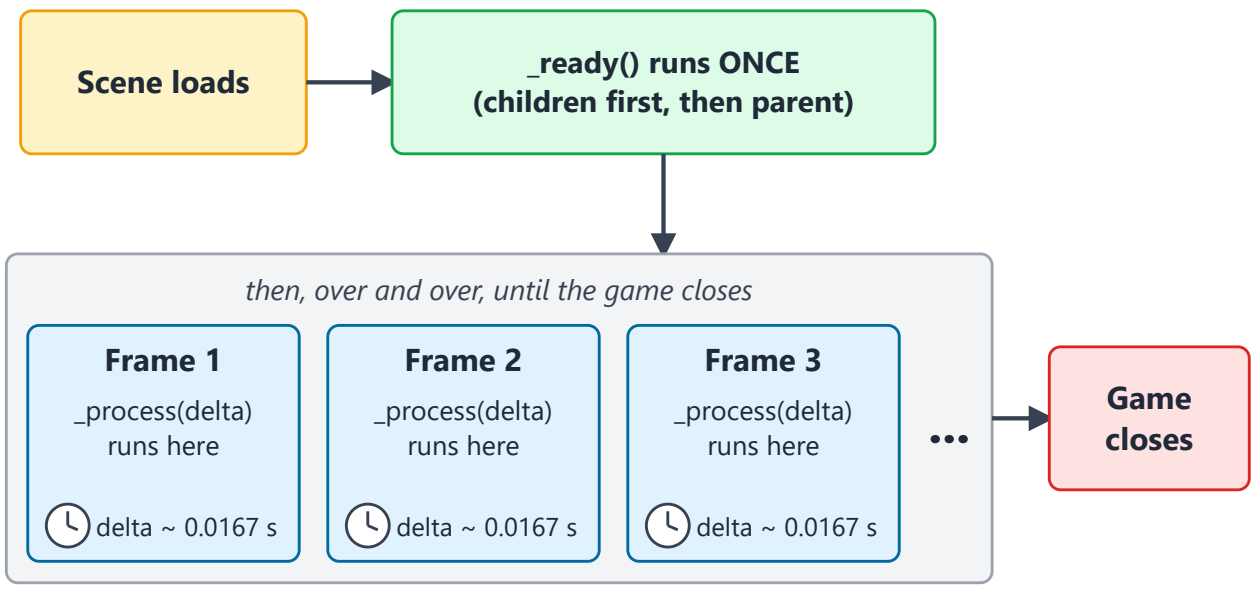


Figure 5.1: When a scene starts, Godot calls `_ready()` once on each node. After that, `_process(delta)` is called on every node once per frame until the game ends.

`_ready()`: set up once

`_ready()` runs one time, when the node and **all of its children** have been added to the scene tree and are ready to use. Use it for start-up work: setting an initial value, finding nodes, connecting signals.

```gdscript
func _ready() -> void:
    print("I am ready!")
```

Order matters here, and it surprises people: children become ready **before** their parent. If a root node has two children, the output order is child one, child two, then the root. That is useful: when the root's `_ready()` runs, you can be sure every child exists and has already set itself up.

`_process(delta)`: do something every frame

`_process` runs on every frame. At 60 frames per second it runs 60 times a second.

```gdscript
extends Sprite2D
# Attached to a Sprite2D. Spins the sprite forever.

func _process(delta: float) -> void:
    rotation += 1.0 * delta
```

The parameter `delta` is the time, in seconds, since the previous frame. At 60 frames per second it is about `0.0167`. Why use it? Because not every computer draws the same number of frames. Compare two ways of moving something 60 pixels per second:

Method	At 60 fps	At 30 fps	Same speed?
`x += 1` every frame	60 px/s	30 px/s	No
`x += 60 * delta` every frame	60 px/s	60 px/s	Yes

The second approach multiplies a **speed** (units per second) by the **time** that passed, and so gives the same distance per second on slow and fast computers. Memorize this rule: **inside `_process`, always multiply by `delta` when something changes continuously.** Chapter 7 builds on it.

Rotation in Godot is measured in **radians**, not degrees. A full turn is about 6.28 (written `TAU` in code), so `rotation += 1.0 * delta` takes about 6.3 seconds to spin around once. If you prefer degrees, the property `rotation_degrees` accepts them.

> **Warning:** `_process` runs *constantly*. Do not do heavy work in it (loading files, searching the whole tree) unless you must. And never `print()` inside it unless you are debugging: you will flood the Output panel with sixty lines a second.

`_physics_process(delta)`: a close cousin

There is a second per-frame callback, `_physics_process(delta)`. It runs at a steady rate (60 times per second by default) and is the right place for movement that involves collisions. You will use it from Chapter 7 on. For now: `_process` for visuals and general updates, `_physics_process` for physics and character movement.

Unused parameters

If you declare `delta` but never use it, Godot shows a harmless warning. Put an underscore in front of the name (`_delta`) to say "I know I am not using it":

```gdscript
func _process(_delta: float) -> void:
    pass
```

5.3 Finding other nodes

A script often needs to talk to a node other than the one it is attached to. First you must **find** it. Godot gives you several ways, all based on the scene tree.

The scene tree is a family tree

Consider the scene we will build in this chapter's project:

```
Main (Node2D)
├─ Pip (Sprite2D)
├─ Lamps (Node2D)
│  ├─ Lamp1 (ColorRect)
│  ├─ Lamp2 (ColorRect)
│  └─ Lamp3 (ColorRect)
├─ BlinkTimer (Timer)
└─ Hud (CanvasLayer)
   ├─ StatusLabel (Label)
   ├─ ToggleButton (Button)
   └─ ResetButton (Button)
```

A **node path** is the address of a node, written from the node that is asking. Slashes separate steps down the tree, `..` means "go up to the parent".

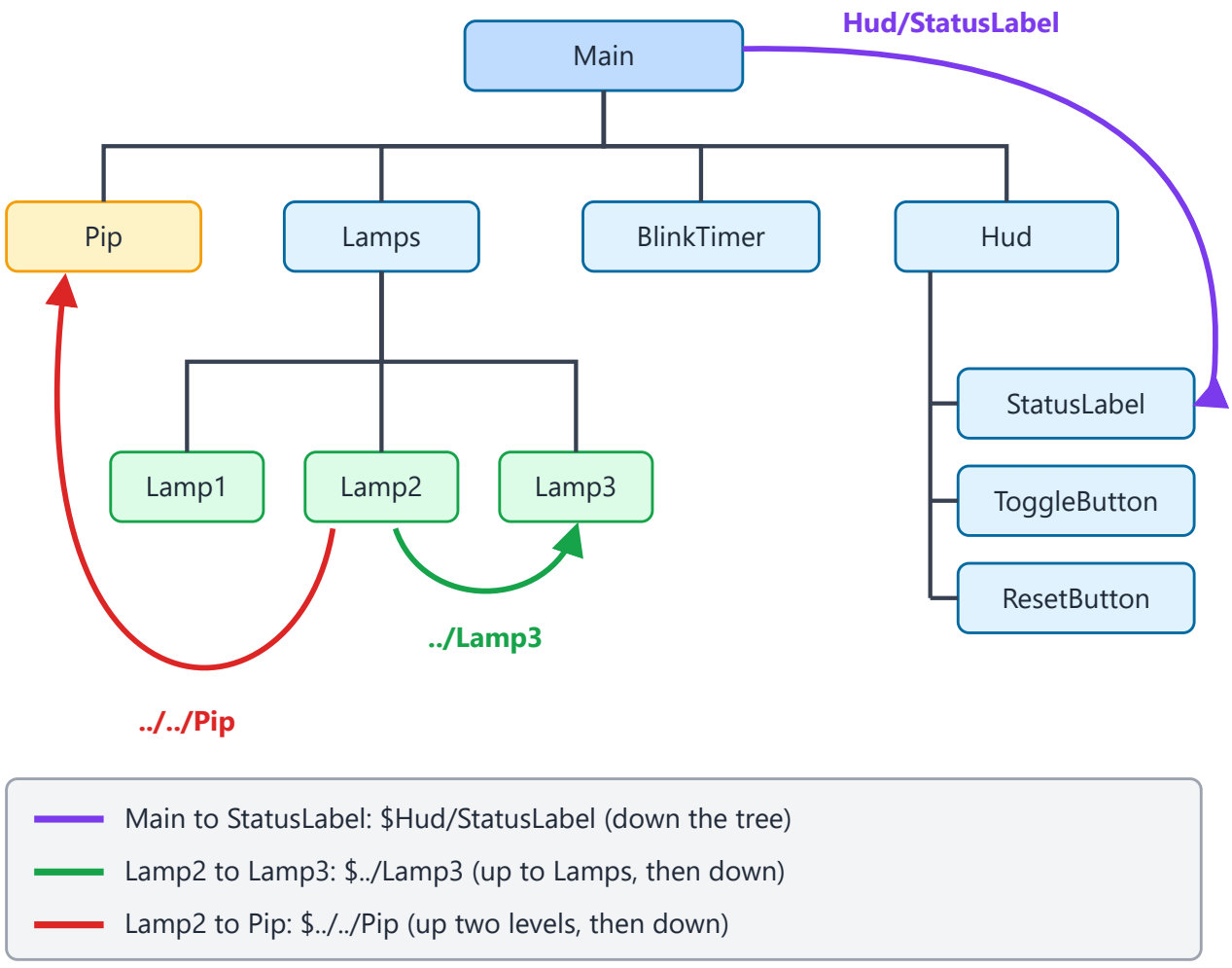


Figure 5.2: Paths start from the node holding the script. Going down uses names; going up uses `..`.

From the script on **Main**:

To reach	Path
Pip	`Pip`
Lamp2	`Lamps/Lamp2`
StatusLabel	`Hud/StatusLabel`

From a script on **Lamp2**:

To reach	Path
its parent (Lamps)	`..`
Lamp3, its sibling	`../Lamp3`
Pip	`../../Pip`

`get_node()` and the `$` shortcut

`get_node("path")` returns the node at that path. Because looking up nodes is so common, GDScript gives you a shorter form with a dollar sign:

```
var hud_label: Label = get_node("Hud/StatusLabel")
var same_label: Label = $Hud/StatusLabel
```

Both lines do the same thing. If the path contains spaces or `..`, put it in quotes after the dollar sign: `$"../Pip"`.

After you have the node, use a dot to reach its properties and methods:

```
$Hud/StatusLabel.text = "Hello"
$Pip.visible = false
```

The `@onready` pattern

Here is a trap. Suppose you write this at the top of a script:

```
extends Node2D

var status_label: Label = $Hud/StatusLabel   # PROBLEM
```

Variables at the top of the script are filled in **when the script object is created**, which is before the node has joined the scene tree and before its children exist. The lookup fails and the variable is `null` (meaning "nothing"). Later, using it causes an error such as "Invalid set index 'text' (on base: 'null instance')".

The fix is `@onready`:

```
extends Node2D

@onready var status_label: Label = $Hud/StatusLabel   # filled in just before _ready()
```

`@onready` means "fill this variable at the moment the node is ready, right before `_ready()` runs". At that moment the children exist. This is the standard way to store a reference to a child node, and you will write it in almost every script.

Typing the variable (`: Label`) is optional but recommended. It tells the editor what the node is, so it can auto-complete `status_label.` with the label's properties and warn you about typing mistakes.

Unique names with `%`

Long paths such as `Hud/StatusLabel` break when you rearrange the tree. If you move the label into a new container, every path that mentions it must change. Godot has a remedy: **unique names**.

1. Right-click a node in the Scene dock and choose **Access as Unique Name**. A small `%` icon appears next to it.
2. In scripts, find that node from anywhere in the same scene with `%Name`:

```
@onready var status_label: Label = %StatusLabel
```

No path needed. The name must be unique within the scene. In this book we use `%` for nodes that are likely to move and `$` for simple direct children.

Other ways to move around the tree

Code	What it gives you
`get_parent()`	the node directly above this one
`get_children()`	an array of all direct children
`get_child(0)`	the first child (by position)
`get_tree()`	the whole scene tree, which has handy tools such as groups and pausing
`get_node_or_null("Name")`	the node, or `null` if it does not exist (no error)
`owner`	the root node of the scene this node was saved in

A script that reaches **up** into its parent is less flexible than one that works on its own. A common guideline is "call down, signal up": a parent may call methods on its children, but a child should announce what happened with a signal (section 5.7) and let the parent decide what to do.

Checking that a node exists

If a node might not exist, use `get_node_or_null` and test the result:

```
var bonus: Node = get_node_or_null("BonusItem")
if bonus != null:
    bonus.queue_free()   # remove it from the game
```

`queue_free()` deletes a node safely at the end of the current frame. You will use it for coins, enemies and bullets.

5.4 Properties and methods: controlling nodes from code

Everything you can change in the Inspector can be changed from code, with the same names (written in snake_case). If you can see `Position` in the Inspector, the script property is `position`. If you see `Modulate`, it is `modulate`.

```
extends Node2D
# Attached to the root of a scene that has a Sprite2D child named "Sprite2D".
```

```gdscript
@onready var sprite: Sprite2D = $Sprite2D

func _ready() -> void:
    sprite.position = Vector2(200, 120)    # move it
    sprite.rotation_degrees = 45.0         # turn it
    sprite.scale = Vector2(2, 2)           # make it twice as big
    sprite.modulate = Color(1, 0.5, 0.5)   # tint it pink
    sprite.visible = true
```

The Inspector fields and the script properties are the *same values*. Changing one changes the other when the game runs. (Changes made by a running game are not saved back to the scene; they live only while the game runs.)

Methods: asking a node to do something

A property is a value. A **method** is an action. You call a method with a dot, then parentheses:

```gdscript
$Pip.hide()                # make it invisible
$Pip.show()                # visible again
$Pip.queue_free()          # delete it
$Pip.look_at(Vector2.ZERO)  # turn to face a point (Node2D method)
```

To discover which properties and methods a node has, use the built-in help: put the text cursor on a word such as `Sprite2D` in the script and press **Ctrl+click**, or press **F1** and search. The editor also lists them as you type after the dot.

Colors

Several examples use `Color`. A color has red, green and blue parts from 0.0 to 1.0, and an optional fourth number for transparency:

```gdscript
var yellow := Color(1.0, 0.85, 0.2)         # red, green, blue
var see_through := Color(1, 1, 1, 0.5)      # half transparent white
var named: Color = Color.RED                # a ready-made name
```

The shorthand `:=` in `var yellow := Color(...)` means "figure out the type from the value on the right". It is the same as writing `var yellow: Color = Color(...)`.

Constants

A value that never changes can be a **constant**. By convention its name is in capitals:

```gdscript
const LIT_COLOR := Color(1.0, 0.85, 0.2)
```

Trying to change a constant later is an error. Constants make scripts clearer: `LIT_COLOR` tells the reader more than three loose numbers.

`@export`: settings in the Inspector

You can show a script variable in the Inspector so you can edit it without opening the script. Put `@export` in front:

```gdscript
extends Sprite2D
# Attached to a Sprite2D.
```

```gdscript
@export var spin_speed: float = 1.0   # now appears in the Inspector under the script's name

func _process(delta: float) -> void:
    rotation += spin_speed * delta
```

Save the script, select the node, and look at the Inspector: a new field "Spin Speed" appears. Change it there and the node spins faster, with no code edits. This is how you build objects whose designer (you, or a friend) can tweak numbers freely. The value you type in the Inspector overrides the default in the script for that node.

Typed variables get appropriate editors: a `float` gets a number box, a `Color` gets a color picker, a `bool` gets a checkbox.

5.5 Calling functions on other scripts

When a node has its own script with custom functions, you can call those from outside. Imagine `pip.gd`:

```gdscript
class_name Pip
extends Sprite2D
# Attached to Pip (Sprite2D).

@export var calm_speed: float = 1.0
@export var excited_speed: float = 8.0

var spin_speed: float = 1.0


func _ready() -> void:
    spin_speed = calm_speed


func _process(delta: float) -> void:
    rotation += spin_speed * delta


func set_excited(value: bool) -> void:
    if value:
        spin_speed = excited_speed
    else:
        spin_speed = calm_speed
```

The line `class_name Pip` gives this script a global **type name**. Now any other script can say `@onready var pip: Pip = %Pip` and Godot knows `pip` has the method `set_excited`. Call it just like a built-in method:

```gdscript
pip.set_excited(true)
```

> **Note:** If you leave out the type (or declare it as `Sprite2D`), the editor does not know about `set_excited` and reports an error. The `class_name` line is what makes your scripts into new node types. Never choose a name that Godot already uses, such as `Sprite2D`, `Timer` or `Button`. If the editor complains that a class already exists, pick a different name.

5.6 Groups: naming a crowd

Sometimes you need to command many nodes at once: turn off every light, hide every coin, hurt every enemy. Looping through the tree by paths would be painful. A **group** is a tag. Any node can be in any number of groups, and you can then ask the scene tree for everyone with a given tag.

Adding nodes to a group

Two ways:

- **In the editor.** Select the node, open the **Groups** tab (next to the Inspector), type a name (for example `lamps`), and press **Add**. Do that for each node.
- **In code**, usually in `_ready`:

```
func _ready() -> void:
    add_to_group("lamps")
```

Using a group

```
# Get an array of every node in the group, then loop over it:
for lamp in get_tree().get_nodes_in_group("lamps"):
    lamp.hide()

# Or call one method on every member at once:
get_tree().call_group("lamps", "hide")

# Ask one node whether it belongs:
if is_in_group("lamps"):
    print("I am a lamp")

# Leave a group:
remove_from_group("lamps")
```

`get_nodes_in_group` returns members in tree order. `call_group` is shorthand for the loop-and-call pattern: it calls the method named in the string on every member that has that method. Because the name is a string, a typing mistake in it is not caught until the game runs, so test it.

> **Why it works:** Groups let the code ask a *question about the game* ("who are the lamps?") instead of knowing the exact tree. You can add a fourth lamp anywhere, tag it, and the existing code handles it without being edited.

5.7 Signals: letting nodes announce things

The idea

A button does not know what should happen when you click it. It could open a door in one game and quit in another. Hard-wiring the button to the door would make it useless elsewhere. Instead, the button just **announces**: "I was pressed." Anyone who cares can listen.

That announcement is a **signal**. The pattern has three parts:

1. A node **emits** a signal when something happens.
2. Another node **connects** to it, saying "when that signal happens, call this function of mine".
3. The connected function, called the **handler** or **receiver**, runs.

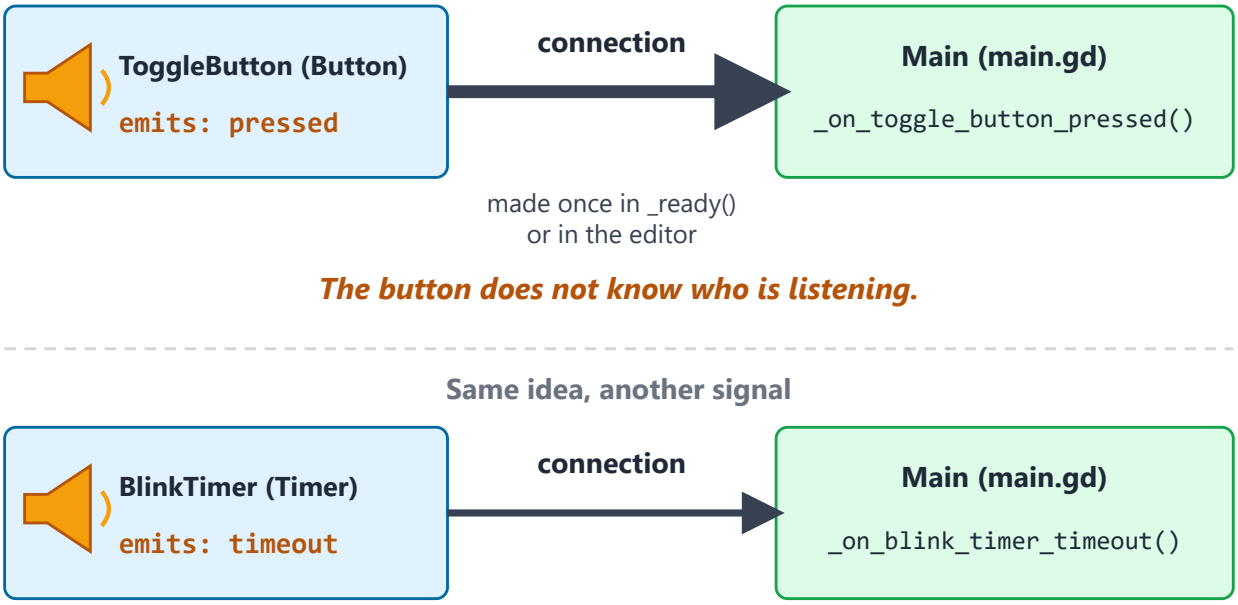


Figure 5.3: The button emits `pressed`. The connection, made once in the editor or in `_ready()`, sends it to the function in Main. The button knows nothing about Main.

A real-life version: you subscribe to a newsletter. The sender does not know your name or what you will do with the issue. It publishes; subscribers react.

Built-in signals

Many nodes already have useful signals. A few you will use soon:

Node	Signal	Emitted when
`Button`	`pressed`	the button is clicked
`Timer`	`timeout`	the countdown reaches zero
`Area2D`	`body_entered(body)`	a physics body enters it (Chapter 8)
`AnimationPlayer`	`animation_finished(name)`	an animation ends (Chapter 10)
`Node`	`tree_exiting`	the node is about to leave the tree

Find the full list for any node in its class reference, under "Signals", or in the editor's **Signals** tab.

Connecting in the editor

This is the quickest way for a beginner and it is visual. Suppose the scene has a `Timer` named `BlinkTimer` and the root script has a function to react.

1. Select the node that emits the signal (**BlinkTimer**).
2. Click the **Signals** tab (beside the Inspector).
3. Double-click the signal you want (**timeout()**). The "Connect a Signal to a Method" window opens.
4. In that window, choose the node that should *receive* it (the one whose script has the handler, here **Main**). Godot suggests a function name such as `_on_blink_timer_timeout`.

5. Press **Connect**. Godot writes an empty function for you in the script and shows a green icon next to the signal.

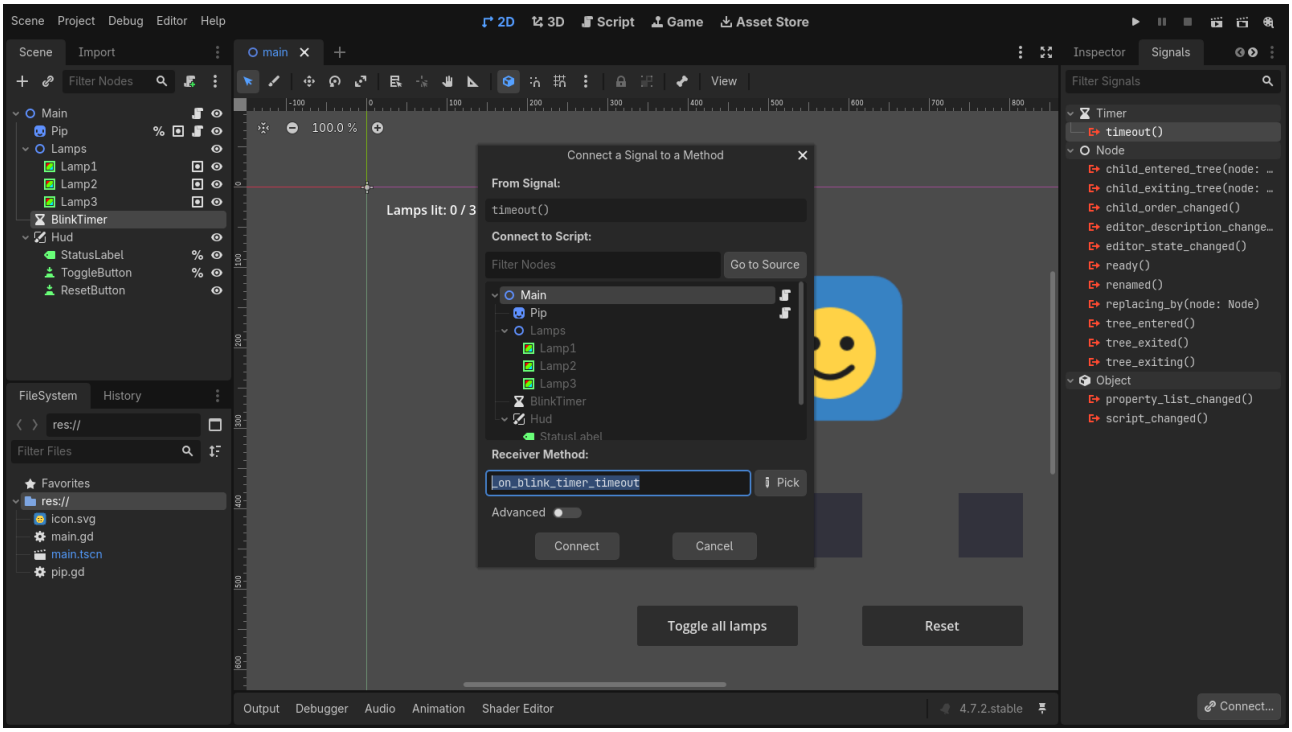


Figure 5.4: The Signals tab lists every signal the selected node can emit. Double-click one to connect it.

The generated function looks like this:

```
func _on_blink_timer_timeout() -> void:
    pass  # replace with your code
```

The name pattern is `_on_` + the emitting node's name in snake_case + `_` + the signal name. Godot does not require that name; it is a convention that helps you read the script.

> **Warning:** If you rename or delete the handler function later without disconnecting, the editor shows a red icon in the Signals tab and the game prints an error. Use the Signals tab to repair or remove the connection.

Connecting in code

You can make the same connection with one line, usually in `_ready()`:

```
@onready var toggle_button: Button = %ToggleButton

func _ready() -> void:
    toggle_button.pressed.connect(_on_toggle_button_pressed)

func _on_toggle_button_pressed() -> void:
    print("The button was pressed!")
```

Read the line left to right: "the button's `pressed` signal… connect to… my function `_on_toggle_button_pressed`". Note that there are **no parentheses** after the function name in `connect(...)`. You are handing over the function itself, not calling it.

Use the editor for nodes that live in the scene, and code when the node is created while the game runs (for example, each spawned enemy).

Signals with arguments

Some signals carry data. The handler must accept the same parameters:

```
func _on_something_happened(value: int) -> void:
    print("It sent me ", value)
```

When you connect in the editor, Godot builds the right parameter list for you.

Your own signals

You can declare a signal in any script with the `signal` keyword, and emit it with `.emit()`:

```
extends Node2D
# Attached to Main.

signal lit_count_changed(count: int)   # declare: this node can announce this

func _announce(count: int) -> void:
    lit_count_changed.emit(count)      # send it, with a value
```

Now anybody can connect to `lit_count_changed`, including the script itself:

```
func _ready() -> void:
    lit_count_changed.connect(_on_lit_count_changed)

func _on_lit_count_changed(count: int) -> void:
    print("Lit lamps: ", count)
```

Naming tip: name signals with a **past-tense verb** describing what already happened (`pressed`, `died`, `coin_collected`). That makes the connection read like a sentence. Chapter 15 returns to custom signals and builds larger structures from them.

Disconnecting and checking

You rarely need to, but you can:

```
toggle_button.pressed.disconnect(_on_toggle_button_pressed)
print(toggle_button.pressed.is_connected(_on_toggle_button_pressed))   # false
```

A glimpse of `await`

A signal can also be waited for. The keyword `await` pauses a function until a signal fires, then continues from that line:

```
func countdown() -> void:
    print("Starting...")
    await get_tree().create_timer(2.0).timeout
    print("Two seconds later!")
```

`get_tree().create_timer(2.0)` makes a one-shot timer, and `.timeout` is its signal. This is a handy trick for quick delays. It is the Godot 4 replacement for the old `yield`.

5.8 When things go wrong

A few errors will appear in this chapter. Recognize them and you will fix them in seconds.

Error	Likely cause	Fix
`Node not found: "Hud/StatusLabel"`	The path is wrong or the node was renamed	Compare the path with the Scene dock, letter by letter. Names are case-sensitive
`Invalid access to property ... on a base object of type 'Nil'` / `null instance`	A variable is `null`: the node was not found, or you used it before `@onready` filled it	Use `@onready`, check the path, check the node exists
`Nonexistent function 'xyz' in base 'Sprite2D'`	You called a method that the node (or its script) does not have	Check the spelling; for your own methods, add `class_name` and type the variable
Signal handler never runs	Signal not connected, or connected to a different node	Look at the Signals tab for a green icon; print inside the handler

A friendly trick for finding a node problem: press **F5**, then in the editor's Scene dock click the **Remote** tab. It shows the *live* tree of the running game, including nodes created by code, so you can see what actually exists. Chapter 22 covers the debugger in depth.

Project 5: Lamp Room

Goal: build a scene where code and signals bring the nodes to life. Three lamps blink one by one on a timer, one button lights or darkens all lamps at once, another button resets them, a label reports how many are lit, and Pip spins faster when all three are on.

This uses `_process`, `@onready`, `@export`, `%UniqueName`, groups, a built-in signal connected in code, two signals connected in the editor, and a custom signal.

Time: about 60 minutes.

Finished copy: `projects/ch05-scripts-and-signals/`.

Files:

```
ch05-scripts-and-signals/
├─ project.godot
├─ icon.svg          (Godot's default icon works; a face drawn in SVG is included)
├─ main.tscn
├─ main.gd
└─ pip.gd
```

Steps

Part A: build the scene

1. Create a new project named `Ch05 Scripts and Signals`.
2. In the Scene dock, add a **Node2D** root node and rename it **Main**.
3. Add a **Sprite2D** child and rename it **Pip**. In the Inspector set **Texture** to `icon.svg` (drag the file from the FileSystem dock onto the Texture slot). Set **Position** to (576, 200) and **Scale** to (1.5, 1.5).
4. Right-click **Pip** and choose **Access as Unique Name** so a `%` appears.

5. Add a second **Node2D** child of **Main** and rename it **Lamps**.
6. Under **Lamps**, add three **ColorRect** nodes named **Lamp1**, **Lamp2** and **Lamp3**. For each, set **Size** to (80, 80), **Color** to a dark blue-gray (R 0.2, G 0.2, B 0.25), and positions of (336, 380), (536, 380) and (736, 380). (Position fields for these are in the Inspector under **Layout**.)
7. For each lamp, open the **Groups** tab next to the Inspector, type `lamps` and click **Add**. Lamp1, Lamp2 and Lamp3 must all be in the group.
8. Select **Pip** and add it to a group named `heroes` the same way. (We will not use it in code; it is practice.)
9. Add a **Timer** child of **Main**, named **BlinkTimer**. Set **Wait Time** to 1 and tick **Autostart**.
10. Add a **CanvasLayer** child of **Main** named **Hud**. A CanvasLayer draws on top of the world and is perfect for interface pieces. Under it, add:
    - a **Label** named **StatusLabel** at position (24, 16) with text `Lamps lit: 0 / 3`
    - a **Button** named **ToggleButton** at (336, 520), size (200, 50), text `Toggle all lamps`
    - a **Button** named **ResetButton** at (616, 520), size (200, 50), text `Reset`
11. Make **StatusLabel** and **ToggleButton** unique names (right-click, **Access as Unique Name**).

The finished tree:

```
Main (Node2D)
├─ Pip (Sprite2D)              [unique name, group "heroes"]
├─ Lamps (Node2D)
│  ├─ Lamp1 (ColorRect)        [group "lamps"]
│  ├─ Lamp2 (ColorRect)        [group "lamps"]
│  └─ Lamp3 (ColorRect)        [group "lamps"]
├─ BlinkTimer (Timer)
└─ Hud (CanvasLayer)
   ├─ StatusLabel (Label)      [unique name]
   ├─ ToggleButton (Button)    [unique name]
   └─ ResetButton (Button)
```

12. Save the scene as `main.tscn`, and in **Project > Project Settings > Application > Run** set **Main Scene** to it (or accept the prompt when you first press F5).

Part B: Pip's script

13. Select **Pip**, click **Attach Script**, name it `pip.gd` and replace the contents with:

```gdscript
class_name Pip
extends Sprite2D
# Attached to the node "Pip" (type Sprite2D) in main.tscn.

# How fast Pip spins, in radians per second. Editable in the Inspector.
@export var calm_speed: float = 1.0
@export var excited_speed: float = 8.0

var spin_speed: float = 1.0


func _ready() -> void:
    spin_speed = calm_speed


func _process(delta: float) -> void:
    rotation += spin_speed * delta


func set_excited(value: bool) -> void:
    if value:
```

```
        spin_speed = excited_speed
    else:
        spin_speed = calm_speed
```

14. Press **F6** (Run Current Scene) or F5. Pip should slowly spin. Stop the game and, in the Inspector for **Pip**, change **Calm Speed** to 3 and run again. Pip spins faster, with no script change.

Part C: Main's script

15. Select **Main**, attach a new script `main.gd`, and write it in sections. First the top:

```gdscript
extends Node2D
# Attached to the root node "Main" (type Node2D) in main.tscn.

# A custom signal: "the number of lit lamps changed."
signal lit_count_changed(count: int)

const LIT_COLOR := Color(1.0, 0.85, 0.2)
const DARK_COLOR := Color(0.2, 0.2, 0.25)

@onready var pip: Pip = %Pip
@onready var status_label: Label = %StatusLabel
@onready var toggle_button: Button = %ToggleButton

var next_lamp: int = 0


func _ready() -> void:
    # Two connections from code. (BlinkTimer and ResetButton are connected in the editor.)
    toggle_button.pressed.connect(_on_toggle_button_pressed)
    lit_count_changed.connect(_on_lit_count_changed)
    _announce()
```

16. Add the helper functions. These read and change the lamps through the group:

```gdscript
func _announce() -> void:
    lit_count_changed.emit(_count_lit())


func _count_lit() -> int:
    var count: int = 0
    for node in get_tree().get_nodes_in_group("lamps"):
        if _is_lit(node as ColorRect):
            count += 1
    return count


func _is_lit(lamp: ColorRect) -> bool:
    return lamp.color == LIT_COLOR


func _set_lit(lamp: ColorRect, lit: bool) -> void:
    lamp.color = LIT_COLOR if lit else DARK_COLOR
```

`node as ColorRect` tells GDScript "treat this node as a ColorRect", which lets the typed helper functions accept it.

17. Add the handlers. These two are used by the built-in signals you connect in code and by your custom signal:

```gdscript
func _on_toggle_button_pressed() -> void:
    # If any lamp is dark, light them all; otherwise turn them all off.
    var all_lit: bool = _count_lit() == get_tree().get_nodes_in_group("lamps").size()
    for node in get_tree().get_nodes_in_group("lamps"):
```

```
        _set_lit(node as ColorRect, not all_lit)
    _announce()


func _on_lit_count_changed(count: int) -> void:
    var total: int = get_tree().get_nodes_in_group("lamps").size()
    status_label.text = "Lamps lit: %d / %d" % [count, total]
    # Pip gets excited when every lamp is on.
    pip.set_excited(count == total)
```

Part D: connect two signals in the editor

18. Select **BlinkTimer**, open **Node > Signals**, double-click **timeout()**, choose **Main** as the receiver, accept the name `_on_blink_timer_timeout` and press **Connect**. Godot adds the function to `main.gd`. Replace its body:

    ```
    func _on_blink_timer_timeout() -> void:
        # Every second, flip the next lamp in the row.
        var lamps: Array[Node] = get_tree().get_nodes_in_group("lamps")
        var lamp: ColorRect = lamps[next_lamp] as ColorRect
        _set_lit(lamp, not _is_lit(lamp))
        next_lamp = (next_lamp + 1) % lamps.size()
        _announce()
    ```

    The line with `%` wraps the index back to 0 after the last lamp. (Modulo, from Chapter 4.)

19. Select **ResetButton**, double-click its **pressed()** signal, receiver **Main**, **Connect**, and write the handler:

    ```
    func _on_reset_button_pressed() -> void:
        for node in get_tree().get_nodes_in_group("lamps"):
            _set_lit(node as ColorRect, false)
        next_lamp = 0
        _announce()
    ```

20. Run the project (**F5**). Test each behavior in turn:

    - Each second, one lamp flips on or off, going left to right and around again.
    - **Toggle all lamps** lights all of them (or turns all off if all are lit).
    - The label always shows the right count.
    - When all three are lit, Pip spins noticeably faster; when not, slowly.
    - **Reset** turns everything off and starts the blinking again from the first lamp.

Expected result

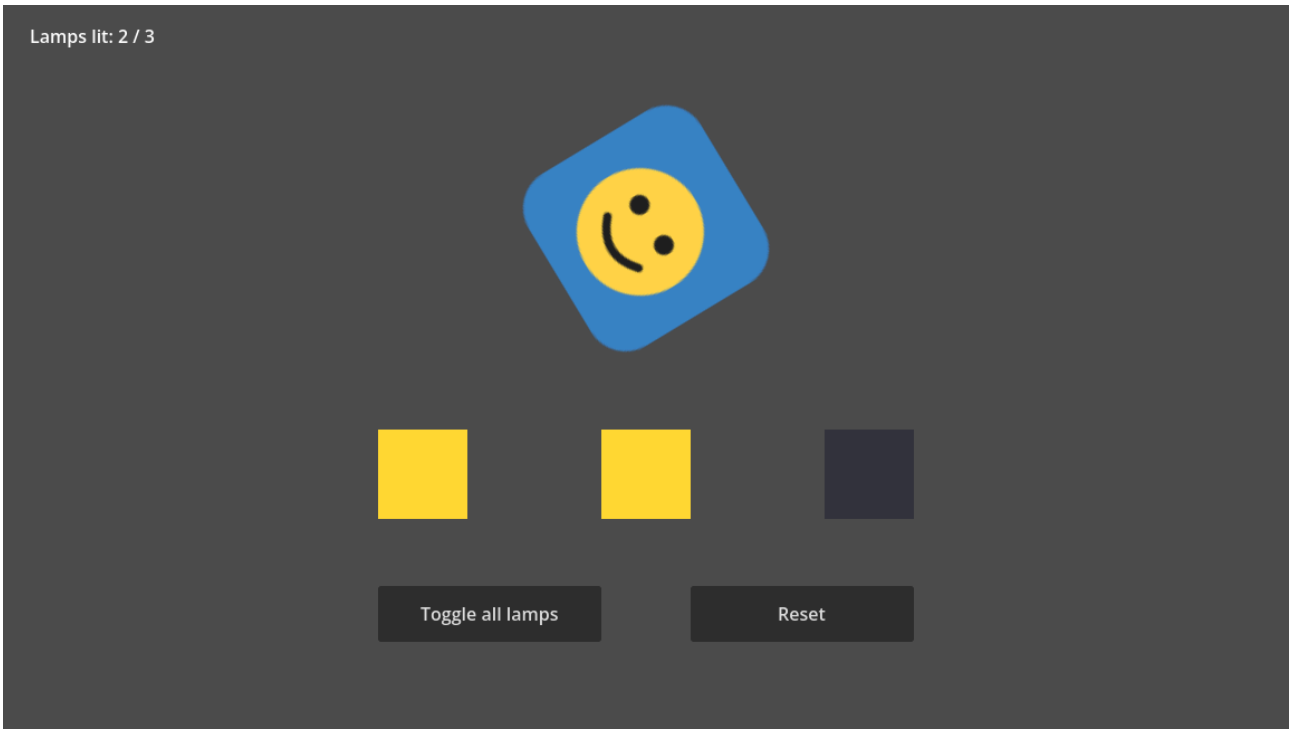


Figure 5.5: Lamp Room running. Two lamps are lit, the label reads "Lamps lit: 2 / 3", and Pip turns slowly.

You have used both ways to connect signals, a custom signal, a group, unique names, `_process` and `@export` in one small game. Notice also that the buttons and timer do not know about the lamps: all decisions live in `main.gd`.

Chapter summary

- Godot calls callbacks for you. `_ready()` runs once when the node and its children are ready; `_process(delta)` runs every frame.
- Multiply continuous changes by `delta` so speed does not depend on the frame rate.
- Find nodes with `$Path`, `get_node()`, `%UniqueName`, `get_parent()` and `get_tree()`. Store child references with `@onready`, otherwise they are `null`.
- Node properties in the Inspector are the same as script properties (`position`, `rotation`, `color`, `text`, `visible`). Methods such as `hide()` and `queue_free()` ask a node to act.
- `@export` shows a variable in the Inspector; `class_name` gives a script a type name other scripts can use.
- Groups are tags: `add_to_group`, `get_tree().get_nodes_in_group()` and `call_group()` let you work on every member at once.
- A signal is an announcement. Connect it in the editor (Signals tab) or in code (`node.signal_name.connect(function)`). Declare your own with `signal` and send it with `.emit()`.
- "Call down, signal up": parents call methods on children; children announce events with signals.

Review questions

1. When does Godot call `_ready()`, and how many times?
2. A script holds the line `rotation += 2` inside `_process(delta)`. Why can this make the object spin at different speeds on different computers, and how do you fix it?

3. In what order do `_ready()` functions run for a parent node with two children: parent first, or children first?

4. Which line correctly stores a reference to a child Label named `ScoreLabel` so that it is ready to use inside `_ready()`? (a) `var score_label = $ScoreLabel` (b) `@onready var score_label: Label = $ScoreLabel` (c) `@export var score_label = $ScoreLabel` (d) `func score_label = $ScoreLabel`

5. A script is on `Lamp2`. Which path reaches its sibling `Lamp3` (both are children of `Lamps`)?

6. What is the advantage of `%Name` (a unique name) over a path like `Hud/Panel/Row/StatusLabel`?
7. True or false: a node can belong to more than one group.
8. What does `get_tree().call_group("enemies", "hide")` do?
9. What is wrong with `button.pressed.connect(_on_pressed())`?
10. Describe, in your own words, the difference between a signal being *emitted* and a signal being *connected*.

Exercises

1. **Practice (easy):** Add a `Label` to any scene and attach a script that changes its text to "Hello from code" in `_ready()`. Then add `@export var message: String = "Hello"` and show that text instead. Change it from the Inspector without touching the script.
2. **Practice (easy):** In the Lamp Room project, make the timer faster by changing **Wait Time** in the Inspector to 0.4. Then change it from code in `_ready()` with `$BlinkTimer.wait_time = 0.4`. Check that both work.
3. **Apply (medium):** Add a fourth lamp to the Lamp Room (copy Lamp3 with **Ctrl+D**, move it, check it is in the `lamps` group). Run it. Does everything still work without a single code change? Explain why.
4. **Apply (medium):** Add a third button, **Hide Pip**, that toggles Pip's `visible` property. Connect it in the editor. Use `pip.visible = not pip.visible`.
5. **Challenge (hard):** Make a custom signal `all_lit` on Main that is emitted only when the count reaches the total. Connect it to a new function that changes the status label's text to "All lit!" and gives the label a green tint with `status_label.modulate`. Make sure the tint is removed when a lamp goes dark again.

Common mistakes

- **Symptom:** "Invalid access to property or key ... on a base object of type 'Nil'" (or "null instance") the moment the game starts. **Cause:** a variable pointing at a child node was filled before the node existed (no `@onready`) or the path is wrong. **Fix:** write `@onready var x: Type = $Path` and check the Scene dock for the exact name.
- **Symptom:** `Node not found` error. **Cause:** a typing mistake, wrong letter case, or the node moved. **Fix:** copy the path from the Scene dock (right-click, **Copy Node Path**), or use a unique name.
- **Symptom:** the thing spins or slides faster on a faster computer. **Cause:** a continuous change in `_process` without `delta`. **Fix:** multiply the speed by `delta`.
- **Symptom:** nothing happens when you click the button. **Cause:** the signal was never connected, or you connected `pressed` on a different node. **Fix:** look at the Signals tab for the green connection icon, and add a `print` to the handler to see if it runs.
- **Symptom:** the Output panel is filled with endless lines of text. **Cause:** a `print` left inside `_process`. **Fix:** delete it once you have finished debugging.
- **Symptom:** an error about the `connect` call and "Cannot convert argument". **Cause:** you wrote `connect(_on_pressed())` with parentheses, which calls the function, instead of passing it. **Fix:** write `connect(_on_pressed)`.

Further reading

- Scripting languages and GDScript basics: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
- Nodes and scene instances / scene organization: https://docs.godotengine.org/en/stable/tutorials/scripting/nodes_and_scene_instances.html
- Groups: https://docs.godotengine.org/en/stable/tutorials/scripting/groups.html
- Using signals: https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html
- Idle and physics processing: https://docs.godotengine.org/en/stable/tutorials/scripting/idle_and_physics_processing.html

Part II: Making 2D games

Chapter 6: 2D Space and Sprites

Learning objectives

By the end of this chapter you will be able to:

- Describe the 2D coordinate system Godot uses, including why "up" has a negative y value.
- Place, rotate and scale a `Node2D` and explain how a child's position depends on its parent.
- Show pictures with `Sprite2D`, including regions of a texture and sprite-sheet frames.
- Control which sprite is drawn on top of another with tree order and `z_index`.
- Add a `Camera2D` that follows a character, zoom it, and keep interface elements still with a `CanvasLayer`.
- Draw simple shapes with `_draw()`, `Polygon2D` and `Line2D`, without any image files.

Key terms

- **Coordinates:** a pair of numbers (x, y) that name a point in 2D space.
- **Origin:** the point (0, 0), where the x and y axes meet.
- **Vector2:** a value holding two numbers, x and y. Used for positions, sizes and directions.
- **Pixel:** one tiny square dot of the screen. Godot's 2D units are pixels by default.
- **Transform:** the combination of position, rotation and scale that places a node.
- **Local space / global space:** a position measured from the parent (local) or from the world origin (global).
- **Texture:** an image stored in memory that can be drawn on screen.
- **Sprite:** a texture drawn as a game object. In Godot, a `Sprite2D` node.
- **Sprite sheet:** one image containing many frames or tiles arranged in a grid.
- **Z-index:** a number that decides which of two overlapping objects is drawn in front.
- **Viewport:** the rectangle of the world the player currently sees on the screen.
- **Camera2D:** a node that decides which part of the 2D world the viewport shows.
- **Canvas item:** any node that draws something in 2D (sprites, labels, shapes, controls).

6.1 Where is everything? The 2D coordinate system

Open a graph in a maths book and the picture is probably this: a point called the origin, an **x** axis going to the right, and a **y** axis going **up**. Games on a screen use a slightly different arrangement, borrowed from how screens are built (they draw rows from the top down).

In Godot 2D:

- The origin (0, 0) is the **top-left corner** of the screen in a brand-new scene, or wherever you decide to put it.
- **x** grows to the **right**.
- **y** grows **downward**.

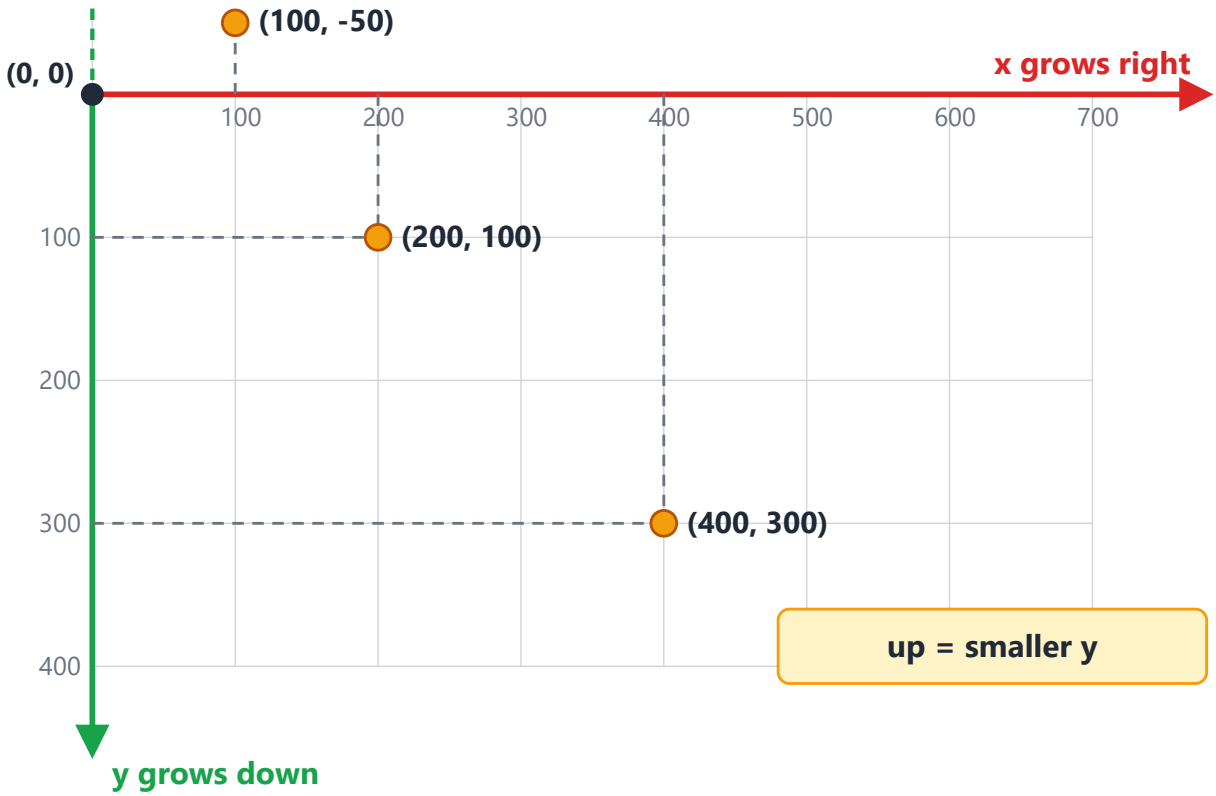


Figure 6.1: In 2D, x increases to the right and y increases downward. The point (200, 100) is 200 pixels right of and 100 pixels below the origin. Negative y means "above".

This catches almost every beginner once: to move something **up**, you make its y **smaller**. A point at (100, -50) is 50 pixels *above* the origin line.

The unit is the **pixel**. A default Godot window is 1152 pixels wide and 648 pixels tall. So the bottom-right corner of the default window is at about (1152, 648) and the centre is at (576, 324).

`Vector2`: one value for two numbers

Positions need two numbers at once. Godot bundles them into a type called **Vector2**:

```
var spot: Vector2 = Vector2(200, 100)
print(spot.x)    # 200
print(spot.y)    # 100
spot.x = 250     # change one part
print(spot)      # (250, 100)
```

A `Vector2` can mean a **position** ("where is it?"), a **size** ("how big?") or a **direction and distance** ("which way and how far?"). Same type, different meaning.

You can do arithmetic with vectors:

```
var a := Vector2(10, 20)
var b := Vector2(5, 5)

print(a + b)      # (15, 25)
print(a - b)      # (5, 15)
print(a * 2)      # (20, 40): multiply both parts
print(a / 2)      # (5, 10)
```

And ask useful questions:

```gdscript
var start := Vector2(0, 0)
var goal := Vector2(300, 400)

print(start.distance_to(goal))   # 500.0, straight-line distance
print(goal.length())             # 500.0, length measured from (0, 0)
print(goal.normalized())         # (0.6, 0.8): same direction, length exactly 1
print(goal.angle())              # the angle of the direction, in radians
```

A vector whose length is exactly 1 is called a **unit vector**; it carries only a direction. Multiplying a unit vector by a speed gives velocity: `direction * speed`. You will do this constantly in Chapter 7.

Godot also gives names to the four main directions:

Constant	Value	Meaning
`Vector2.RIGHT`	(1, 0)	right
`Vector2.LEFT`	(-1, 0)	left
`Vector2.DOWN`	(0, 1)	down (y grows downward)
`Vector2.UP`	(0, -1)	up
`Vector2.ZERO`	(0, 0)	no movement, the origin

> **Try it:** In a script's `_ready()`, print `Vector2(3, 4).length()`. The answer is `5.0`: a 3-4-5 triangle. Then print `Vector2(3, 4).normalized()` and check that the result's length is 1.

6.2 `Node2D`: giving a node a place in space

Everything that appears in a 2D game is some kind of **canvas item**, and most of those are `Node2D` or inherit from it. `Node2D` is the simplest 2D node: it draws nothing itself, but it has a **transform**: a position, a rotation, and a scale. Sprites, cameras, characters, particles and tile maps all share these.

Property	Type	Meaning
`position`	`Vector2`	where the node is, measured from its parent
`rotation`	`float`	angle in **radians** (clockwise on screen)
`rotation_degrees`	`float`	the same, but in degrees
`scale`	`Vector2`	size multiplier: (2, 2) is twice as big, (-1, 1) is mirrored
`skew`	`float`	slant in radians (rarely used)
`visible`	`bool`	whether it is drawn
`z_index`	`int`	draw order (section 6.3)
`modulate`	`Color`	a tint multiplied into everything drawn by this node and its children

```gdscript
extends Node2D
# Attached to any Node2D.

func _ready() -> void:
    position = Vector2(300, 200)
    rotation_degrees = 30.0
```

```
    scale = Vector2(1.5, 1.5)
    modulate = Color(1, 1, 1, 0.5)   # half transparent
```

Radians and degrees

An angle can be measured in degrees (a full turn is 360) or in **radians** (a full turn is about 6.283, which equals 2 times pi). Godot's code uses radians; the Inspector shows degrees. Convert them with helpers:

```
print(deg_to_rad(180.0))   # 3.14159... (that is PI)
print(rad_to_deg(PI))      # 180.0
print(TAU)                 # 6.28318... one full turn
```

Constants `PI` (half turn), `TAU` (full turn) and `PI / 2` (quarter turn) are available everywhere. In Godot, a positive rotation turns **clockwise** on screen, because y points downward.

The origin of a node

Every `Node2D` has its own little coordinate system, and its own (0, 0) is the point where the node sits, called its **origin**. When you rotate a node, it turns around that point. When you scale it, it grows from that point. The editor shows a small blue and red pair of arrows at it, pointing along the node's own x axis (red) and y axis (green).

Parents carry their children along

The most important rule in 2D scene building:

> A child's `position` is measured **from its parent**, not from the world.

If you move, rotate or scale the parent, **every child moves with it**, turning and scaling too. This is how a character is built from parts: the head, the arms, and a sword all hang beneath the body node. Move the body and everything follows.

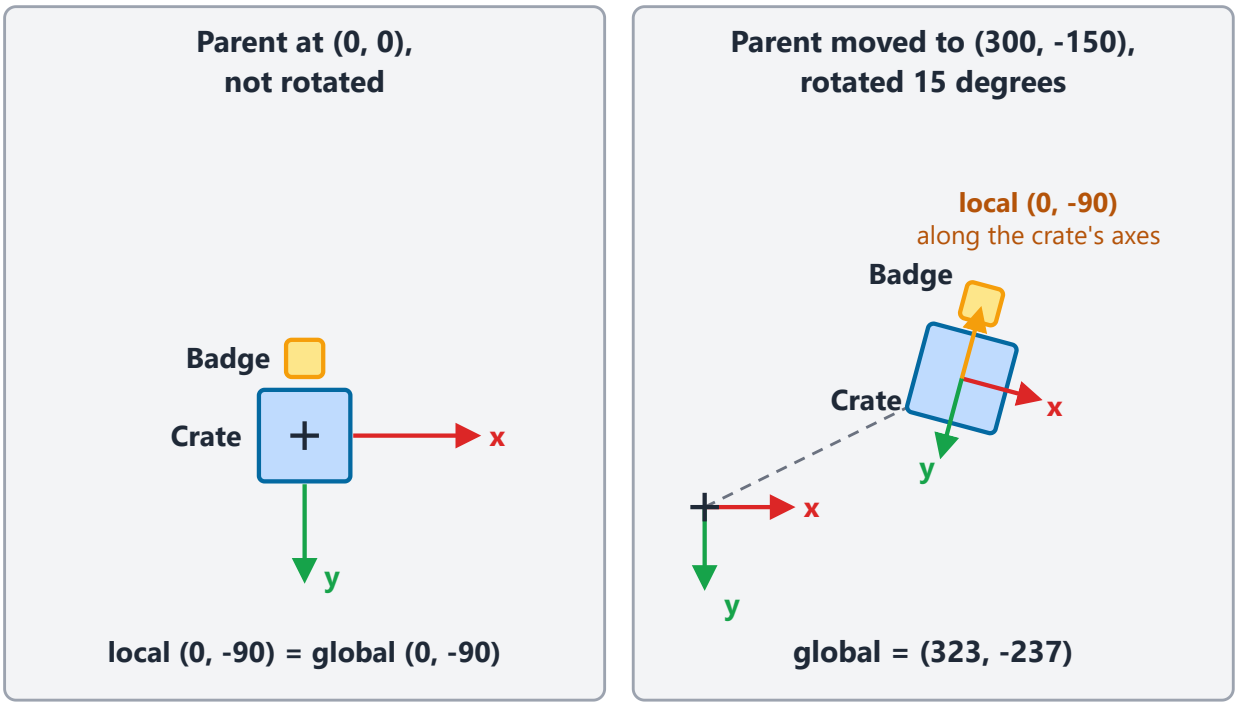


Figure 6.2: The Badge node has a local position of (0, -90): 90 pixels above its parent, the Crate. Because the Crate is rotated and moved, the Badge's global position is a different number.

That gives each node two ways to report its position:

Property	Space	Meaning
`position`	local	measured from the parent's origin, in the parent's turned and scaled axes
`global_position`	global	measured from the world origin; the real place on the canvas

If a node has no parent that is a `Node2D`, or the parent sits at the origin without rotation or scale, the two are equal. As soon as a parent moves, they differ.

Use `position` when you think "relative to my parent" ("the sword is 20 pixels to the right of the hand"). Use `global_position` when you think "where in the world?" ("is the enemy near the exit at (900, 300)?"). Rule of thumb: **read** `global_position` when comparing two things in different branches of the tree, **set** `position` for building within one branch.

Convert between spaces with `to_global()` and `to_local()`:

```
# In a script on a Node2D:
var world_point: Vector2 = to_global(Vector2(0, -90))   # a spot 90 px above me, in world coordinates
var local_point: Vector2 = to_local(Vector2(500, 500))  # world spot, expressed in my own space
```

Facing a point

`look_at(point)` turns a `Node2D` so its x axis (its "right") points at a **global** position:

```
look_at(Vector2(600, 100))
```

A sprite drawn facing right will then face the target. Sprites drawn facing up or down need an extra quarter turn, or the artwork redrawn.

6.3 Sprites and textures

A **texture** is an image in memory. A `Sprite2D` is a node that draws a texture at its position. This pairing is the heart of 2D games: each character, coin and tree is usually a Sprite2D (or a related node) with a texture.

Getting a picture on the screen

1. Add a `Sprite2D` node.
2. In the Inspector, find **Texture**, open the drop-down and choose **Load**, or drag an image from the FileSystem dock onto the slot.

Every Godot project includes `icon.svg`, so you always have an image to practise with. To use your own images, drag a PNG file from your computer into the FileSystem dock; Godot **imports** it, which means it makes a copy in a form the engine loves. You will see the import settings in the **Import** dock next to the Scene dock.

> **Note:** The book's projects use only shapes and the bundled `icon.svg`, so you do not need anything else. If you want real art, free and generous sources are Kenney.nl, itch.io (filter by "free" assets) and OpenGameArt.org. Always check each pack's license.

Sprite2D properties you will use

Property	What it does
`texture`	which image to draw
`centered`	on by default: the picture's centre sits on the node's position. Off: the top-left corner does
`offset`	shifts the picture without moving the node
`flip_h`, `flip_v`	mirror the image horizontally or vertically
`modulate`	tint the sprite (white means unchanged)
`region_enabled`, `region_rect`	draw only a rectangle of the texture
`hframes`, `vframes`, `frame`	cut the texture into a grid and show one cell

A common beginner question: "My sprite is blurry (or crisp) and I want the other." For pixel-art games you want hard edges. Go to **Project > Project Settings > Rendering > Textures** and set **Default Texture Filter** to **Nearest**. (In the General tab you may need to turn on **Advanced Settings** to see some options.)

Sprite sheets: many pictures in one

Game artists usually pack many small images into one big one, a **sprite sheet**, because it is efficient. You can then draw just one piece of it. There are two ways.

1. A region. Use this when pieces are different sizes or placed freely. Tick `region_enabled` and give `region_rect` a rectangle `Rect2(x, y, width, height)`:

```
extends Sprite2D
# Attached to a Sprite2D whose texture is icon.svg (128 x 128 pixels).

func _ready() -> void:
    region_enabled = true
    region_rect = Rect2(0, 0, 64, 64)   # only the top-left quarter
```

2. A frame grid. Use this when the sheet is an even grid. Set `hframes` (columns) and `vframes` (rows), then choose a `frame` number:

```
extends Sprite2D
# Attached to a Sprite2D whose texture is a sheet divided into 4 columns and 2 rows.

func _ready() -> void:
    hframes = 4
    vframes = 2
    frame = 5     # frames count left to right, then top to bottom: 0..7
```

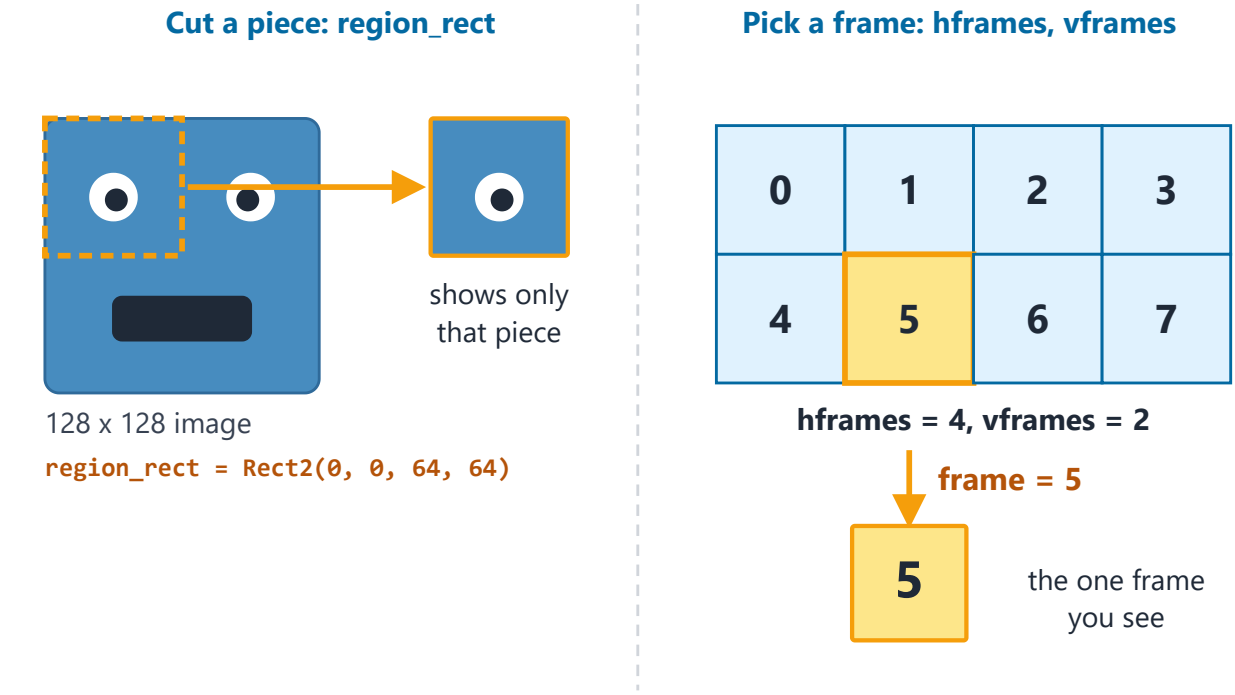


Figure 6.3: Left: `region_rect` cuts any rectangle out of the texture. Right: `hframes` = 4 and `vframes` = 2 number the cells 0 to 7; `frame` picks one.

Changing `frame` over time is exactly what an animation is. Chapter 10 does that for you with `AnimatedSprite2D`.

Art without art: `PlaceholderTexture2D`

While you are designing a level you may not have final art. A `PlaceholderTexture2D` is a plain stand-in rectangle with a size you choose. Assign it to a Sprite2D's **Texture** (choose **New PlaceholderTexture2D** from the drop-down), set its **Size** to (32, 32), and you have a visible box that you can tint with `modulate`. Use it for markers and prototypes.

Which sprite is in front?

When two sprites overlap, one has to cover the other. Godot uses these rules, in this order:

1. A higher `z_index` is drawn on top of a lower one. The default is 0. `z_index` is *relative* to the parent by default, so a child with `z_index = 1` is above its parent.
2. If the `z_index` values tie, the node **lower in the Scene dock** (later in the tree) is drawn in front.

So the quickest tool is simply **order in the tree**: drag the node you want in front below the other. Use `z_index` when order in the tree cannot give the answer, for example a flying bird that must be in front of everything.

```
$Bird.z_index = 10
```

For top-down games where characters should hide behind objects that are "nearer" the bottom of the screen, a parent can turn on `y_sort_enabled` so its children are drawn in order of their y position. You will meet it in Chapter 12.

6.4 The viewport and the window

The **viewport** is the rectangular view onto your 2D world that fills the game window. By default the window is 1152 by 648, but a player might make it bigger or full screen. What should happen then?

Godot's answer is in **Project > Project Settings > Display > Window**:

- **Viewport Width / Height:** the design size of your game, such as 1152 by 648.
- **Stretch > Mode:** what to do when the real window differs. `disabled` leaves things at the same pixel size and shows more or less of the world. `canvas_items` scales everything to fill the window, which suits most 2D games. `viewport` renders at the small size and scales the picture up, which suits pixel art.
- **Stretch > Aspect:** `keep` adds black bars to preserve the proportions; `expand` reveals more of the world.

For now, set **Mode** to `canvas_items` and **Aspect** to `keep`. Your game will then look the same at any window size.

You can read the viewport size in code:

```
var size: Vector2 = get_viewport_rect().size
print(size)   # (1152, 648) by default
```

That is handy for placing something at the centre of the screen: `position = size / 2.0`.

6.5 The camera

A game world can be much larger than the screen. A `Camera2D` chooses the part of the world that you see. Think of it as a movie camera: whatever the camera points at appears in the window.

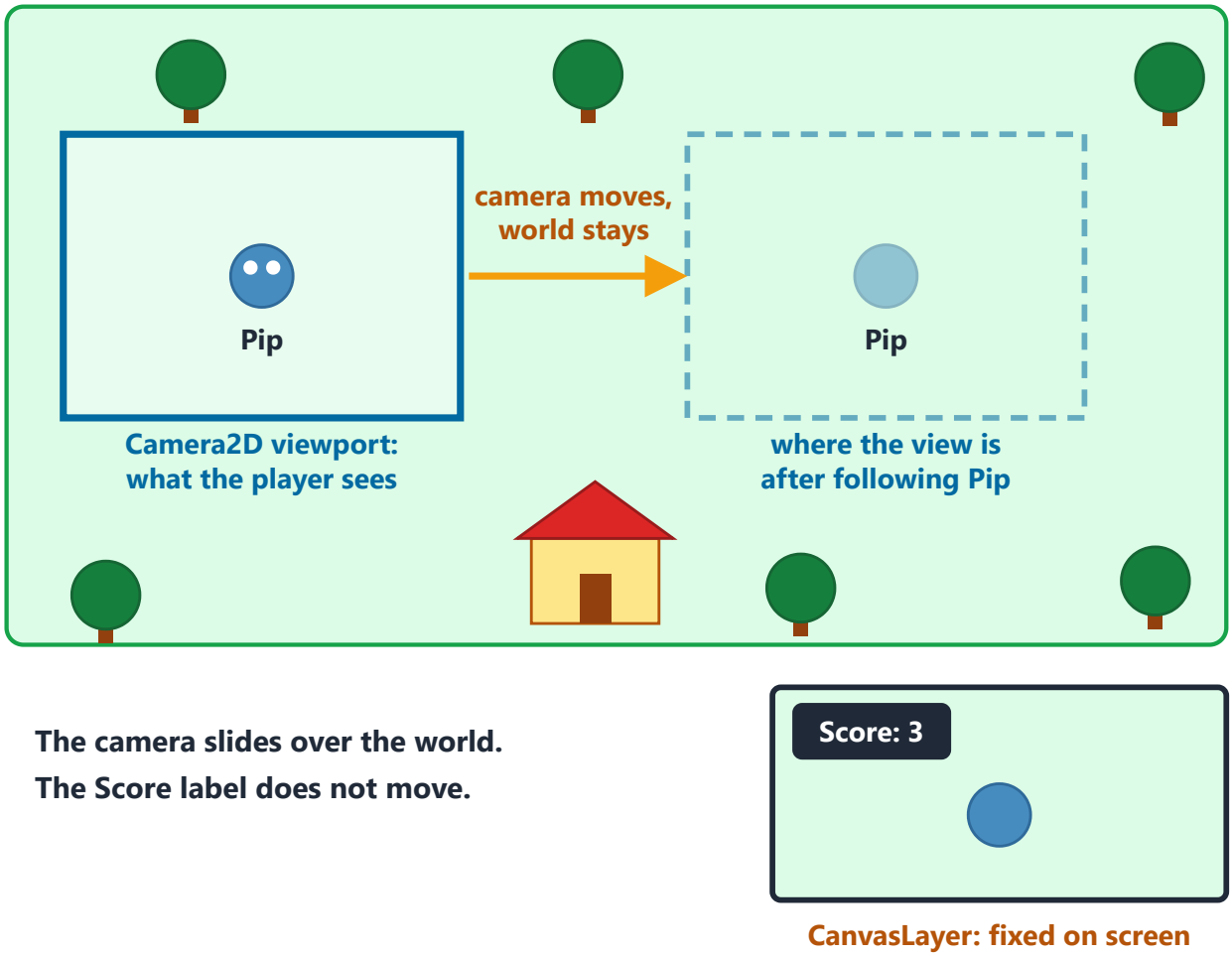


Figure 6.4: The viewport rectangle is what the player sees. When the camera moves, the rectangle slides across the world. Objects do not move; the view does.

Using a Camera2D

1. Add a `Camera2D` node. If it has no parent that moves, it just points at its own position.
2. In the Inspector, make sure **Enabled** is on. (If there are several cameras, the most recently activated enabled one is used; `make_current()` forces one.)

By default the centre of the screen is the camera's position. Move the camera and the view moves.

The classic setup for a character: make the camera a **child of the character**. Because children travel with their parent, the camera follows automatically.

```
Hero (Node2D)
├─ Sprite2D
└─ Camera2D
```

Handy camera properties

Property	What it does
`zoom`	(2, 2) shows things twice as large (view covers half the area); (0.5, 0.5) zooms out
`position_smoothing_enabled`	the camera glides to its target rather than snapping

Property	What it does
`position_smoothing_speed`	how quickly it catches up
`offset`	shift the view without moving the camera
`limit_left`, `limit_top`, `limit_right`, `limit_bottom`	stop the camera from showing space outside the level
`enabled`	turn this camera on or off

Zoom is a `Vector2` because it could scale x and y differently, but you will almost always use equal numbers:

```
extends Camera2D
# Attached to a Camera2D.

func _ready() -> void:
    zoom = Vector2(1.5, 1.5)
    position_smoothing_enabled = true
```

Interface that does not move: `CanvasLayer`

If the camera moves, everything in the world moves relative to the screen, including a score label that is part of the world. You want labels, health bars and buttons to **stay in place**. Put them under a `CanvasLayer`. A CanvasLayer is drawn on its own layer, independent of the camera. Its `layer` number says whether it is drawn behind (negative) or in front of (positive, default 1) the world.

```
Main (Node2D)
├─ World stuff...
├─ Hero (Node2D)
│  └─ Camera2D
└─ Hud (CanvasLayer)
   └─ ScoreLabel (Label)
```

Mouse position in the world

When the camera zooms or moves, the mouse position in the window no longer equals a position in the world. Any `CanvasItem` (a Node2D, for example) can answer the question properly:

```
var mouse_in_world: Vector2 = get_global_mouse_position()
```

Use that rather than guessing from the window. In the project you will print it live.

6.6 Drawing with code

You do not always need an image. Any `Node2D` can draw its own shapes by defining a function named `_draw()`. Godot calls it when the node needs to be painted, and inside it you may call drawing functions. Everything is drawn in the node's own local coordinates, with (0, 0) at the node's origin.

```
extends Node2D
# Attached to a Node2D. Draws a red square, a blue circle and a green line.

func _draw() -> void:
    draw_rect(Rect2(-50, -50, 100, 100), Color.RED)           # x, y, width, height
    draw_circle(Vector2(100, 0), 30.0, Color.BLUE)            # centre, radius, colour
    draw_line(Vector2(-100, 80), Vector2(100, 80), Color.GREEN, 4.0)   # from, to, colour, width
```

Draw functions you will use most:

Function	Draws
`draw_line(from, to, color, width)`	a straight line
`draw_rect(rect, color, filled, width)`	a rectangle (filled by default; pass `false` for just an outline)
`draw_circle(center, radius, color)`	a filled circle
`draw_arc(center, radius, start_angle, end_angle, point_count, color, width)`	part of a circle outline
`draw_colored_polygon(points, color)`	a filled shape from a list of corner points
`draw_string(font, position, text, alignment, width, font_size)`	text

`_draw()` is called **once** and then cached, which is efficient. If what you draw must change, ask Godot to call it again with `queue_redraw()`:

```
extends Node2D

var radius: float = 20.0

func _process(delta: float) -> void:
    radius += 10.0 * delta
    queue_redraw()      # ask Godot to call _draw() again this frame

func _draw() -> void:
    draw_circle(Vector2.ZERO, radius, Color.ORANGE)
```

> **Warning:** You can only call `draw_*` functions from inside `_draw()`. Calling them from `_ready()` or `_process()` does nothing useful, or reports an error.

Shapes as nodes: `Polygon2D` and `Line2D`

Two nodes draw shapes without any script:

- **`Polygon2D`** fills a closed outline. Its `polygon` property is a `PackedVector2Array`, a list of corner points. It also has `color`.
- **`Line2D`** draws a line through a list of `points`, with a `width` and `default_color`.

You can edit these visually in the editor: select the node and use the toolbar buttons to add and drag points. Or set them in code:

```
extends Node2D
# Attached to a Node2D with no required children; builds a triangle flag at runtime.

func _ready() -> void:
    var flag := Polygon2D.new()                  # create a node from code
    flag.polygon = PackedVector2Array([Vector2(0, 0), Vector2(60, 20), Vector2(0, 40)])
    flag.color = Color(0.9, 0.3, 0.3)
    add_child(flag)                              # put it into the tree
```

`Polygon2D.new()` creates a node in code, and `add_child()` places it in the scene. Both will be important in Chapter 9, where coins appear as the game runs.

Draw Pip with code

Pip, the hero of this book, is a circle with eyes. Here is the whole of Pip as a script on a `Node2D`:

```
extends Node2D
# Attached to a Node2D named "Pip".

func _draw() -> void:
    draw_circle(Vector2.ZERO, 28.0, Color(1.0, 0.84, 0.29))                # yellow body
    draw_arc(Vector2.ZERO, 28.0, 0.0, TAU, 32, Color(0.2, 0.15, 0.05), 3.0)   # dark outline
    draw_circle(Vector2(-9, -6), 4.0, Color.BLACK)                         # left eye
    draw_circle(Vector2(9, -6), 4.0, Color.BLACK)                          # right eye
    draw_arc(Vector2(0, 4), 12.0, 0.2, PI - 0.2, 12, Color.BLACK, 2.5)     # smile
```

The smile is the bottom half of a small circle: angles from just over 0 to just under PI (a half turn). Because y grows downward, angles between 0 and PI sweep through the lower half, which is why the smile curves down and not up.

6.7 Motion on a circle (a little maths, with pictures)

To make Pip walk in a circle, you need a way to turn an angle into a point. The functions `cos` and `sin` do it: for an angle `a`, the point `Vector2(cos(a), sin(a))` lies on a circle of radius 1 around the origin. Multiply by a radius `r` to make a bigger circle:

```
extends Node2D
# Attached to a Node2D. Orbits the origin.

var angle: float = 0.0

func _process(delta: float) -> void:
    angle += 0.8 * delta                                  # turn 0.8 radians per second
    position = Vector2(cos(angle), sin(angle)) * 220.0  # circle of radius 220
```

You do not need to understand why `cos` and `sin` work yet; see Appendix G for a picture-based explanation. The recipe "advance an angle with delta, then convert it with `cos` and `sin`" will serve you for orbiting moons, bobbing pickups and swinging pendulums.

Project 6: Pip's Yard

Goal: build a scene that exercises every idea in this chapter: a drawn coordinate grid, sprites with regions, parent and child transforms, a drawn character walking in a circle, a camera that follows and zooms, and a live coordinate display.

Time: about 60 minutes.

Finished copy: `projects/ch06-2d-space-and-sprites/`.

```
ch06-2d-space-and-sprites/
├─ project.godot
├─ icon.svg
├─ main.tscn
├─ yard.gd          (grid and axes)
├─ pip.gd           (Pip's drawing and circular walk)
├─ camera_rig.gd    (zoom control)
└─ hud.gd           (live coordinates)
```

Steps

Part A: set up the scene

1. Create a new project `Ch06 2D Space and Sprites`.
2. Make the root a **Node2D**, rename it **Yard**, and save as `main.tscn`.
3. Add a **Sprite2D** child named **Crate**. Set its **Texture** to `icon.svg`, **Position** to (300, -150) and **Rotation** to 15 degrees.
4. Add a **Sprite2D** child of **Crate** named **Badge**. In its Texture slot choose **New PlaceholderTexture2D**, click the new resource and set **Size** to (32, 32). Set **Position** to (0, -90) and set **Modulate** (under CanvasItem > Visibility) to a light red.
5. Add another **Sprite2D** child of **Yard** named **Tile**. Texture `icon.svg`, **Position** (-300, -150). Under **Region**, tick **Enabled** and set **Rect** to position (0, 0), size (64, 64). Only the top-left quarter of the icon shows.
6. Add a **Polygon2D** child of **Yard** named **Flag** at position (-300, 150). In the Inspector set its **Color** to red-ish and its **Polygon** array to three points: (0, 0), (60, 20), (0, 40). (You can click the array in the Inspector and add elements, or use the toolbar point editor.)
7. Add a **Line2D** child of **Flag** named **Pole**. Set **Points** to (0, 0) and (0, 90), **Width** to 6, and a brown **Default Color**.
8. Add a **Node2D** child of **Yard** named **Pip**.
9. Add a **Camera2D** as a child of **Pip**. In the Inspector turn on **Position Smoothing > Enabled**.
10. Add a **CanvasLayer** child of **Yard** named **Hud**, and under it a **Label** named **InfoLabel** with **Position** about (16, 12) and **Size** about (700, 128).

The finished tree:

```
Yard (Node2D)                <- yard.gd
├─ Crate (Sprite2D)
│  └─ Badge (Sprite2D)
├─ Tile (Sprite2D)
├─ Flag (Polygon2D)
│  └─ Pole (Line2D)
├─ Pip (Node2D)              <- pip.gd
│  └─ Camera2D               <- camera_rig.gd
└─ Hud (CanvasLayer)         <- hud.gd
   └─ InfoLabel (Label)
```

Part B: draw the grid

11. Select **Yard**, attach a script `yard.gd`, and write:

```gdscript
extends Node2D
# Attached to the root node "Yard" (type Node2D).

const GRID_STEP: int = 100
const GRID_EXTENT: int = 1000


func _draw() -> void:
    var font: Font = ThemeDB.fallback_font
    var grid_color := Color(1, 1, 1, 0.12)

    # Grid lines every 100 pixels.
    for i in range(-GRID_EXTENT, GRID_EXTENT + 1, GRID_STEP):
        draw_line(Vector2(i, -GRID_EXTENT), Vector2(i, GRID_EXTENT), grid_color, 1.0)
        draw_line(Vector2(-GRID_EXTENT, i), Vector2(GRID_EXTENT, i), grid_color, 1.0)

    # The axes: x is red, y is green.
```

```
    draw_line(Vector2(-GRID_EXTENT, 0), Vector2(GRID_EXTENT, 0), Color(1, 0.3, 0.3), 3.0)
    draw_line(Vector2(0, -GRID_EXTENT), Vector2(0, GRID_EXTENT), Color(0.3, 1, 0.3), 3.0)

    # Number labels along the axes.
    for i in range(-GRID_EXTENT, GRID_EXTENT + 1, GRID_STEP):
        if i == 0:
            continue
        draw_string(font, Vector2(i + 4, 18), str(i), HORIZONTAL_ALIGNMENT_LEFT, -1, 14)
        draw_string(font, Vector2(6, i - 4), str(i), HORIZONTAL_ALIGNMENT_LEFT, -1, 14)

    # A dot and label at the origin.
    draw_circle(Vector2.ZERO, 8.0, Color.WHITE)
    draw_string(font, Vector2(12, -12), "(0, 0)", HORIZONTAL_ALIGNMENT_LEFT, -1, 16)
```

`ThemeDB.fallback_font` is the font Godot ships with, so you do not need to import one. In `draw_string`, the `-1` means "no width limit".

Because **Yard** is the first node, its drawing lies *behind* its children. That keeps the grid under the sprites.

12. Run the project (F5). Because the Camera2D is a child of Pip and Pip sits at the origin for now, the view is centred on (0, 0). You should see the red and green axes crossing in the middle of the window, the grid around them, and the crate, tile and flag. Pip does not move or show yet.

Part C: Pip

13. Select **Pip**, attach `pip.gd`, and write:

```
extends Node2D
# Attached to the node "Pip" (type Node2D). Pip is drawn with code and walks in a circle.

@export var radius: float = 220.0
@export var speed: float = 0.8   # radians per second

var angle: float = 0.0


func _process(delta: float) -> void:
    angle += speed * delta
    position = Vector2(cos(angle), sin(angle)) * radius


func _draw() -> void:
    draw_circle(Vector2.ZERO, 28.0, Color(1.0, 0.84, 0.29))
    draw_arc(Vector2.ZERO, 28.0, 0.0, TAU, 32, Color(0.2, 0.15, 0.05), 3.0)
    draw_circle(Vector2(-9, -6), 4.0, Color.BLACK)
    draw_circle(Vector2(9, -6), 4.0, Color.BLACK)
    draw_arc(Vector2(0, 4), 12.0, 0.2, PI - 0.2, 12, Color.BLACK, 2.5)
```

14. Run again. Pip circles the origin. The camera, being Pip's child, follows with a smooth lag, so the grid scrolls past.

Part D: the camera zoom

15. Select **Camera2D** under Pip, attach `camera_rig.gd`:

```
extends Camera2D
# Attached to the node "Camera2D" (type Camera2D), a child of Pip.
# Zoom with the Up and Down arrow keys (built-in actions "ui_up" and "ui_down").

@export var zoom_speed: float = 1.5
@export var min_zoom: float = 0.4
@export var max_zoom: float = 3.0
```

```
func _process(delta: float) -> void:
    var amount: float = zoom.x
    if Input.is_action_pressed("ui_up"):
        amount *= 1.0 + zoom_speed * delta
    if Input.is_action_pressed("ui_down"):
        amount /= 1.0 + zoom_speed * delta
    amount = clampf(amount, min_zoom, max_zoom)
    zoom = Vector2(amount, amount)
```

This is a sneak peek at input handling (the full story is in Chapter 7): `Input.is_action_pressed("ui_up")` asks whether the Up arrow key is held now. `clampf(value, low, high)` forces a number to stay between two limits.

Part E: the live display

16. Select **Hud**, attach `hud.gd`:

```
extends CanvasLayer
# Attached to the node "Hud" (type CanvasLayer). Shows live coordinates.

@onready var info_label: Label = $InfoLabel
@onready var pip: Node2D = $"../Pip"
@onready var badge: Sprite2D = $"../Crate/Badge"
@onready var camera: Camera2D = $"../Pip/Camera2D"


func _process(_delta: float) -> void:
    # A CanvasLayer is not a Node2D, so ask Pip for the mouse position in the world.
    var mouse_world: Vector2 = pip.get_global_mouse_position()
    info_label.text = "Pip position: %s\nBadge local: %s   Badge global: %s\nMouse in the world:
%s\nCamera zoom: %.2f   (Up / Down arrows to zoom)" % [
        pip.position.round(),
        badge.position,
        badge.global_position.round(),
        mouse_world.round(),
        camera.zoom.x,
    ]
```

The `%s` slot prints any value as text, including a `Vector2`; `%.2f` prints a decimal with two digits.

17. Run (F5). Test:

    - Pip walks a circle, with the camera trailing.
    - The info label stays fixed in the corner while the world scrolls (because it is in a CanvasLayer).
    - "Badge local" is always (0, -90), while "Badge global" is a different pair of numbers because the Crate is moved and tilted. Select the **Crate** in the editor, change its position or rotation, run again, and watch "Badge global" change while "Badge local" stays (0, -90).
    - Move the mouse over the world. "Mouse in the world" follows the cursor even when the camera moves.
    - Press the Up and Down arrows to zoom.

Expected result

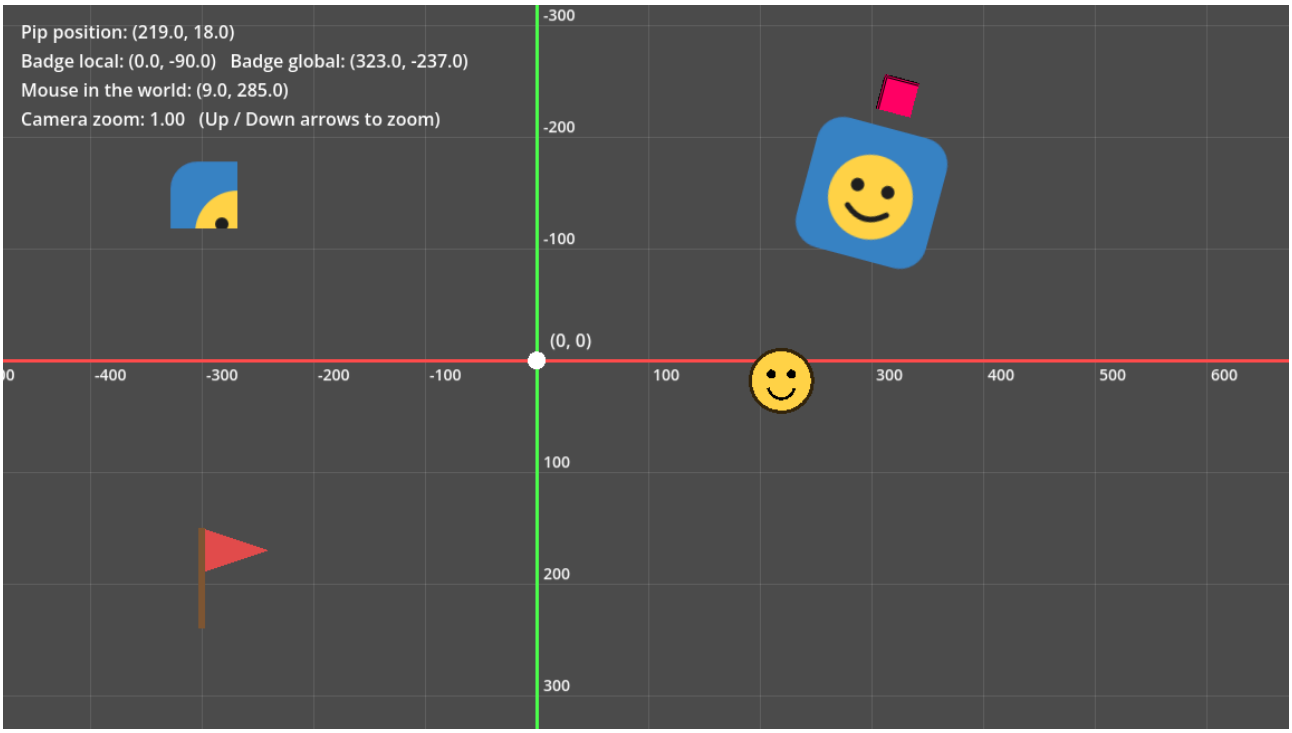


Figure 6.5: Pip's Yard in action. The grid shows the world's coordinates, the HUD text shows live values, and Pip circles the origin with the camera gliding behind.

Chapter summary

- Godot 2D uses pixels, with the origin at (0, 0), x growing right and y growing **down**. Up is negative y.
- `Vector2` holds two numbers and supports arithmetic, `length()`, `normalized()` and `distance_to()`. Unit vectors carry only direction.
- `Node2D` has `position`, `rotation` (radians), `scale`, `z_index` and `modulate`. Children are positioned relative to their parent. `global_position` is the position in the world.
- `Sprite2D` draws a texture. Use `region_rect` or `hframes` / `vframes` / `frame` to show part of a sprite sheet, and `PlaceholderTexture2D` for stand-ins.
- Draw order follows `z_index` first, and then position in the tree: lower in the Scene dock draws on top.
- `Camera2D` chooses what part of the world is shown. Make it a child of a character to follow it, use `zoom` and smoothing, and put interface elements in a `CanvasLayer` so they stay put.
- `_draw()` with `draw_circle`, `draw_line`, `draw_rect`, `draw_arc` and others draws shapes with no art files. `queue_redraw()` repaints when the picture changes.
- `Polygon2D` and `Line2D` provide editable shapes as nodes.

Review questions

1. In Godot's 2D system, which direction does the y coordinate increase?
2. A node is at position (100, 200). To move it 50 pixels *up*, what should its new position be?
3. What is the length of `Vector2(6, 8)`?
4. A child has `position = Vector2(0, -90)` and its parent is rotated by 90 degrees. Is the child's `global_position` also (parent position plus (0, -90))? Explain.
5. What is the difference between `position` and `global_position`?
6. Two sprites overlap, and both have `z_index` 0. Which one is drawn in front?

7. You have a sprite sheet laid out as a regular grid with 6 columns and 3 rows. Which two properties do you set so you can choose a cell, and which property then picks it?
8. A health label sits in the scene next to the player, and when the camera follows the player the label slides out of view. What node should you put the label under to keep it fixed on screen?
9. True or false: you can call `draw_circle()` from inside `_ready()` to paint a circle on a `Node2D`.
10. What does `queue_redraw()` do, and when do you need it?

Exercises

1. **Practice (easy):** In a new scene, place three `Sprite2D` nodes using `icon.svg` at (200, 200), (400, 200) and (600, 200). Give them scales of (0.5, 0.5), (1, 1) and (1.5, 1.5). Run and confirm the sizes.
2. **Practice (easy):** In Pip's Yard, change `pip.gd` so Pip walks the other way around the circle (hint: a negative speed). Then export a `color` variable for his body and change it in the Inspector.
3. **Apply (medium):** Add a second child to the Crate: a `Sprite2D` named `Marker` whose position is (60, 0), using a `PlaceholderTexture2D` of size (20, 20). Without writing any code, rotate the Crate in the editor and observe both children. Then print the `global_position` of both in `hud.gd`.
4. **Apply (medium):** Make the tile show frame number 2 of a sheet. Use `icon.svg` with `hframes = 2` and `vframes = 2`, set `frame` to 2 in a script's `_ready()`, and describe which quarter of the icon appears and why.
5. **Challenge (hard):** Draw a "compass rose" in `yard.gd` using `_draw()`: four short arrows from the origin pointing along `Vector2.UP`, `DOWN`, `LEFT` and `RIGHT` in different colours, with a letter for each. Hint: an arrowhead is a small triangle you can draw with `draw_colored_polygon`.

Common mistakes

- **Symptom:** an object meant to go "up" goes down. **Cause:** y grows downward. **Fix:** subtract from y to move up, or use `Vector2.UP`, which is `(0, -1)`.
- **Symptom:** a child sprite ends up in a strange place after you move or rotate its parent. **Cause:** child positions are local to the parent. **Fix:** set `position` in the parent's space, or use `global_position` if you really need a world spot.
- **Symptom:** the sprite turns the wrong amount; 90 makes it spin around nearly 15 times. **Cause:** `rotation` is in radians. **Fix:** use `rotation_degrees`, or `deg_to_rad(90)`.
- **Symptom:** nothing shows up from your `draw_*` calls. **Cause:** they were called outside `_draw()`, or `_draw()` ran before your data changed. **Fix:** put them in `_draw()` and call `queue_redraw()` when the data changes.
- **Symptom:** the score label scrolls away with the world. **Cause:** it is a child of a world node, not of a `CanvasLayer`. **Fix:** move the label under a `CanvasLayer`.
- **Symptom:** the pixel art looks blurry. **Cause:** the default texture filter smooths pixels. **Fix:** Project Settings > Rendering > Textures > Default Texture Filter = Nearest.

Further reading

- Introduction to 2D: https://docs.godotengine.org/en/stable/tutorials/2d/index.html
- Custom drawing in 2D: https://docs.godotengine.org/en/stable/tutorials/2d/custom_drawing_in_2d.html
- Using Transforms (local and global space): https://docs.godotengine.org/en/stable/tutorials/math/matrices_and_transforms.html
- Vector math: https://docs.godotengine.org/en/stable/tutorials/math/vector_math.html

- Class reference for `Camera2D`: https://docs.godotengine.org/en/stable/classes/class_camera2d.html
- Class reference for `Sprite2D`: https://docs.godotengine.org/en/stable/classes/class_sprite2d.html
- Multiple resolutions (stretch settings): https://docs.godotengine.org/en/stable/tutorials/rendering/multiple_resolutions.html

Chapter 7: Input and Movement

Learning objectives

By the end of this chapter you will be able to:

- Explain the difference between *polling* input (asking "is the key down right now?") and *reacting to events* (being told "a key was just pressed").
- Create, name, and edit actions in the Input Map so one action works with the keyboard, a gamepad, and the mouse.
- Read player input in a script with `Input.is_action_pressed()`, `Input.is_action_just_pressed()`, `Input.get_axis()`, and `Input.get_vector()`.
- Handle input events with `_input()` and `_unhandled_input()`, and tell the common event types apart.
- Use `delta` so that movement runs at the same speed on every computer, and explain why this works.
- Build a character that moves in eight directions, dashes, stays inside the screen, and walks to a mouse click.

Key terms

- **Input device**: Any hardware the player uses to talk to the game: keyboard, mouse, gamepad, touch screen.
- **Input event** (`InputEvent`): A small object that describes one thing that happened on a device, such as "the W key was pressed" or "the mouse moved".
- **Action**: A name you invent for something the player can *do* ("move_left", "jump", "dash"). Your code asks about actions instead of about specific keys.
- **Input Map**: The list in Project Settings that connects each action to one or more physical inputs (keys, buttons, stick directions).
- **Polling**: Checking the state of the input yourself, on every frame, by calling a function such as `Input.is_action_pressed()`.
- **Event-driven input**: Letting the engine call your function (`_input()` or `_unhandled_input()`) only when something happens.
- **Deadzone**: A small region near the centre of a thumb stick where tiny movements are ignored, because real sticks never rest at exactly zero.
- **Frame**: One picture drawn on the screen. A game draws many frames per second.
- **Delta** (`delta`): The number of seconds that passed since the previous frame. At 60 frames per second it is about 0.0167.
- **Velocity**: How fast something moves and in which direction. In this chapter we measure it in pixels per second.
- **Normalize**: To change a vector so that its length becomes exactly 1 while keeping its direction.

7.1 How a key press becomes a game action

Until now your games have run by themselves. They have shown things, but the player could not *do* anything. This chapter changes that. We will teach Pip, our small round hero, to listen.

Before we write any code, it helps to understand the path that a single key press takes, because every input question in this chapter is a question about some step on that path.

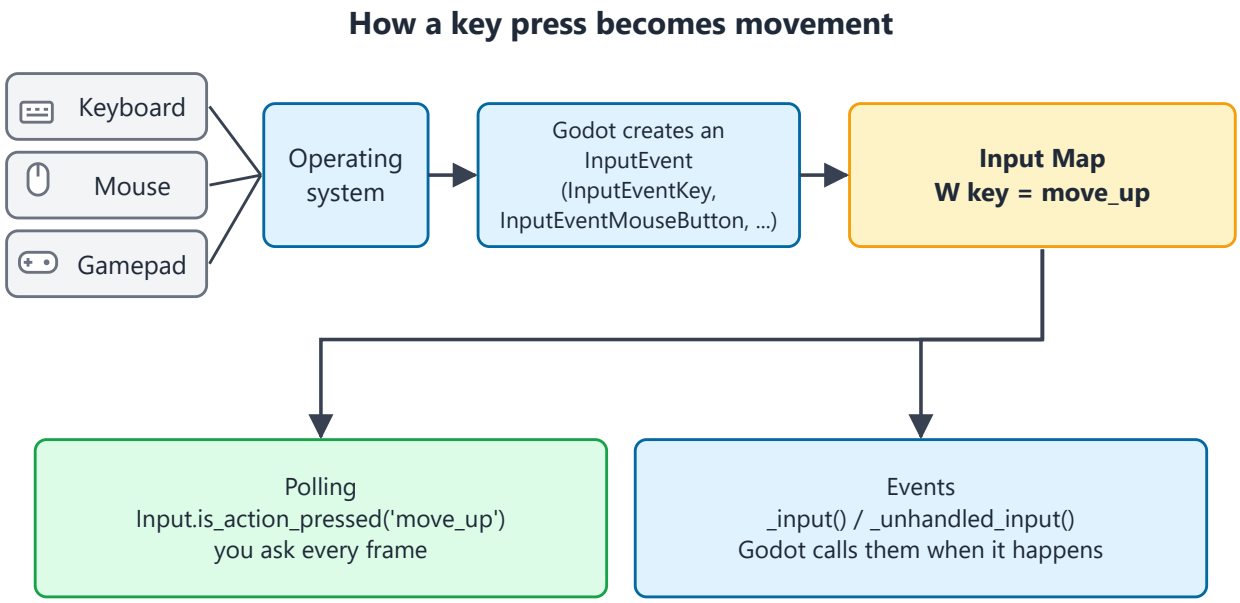


Figure 7.1: The path of an input. Hardware sends a signal to the operating system, Godot turns it into an `InputEvent`, the Input Map translates it into an action, and your script either asks about the action or is told about it.

Follow Figure 7.1 from left to right.

1. **The player presses the W key.** The keyboard sends an electrical signal to the computer.
2. **The operating system** (Windows, macOS, Linux) receives it and passes a message to the running program, which is your game.
3. **Godot creates an `InputEvent` object.** For a key, the object is of the more specific type `InputEventKey`. It records which key, whether it went down or up, and whether Shift or Ctrl were held.
4. **The Input Map checks its list.** If you have told it "the W key means `move_up`", the event is recognised as belonging to the action `move_up`.
5. **Your script finds out.** There are two ways, and we will use both in this chapter:
    - **Polling.** On each frame, your script *asks*: "Is `move_up` held down right now?"
    - **Events.** Godot *tells* your script: "An event just arrived. Do you want it?"

An everyday comparison may help. Imagine you are waiting for a friend to arrive at your door.

- **Polling** is walking to the door and looking through the peephole every few seconds. You always know the current state ("someone is there" or "no one is there"), but you spend effort checking even when nothing changes.
- **Event-driven input** is having a doorbell. You do other things, and the bell *interrupts* you at the exact moment something happens. You never see what happened between rings.

Neither is better. A character that moves while a button is *held* is naturally a polling job: "while the key is down, keep moving". A menu that opens when you press Escape *once* is naturally an event job: "when this happens, do that". Good game code uses both.

> **Note:** You might wonder why step 4 exists at all. Why not ask "is the W key down?" directly? You can. Godot allows `Input.is_key_pressed(KEY_W)`. But then your game works only with a keyboard, only with that key, and you must edit the code to change the controls. Actions solve all three problems, which is why the rest of this chapter is built on them.

7.2 The Input Map and actions

An **action** is a name that you choose for something the player can do. It describes the *intention*, not the device. "Move left" is an action. "Press A" is a way to trigger it. One action can have many triggers.

Using actions gives you three benefits:

- **Readable code.** `Input.is_action_pressed("dash")` explains itself. `Input.is_key_pressed(KEY_SHIFT)` does not.
- **Several devices at once.** One action can be wired to a key, a gamepad button, and a mouse button together. The code does not change.
- **Rebindable controls.** Because the code only mentions the action name, you (or later, your players) can change which key triggers it.

Looking at the Input Map

Godot already includes some actions. Open any project and choose **Project > Project Settings**, then click the **Input Map** tab.

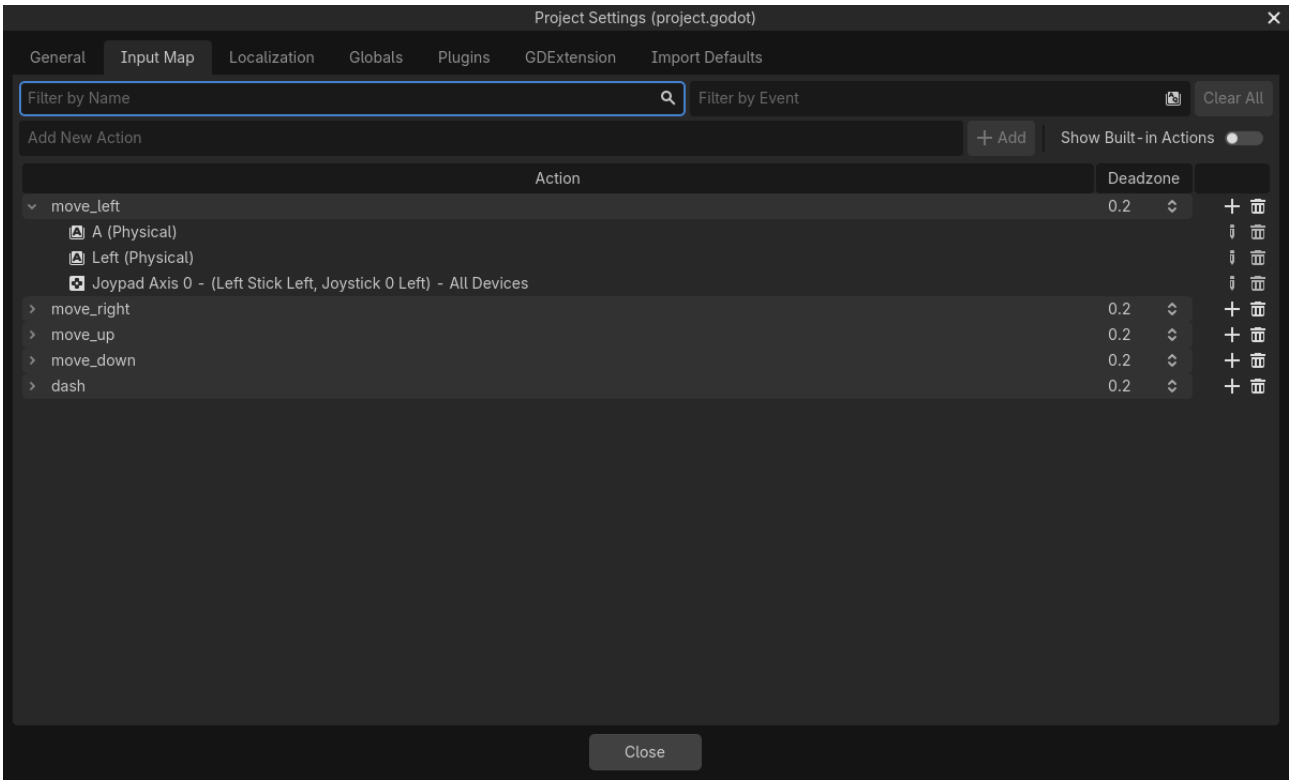


Figure 7.2: The Input Map tab. Each row is an action; expanding a row shows the inputs wired to it. The "Add New Action" box is at the top.

The list is long because Godot ships with many built-in actions whose names start with `ui_`, such as `ui_accept`, `ui_cancel`, `ui_left`, and `ui_right`. They exist so that menus work out of the box (Enter or Space presses a button; arrow keys move between buttons). If the **Show Built-in Actions** switch is off, you only see your own actions. Turn it on to look; just do not rename or delete the `ui_` ones.

Creating an action

Here is the procedure, which you will repeat in this chapter's project.

1. Open **Project > Project Settings > Input Map**.
2. In the box labelled **Add New Action**, type `move_left` and press **Enter** (or click **Add**). A new row appears.
3. Click the + at the right end of that row. A window opens titled "Event Configuration".
4. Press the key you want to bind, for example **A**. The window shows the name of the key. Click **OK**.
5. Click + again and press **Left Arrow**. Click **OK**.
6. Click + once more. This time, use the drop-down list in the window to choose **Joypad Axes**, pick **Left Stick, Joystick 0 Left** (the exact wording may differ slightly by version), and click **OK**.

The row now lists three inputs. Pressing any one of them triggers `move_left`.

Naming advice

Use lowercase words joined by underscores: `move_left`, `fire`, `open_menu`. Name the action for what it *means*. Avoid names like `press_a` or `shift_key`, because the day you bind the action to something else the name becomes a lie.

Physical keys and WASD on other keyboards

When you add a keyboard key, Godot offers a choice between **physical** and **label** keys. A *physical* key refers to the position of the key on the keyboard, regardless of what is printed on it. This matters because the W-A-S-D cluster sits at the same place on every keyboard, but on a French keyboard (AZERTY) those positions are labelled Z-Q-S-D. If you bind by *physical* position, your game's movement keys stay in the comfortable cluster on every layout. For movement keys this is nearly always what you want. The project files in this chapter use physical keys.

Deadzone

Next to each action you will find a small number called the **deadzone**, with a default of 0.2 (or 0.5 for built-in actions). It matters for analog inputs such as thumb sticks, which report a value between 0 and 1 depending on how far they are tilted. A stick that rests untouched often reports 0.03 instead of 0.0. Without a deadzone, your character would slowly drift while the player's hands are off the controller. A deadzone of 0.2 means "ignore any tilt smaller than 20 percent". Keys are either fully on or fully off, so the deadzone does not affect them.

The actions this chapter uses

Action	Keyboard	Gamepad
`move_left`	A, Left Arrow	Left stick left
`move_right`	D, Right Arrow	Left stick right
`move_up`	W, Up Arrow	Left stick up
`move_down`	S, Down Arrow	Left stick down
`dash`	Shift	A button (bottom button)

We do not make an action for the mouse click, on purpose. In section 7.4 we will see a different way to read the mouse that gives us the *position* of the click, which an action alone cannot do.

> **Try it:** Open Project Settings, go to the Input Map, and create an action named `test_action`. Bind your favourite key to it. Then delete the action again with the trash can icon on its row. Creating and deleting is free, so experiment.

7.3 Polling: asking about input every frame

Polling means that your script calls a function on every frame and acts on the answer. All polling functions live in a built-in object called `Input`. You never create it; it is always available, just like `print()`.

The four functions you will use most:

Function	Returns	Meaning
`Input.is_action_pressed("name")`	`bool`	`true` for every frame the action is *held down*
`Input.is_action_just_pressed("name")`	`bool`	`true` for *only the single frame* in which it went down
`Input.get_axis("negative", "positive")`	`float` from -1 to 1	a single direction line, such as left/right
`Input.get_vector("neg_x", "pos_x", "neg_y", "pos_y")`	`Vector2`	two direction lines at once, such as a full 2D move

(A *bool* is a value that is either `true` or `false`; you met it in Chapter 3. A *float* is a number with a decimal point.)

Held versus just pressed

This difference is the most common source of confusion for newcomers, so let us be exact.

- A **held** check answers "is the button down now?". If the player keeps the button down for one second at 60 frames per second, it answers `true` 60 times. Use it for walking, charging, holding a shield up.
- A **just pressed** check answers "did the button go down *this frame*?". During that same second of holding, it answers `true` exactly once. Use it for jumping, firing a single shot, opening a menu, toggling something.

If you jump using `is_action_pressed`, a quick tap could make you jump on several consecutive frames. If you walk using `is_action_just_pressed`, you would move one tiny step and then stop. Match the check to the idea.

There is a third sibling, `Input.is_action_just_released("name")`, which is `true` for the single frame in which the button goes *up*. It is handy for "fire when the player lets go of the bow string".

A first movement attempt

Here is the most direct way to move something left and right. Imagine a script on a `Node2D` that is already in your scene.

```
extends Node2D

func _process(delta: float) -> void:
    if Input.is_action_pressed("move_right"):
        position.x += 200.0 * delta
    if Input.is_action_pressed("move_left"):
        position.x -= 200.0 * delta
```

(Do not worry yet about `* delta`. Section 7.5 is devoted to it. For now, read it as "200 pixels per second".)

This works, but it has two weaknesses. First, it takes four `if` statements to handle four directions. Second, it does not understand analog sticks: a half-tilted stick should walk at half speed, and `is_action_pressed` only says "yes" or "no". The remedy is `get_axis` and `get_vector`.

`get_axis`: one line

`Input.get_axis("move_left", "move_right")` returns a number between -1 and 1:

- -1.0 when only the left action is held,
- +1.0 when only the right action is held,
- 0.0 when neither or both are held,
- something in between when a stick is partly tilted.

The order matters: the first name is the *negative* direction and the second the *positive* one. Our first attempt becomes:

```
extends Node2D

func _process(delta: float) -> void:
    var direction: float = Input.get_axis("move_left", "move_right")
    position.x += direction * 200.0 * delta
```

Three lines instead of five, and it supports a stick. Recall from Chapter 6 that in Godot's 2D space the x axis grows to the right, so a positive number moves things right.

`get_vector`: two lines at once

For a character that moves in all directions, use `get_vector`. It takes four action names: negative x, positive x, negative y, positive y. Remember that in Godot's 2D space the **y axis grows downward**, so "up" is the *negative* y action.

```
var direction: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
```

`direction` is now a `Vector2`. Its `x` is the horizontal push and its `y` the vertical push. A *vector* is a pair of numbers that can mean either a position or a direction with a size; here it is a direction with a size.

Why not just use two `get_axis` calls?

You can build a vector by hand like this:

```
var direction := Vector2(
    Input.get_axis("move_left", "move_right"),
    Input.get_axis("move_up", "move_down")
)
```

It works, but it has a famous flaw, shown in Figure 7.3.

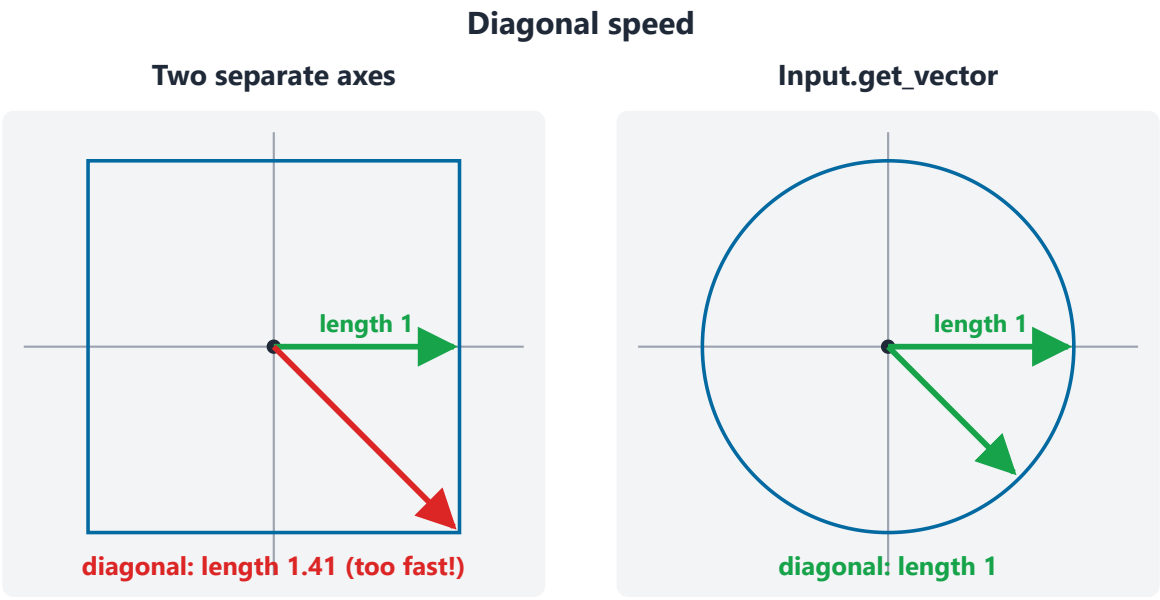


Figure 7.3: Left: two separate axes give a square of possible inputs, so a diagonal has length about 1.41 and the character runs faster diagonally. Right: `get_vector` limits the result to a circle of radius 1, so every direction has the same top speed.

If the player holds Right *and* Down, the hand-built vector is `(1, 1)`. Its length is the square root of 1 squared plus 1 squared, which is about **1.41**. Multiply by the speed and the character moves 41 percent faster along the diagonal than straight along an axis. Players call this "diagonal speed bug", and it is in the history of many real games.

`Input.get_vector` solves it for you. It limits the length of the result to at most 1 and also applies the deadzone to the pair as a whole, which feels much better with sticks than applying it to each axis separately. In practice: **for 2D movement, always use `get_vector`.**

If you ever build a direction by hand, you can fix it with `direction.normalized()`, which returns the same direction with length exactly 1. (A small catch: normalizing a vector of length 0 gives `(0, 0)`, which is fine, and there is a method `limit_length(1.0)` which only shrinks vectors that are too long, preserving half-tilted sticks. `get_vector` does the latter for you.)

Reading a pressed state with `is_action_just_pressed`

We will use `dash` as a held action in this chapter's project (hold Shift to go faster). Here is a quick example of the *just pressed* version: a script that counts how many times the player taps the dash button.

```gdscript
extends Node2D

var taps: int = 0

func _process(_delta: float) -> void:
    if Input.is_action_just_pressed("dash"):
        taps += 1
        print("Dash tapped ", taps, " times")
```

Hold Shift down for a full second: the Output panel prints one line, not sixty. The underscore in `_delta` tells Godot and other programmers that we know the parameter exists and chose not to use it. Without the underscore you receive a harmless warning.

> **Why it works:** The `Input` object records, for each action, its state in the current frame and in the previous frame. "Just pressed" simply means "down now, but up last frame". That is why it can only be `true` once per press.

7.4 Events: being told when something happens

Polling is perfect for "while held" behaviour. For one-time happenings, and for anything that carries extra information (such as *where* the mouse was clicked), Godot gives your scripts a second route: **input events**.

The callback functions

Any script can define these special functions. Godot calls them for you when an event arrives:

```
func _input(event: InputEvent) -> void:
    pass

func _unhandled_input(event: InputEvent) -> void:
    pass
```

`event` is the `InputEvent` object. The `pass` keyword means "do nothing" and is a placeholder so that the function body is not empty.

Why two functions? Because a game usually has *something else* that wants input first. Think of a pause menu with buttons: when the player clicks a button, you do not want Pip to also walk to that spot. Godot sends each event down a chain, as in Figure 7.4.

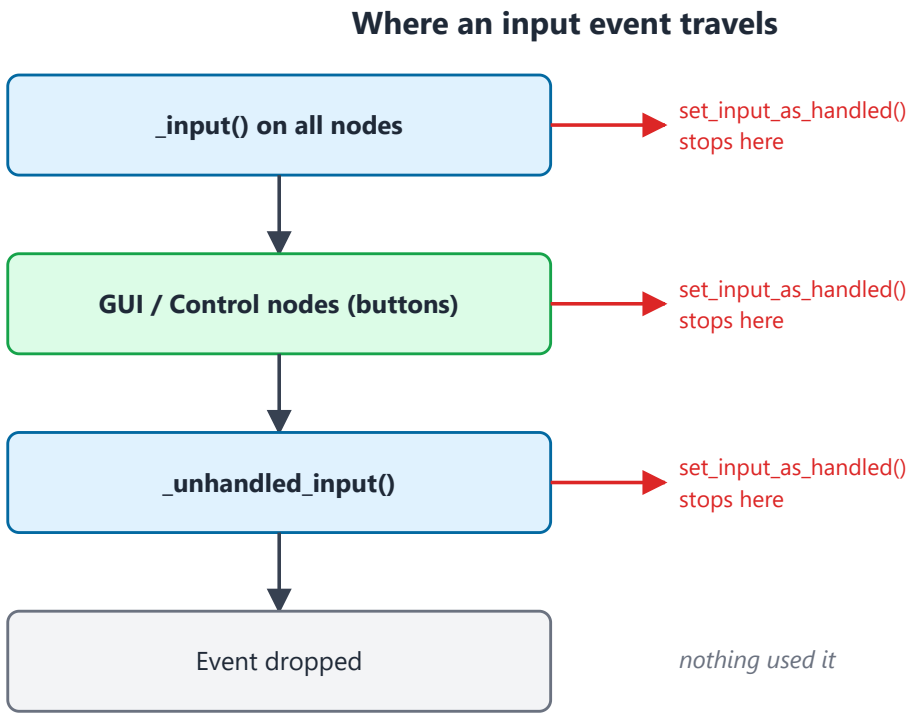


Figure 7.4: An event is offered first to `_input()` in every node, then to the user interface (buttons and other Controls), and last to `_unhandled_input()`. Any step may mark the event "handled" and stop the chain.

1. First, `_input()` is called on every node that defines it. This is the "see everything" function.
2. Next, user interface nodes (the Control nodes of Chapter 14, such as buttons) get a chance to use the event. If a button was clicked, it *consumes* the event.
3. Finally, `_unhandled_input()` is called, but only for events that nobody consumed.

Rule of thumb: use `_unhandled_input()` for gameplay (walking to a click, jumping, firing). Use `_input()` only when you really must see events even if the UI uses them. With `_unhandled_input()`, your game automatically ignores clicks that were meant for a button.

To consume an event yourself, call `get_viewport().set_input_as_handled()`. Nodes later in the chain will not see it.

Telling event types apart

Different devices produce different event classes, all of which are kinds of `InputEvent`. The most common ones:

Class	When it appears	Useful properties
`InputEventKey`	a keyboard key goes down or up	`pressed`, `keycode`, `physical_keycode`, `echo`
`InputEventMouseButton`	a mouse button or scroll wheel	`pressed`, `button_index`, `position`
`InputEventMouseMotion`	the mouse moves	`position`, `relative`
`InputEventJoypadButton`	a gamepad button	`pressed`, `button_index`
`InputEventJoypadMotion`	a gamepad stick or trigger moves	`axis`, `axis_value`
`InputEventScreenTouch`	a finger touches a touch screen	`pressed`, `position`, `index`

You check the kind of event with the `is` keyword, which you may remember from Chapter 4's type discussions. It asks: "is this thing of that class?"

```
func _unhandled_input(event: InputEvent) -> void:
    if event is InputEventMouseButton and event.pressed:
        print("Mouse button ", event.button_index, " at ", event.position)
```

Notice how the second half of the `if` uses `event.pressed`. Godot's editor knows that inside the `if`, `event` has been confirmed to be an `InputEventMouseButton`, so these properties exist.

Actions work in events, too

You can also ask an event whether it belongs to an action:

```
func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("dash"):
        print("Dash!")
```

`event.is_action_pressed("dash")` is `true` for the event in which the dash input goes down. Note that a *held* key repeats ("echoes") and generates many key events; `is_action_pressed` on an event ignores echoes by default, so this fires once per press.

See it for yourself: the Input Explorer

Reading about events is less convincing than watching them. The finished project for this chapter contains a scene called `input_explorer.tscn` whose script prints every event it receives, as text.

```
extends Control

const MAX_LINES: int = 12

@onready var log_label: Label = $Log

var _lines: Array[String] = []

func _input(event: InputEvent) -> void:
    if event is InputEventMouseMotion:
        return                      # far too many of these to list
    if event is InputEventKey and event.echo:
        return                      # skip auto-repeat while a key is held
    _lines.append(event.as_text())
    if _lines.size() > MAX_LINES:
        _lines.pop_front()
    log_label.text = "\n".join(_lines)
```

The scene tree is a `Control` called `InputExplorer` with two `Label` children, `Title` and `Log`. `event.as_text()` turns any event into readable words such as "A (Physical)" or "Left Mouse Button". `pop_front()` removes the oldest line of an array so that the list never grows too long, and `"\n".join(_lines)` glues the lines together with line breaks. Run the scene (press **F6**) and try your keyboard, mouse buttons, and, if you have one, a gamepad.

> **Try it:** Press a key and *hold* it. Because we skip echo events, you see one line. Delete the `if ... echo` check, run again, and watch the repeats flood in. Put the check back.

The mouse position

For mouse *position* in the game world, use `get_global_mouse_position()`, a method every `Node2D` has. It converts the mouse position on the window into your game world's coordinates, taking any camera movement into account (Chapter 6). When there is no camera and no zoom, it equals `event.position`, but the method stays right when you add a camera later. Prefer it.

You can also hide or capture the mouse with `Input.mouse_mode`. As a fragment (not a complete script), `Input.mouse_mode = Input.MOUSE_MODE_HIDDEN` hides the pointer, and `Input.mouse_mode = Input.MOUSE_MODE_VISIBLE` brings it back.

Changing actions while the game runs

The Input Map can be edited with code through the `InputMap` object, which is how games implement a "rebind keys" screen. Here is a fragment that replaces the keys for `dash` with the Space bar; you could place it in any `_ready()`:

```
func _ready() -> void:
    InputMap.action_erase_events("dash")
    var space_key := InputEventKey.new()
    space_key.physical_keycode = KEY_SPACE
    InputMap.action_add_event("dash", space_key)
```

You will not need this to finish the chapter, but it shows that actions are ordinary data, and Chapter 17 will use the idea to save a player's custom controls.

7.5 Delta time: movement that does not depend on your computer

Look back at our first movement example and notice the multiplication by `delta`. Without it, the code would be subtly wrong. This section explains why, and it is the single most important idea in the chapter.

Frames take time, and the time varies

A game draws pictures, called **frames**, one after another, like the pages of a flip-book. How many it can draw per second (the **frame rate**, in FPS) depends on the computer, on what else is running, and on the game itself. A fast machine may produce 144 frames per second; an old laptop may manage 30; and even on one machine the rate wobbles from moment to moment.

Godot calls your `_process()` function once per frame. Consider this beginner's script:

```
extends Node2D

func _process(_delta: float) -> void:
    position.x += 4.0
```

It moves the node 4 pixels per frame. At 60 FPS that is 240 pixels per second. On a computer running at 30 FPS it is only 120 pixels per second, so your game is *half as fast*. On a 144 FPS monitor it is 576 pixels per second, a blur. Same game, wildly different experience. Worse, if the frame rate dips for a moment, the character visibly lurches.

The fix: multiply by the time that passed

`_process()` receives a parameter named `delta`: the number of **seconds since the previous frame**. (The Greek letter delta, Δ, is the mathematician's symbol for "change in".) At 60 FPS, `delta` is about 0.0167. At 30 FPS it is about 0.0333.

Instead of saying "move 4 pixels each frame", say "move at 240 pixels *per second*, and in this frame, a `delta` fraction of a second went by":

```
extends Node2D

const SPEED: float = 240.0   # pixels per second

func _process(delta: float) -> void:
    position.x += SPEED * delta
```

Check the maths for both machines:

Frame rate	`delta` (seconds)	Pixels moved each frame	Frames in a second	Pixels in one second
60 FPS	0.0167	240 × 0.0167 = 4.0	60	240
30 FPS	0.0333	240 × 0.0333 = 8.0	30	240
144 FPS	0.0069	240 × 0.0069 = 1.67	144	240

Fewer, bigger steps or more, smaller steps: the *total* is always 240 pixels per second.

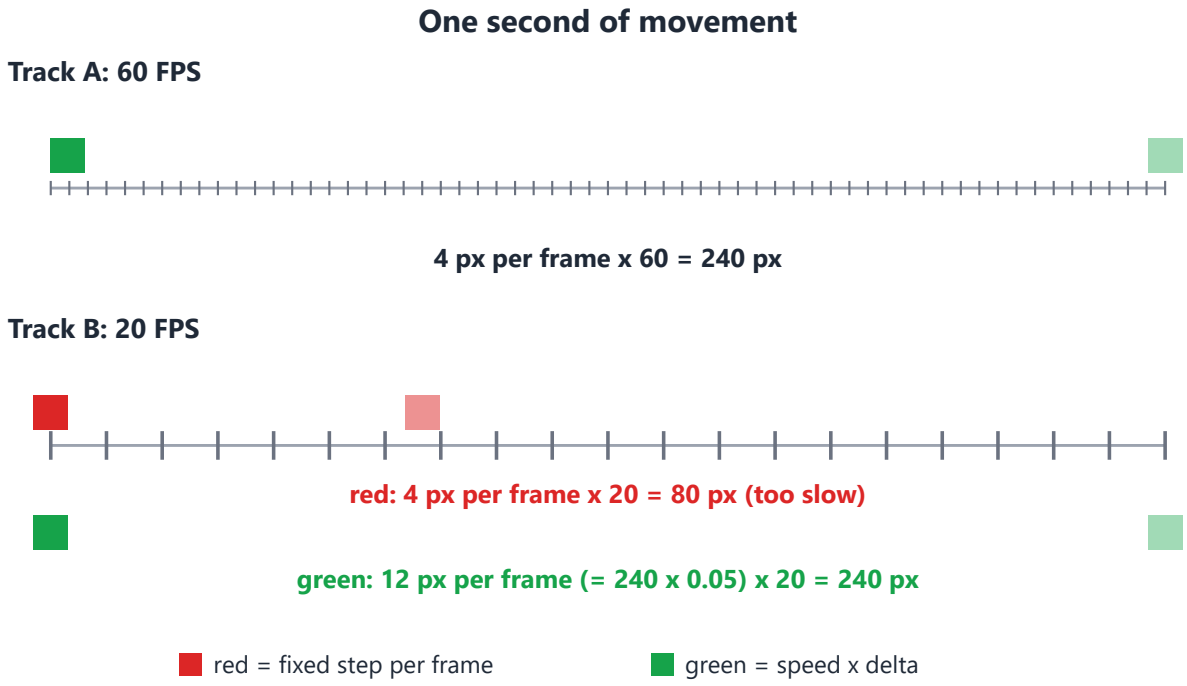


Figure 7.5: With a fixed step per frame (red), a slower frame rate means a slower square. With `speed * delta` (green), the square covers the same distance in the same time at any frame rate; only the smoothness changes.

A units sentence you can lean on

Whenever you write movement code, say the units out loud:

> `SPEED` is in **pixels per second**. `delta` is in **seconds**. Their product is in **pixels**, which is exactly what `position` is measured in.

If the units on both sides of your `+=` do not match, something is wrong. This habit will save you in 3D and in physics, too.

See it run: the delta demo

The project folder includes `scenes/delta_demo.tscn`. It shows two coloured squares: a red one that moves a fixed 4 pixels per frame and a green one that moves `240 * delta` pixels per frame. Run it with **F6**. At first they travel together. Press **Space** to cap the game at 20 frames per second, using this line:

```
Engine.max_fps = 20   # 0 means "no limit"
```

The red square slows to a third of its speed. The green one keeps pace, just with coarser steps. Press Space again to remove the cap. The script uses `Engine.get_frames_per_second()` to display the current frame rate.

`_process` or `_physics_process`?

Godot has a second per-frame function:

```
func _physics_process(delta: float) -> void:
    pass
```

- `_process(delta)` runs once for every drawn frame. Its `delta` varies.
- `_physics_process(delta)` runs at a fixed rate, 60 times per second by default, independent of how fast pictures are drawn. Its `delta` is always (nearly) 1/60.

The rule: **anything that interacts with the physics engine belongs in `_physics_process`**. That includes moving bodies with collisions, which is Chapter 8's topic. For a plain `Node2D` that only changes its `position`, `_process` is fine, and the picture updates in step with the screen. In this chapter we use `_process`. From Chapter 8 on, our moving characters will use `_physics_process`. In *both* functions you multiply by `delta`.

> **Warning:** Do not multiply input directions by `delta` *twice*, and do not multiply a value that is already a per-frame amount. If your character crawls, check that `speed` is a large number (hundreds of pixels per second, not 3 or 4) and that you apply `delta` exactly once.

7.6 Making Pip move

Now we combine everything. Our goal is a Pip who:

- moves in eight directions with WASD, the arrow keys, or the left stick, at an even speed;
- runs twice as fast while the dash action is held;
- cannot leave the window;
- turns his eyes toward where he is going;

- walks to a spot where you left-click, and stops there;
- reports each new dash press to anyone who cares, using a signal.

We build the script in stages. Each stage runs on its own so you can test as you go. The complete scene tree is:

```
Main (Node2D)            <- scripts/main.gd
├── Background (ColorRect)
├── Pip (Node2D)         <- scripts/pip.gd
└── Info (Label)
```

Stage 1: a script that draws Pip

Pip is drawn with code, just as in Chapter 6. Create `scripts/pip.gd` and attach it to the `Pip` node:

```
extends Node2D
## pip.gd -- attach to the Pip node (Node2D) in scenes/main.tscn.

@export var speed: float = 260.0
@export var radius: float = 22.0
@export var body_color: Color = Color("f4b942")

func _draw() -> void:
    draw_circle(Vector2(4, 8), radius, Color(0, 0, 0, 0.25))   # shadow
    draw_circle(Vector2.ZERO, radius, body_color)              # body
```

A word about two new pieces of syntax:

- `@export` before a variable makes it show up in the **Inspector**, so you can change `speed` without editing code. It is a favourite tool for tuning how a game *feels*.
- The part after the colon, `: float`, is a **type hint**. It tells Godot (and you) that `speed` always holds a number with a decimal point. This helps the editor warn you about mistakes and offer better auto-completion. In this book we use type hints for almost everything from now on.

The double-hash comment `##` at the top is a "documentation comment"; Godot shows it as a tooltip. It is a good habit to say at the top of each script which node it belongs to.

Stage 2: move with the keyboard

Add a `_process` function:

```
func _process(delta: float) -> void:
    var direction: Vector2 = Input.get_vector(
            "move_left", "move_right", "move_up", "move_down")
    position += direction * speed * delta
```

Run the scene. Pip should slide around smoothly. Try two keys at once, and notice the diagonal speed matches the straight speed.

Let us read the movement line piece by piece:

- `direction` has a length between 0 and 1. It is `(0, 0)` when nothing is pressed.
- `direction * speed` turns the direction into a velocity in pixels per second.
- `* delta` turns the velocity into the distance to travel *in this frame*.
- `position +=` adds that distance to where Pip is.

Stage 3: stay inside the window

Pip can currently walk off the screen and be lost. We confine him to the window rectangle, shrunk by his radius so that his *edge*, not his centre, stops at the border. Add a variable and `_ready()`:

```
var _bounds: Rect2 = Rect2()

func _ready() -> void:
    # The area Pip may stand in: the window, shrunk by Pip's radius.
    _bounds = get_viewport_rect().grow(-radius)
```

`get_viewport_rect()` returns a `Rect2`, a rectangle described by a position and a size. `grow(-radius)` shrinks it by `radius` on all four sides (a negative value shrinks; a positive one expands). The leading underscore on `_bounds` is a naming habit meaning "this variable is private to this script".

At the end of `_process`, after moving, clamp the position into that rectangle:

```
    position = position.clamp(_bounds.position, _bounds.end)
```

`Vector2.clamp(min, max)` forces each coordinate to lie between the minimum and maximum. `_bounds.position` is the top-left corner and `_bounds.end` the bottom-right. Run the scene and push Pip into each wall.

Stage 4: dash

Dashing is "move faster while a button is held", which is a polling job. Make the speed local to the frame:

```
@export var dash_multiplier: float = 2.0
```

and at the start of `_process`:

```
    var current_speed: float = speed
    if Input.is_action_pressed("dash"):
        current_speed *= dash_multiplier
```

then use `current_speed` instead of `speed` in the movement line. `*=` multiplies a variable by a value and stores the result back, the sibling of `+=`.

We store the result in a **local variable**, `current_speed`, rather than changing `speed` itself. If we modified `speed`, Pip would be fast forever after the first dash.

Stage 5: eyes that follow

A character that always looks the same feels like a sliding coin. A little feedback helps. We store which way Pip last moved and draw his eyes shifted that way. Add:

```
var _facing: Vector2 = Vector2.RIGHT
```

In `_process`, whenever there is movement, update it:

```
    if direction != Vector2.ZERO:
        _facing = direction.normalized()
```

`Vector2.ZERO` is shorthand for `Vector2(0, 0)` and `Vector2.RIGHT` for `Vector2(1, 0)`. We normalize because we want only the *direction* of the stick, not how far it is tilted.

Finally, extend `_draw()`. We also need to tell Godot to redraw every frame, since `_draw()` normally runs only once. Add `queue_redraw()` at the end of `_process`:

```
    queue_redraw()
```

and in `_draw()` after the body:

```
    var eye_center: Vector2 = _facing * 8.0
    for side in [-1.0, 1.0]:
        var offset: Vector2 = _facing.orthogonal() * 7.0 * side
        draw_circle(eye_center + offset, 4.5, Color.WHITE)
        draw_circle(eye_center + offset + _facing * 1.5, 2.2, Color("222222"))
```

`orthogonal()` returns the vector turned a quarter turn, which is "to the side" of the facing direction. The `for` loop runs twice, with `side` equal to -1 and then 1, to draw the left and right eyes.



Figure 7.6: The finished scene. Pip faces the way he moves, and a blue ring marks the spot he is walking to after a mouse click.

Stage 6: walk to the mouse click

Now the event half. We want a left click to set a destination. Keyboard movement cancels the trip. Add two variables:

```
var _target: Vector2 = Vector2.ZERO
var _has_target: bool = false
```

and set the target in `_ready()` to the starting position (`_target = position`). Then add the event handler:

```
func _unhandled_input(event: InputEvent) -> void:
    if event is InputEventMouseButton and event.pressed \
```

```
            and event.button_index == MOUSE_BUTTON_LEFT:
        _target = get_global_mouse_position()
        _has_target = true
```

The backslash `\` at the end of a line means "this statement continues on the next line". `MOUSE_BUTTON_LEFT` is a named constant for the left button (the number 1). We only react to the moment of pressing, not releasing.

Now rewrite the movement part of `_process` so keyboard and mouse share one place:

```
    if direction != Vector2.ZERO:
        _has_target = false            # the keyboard cancels a mouse trip
        _facing = direction.normalized()
        position += direction * current_speed * delta
    elif _has_target:
        var to_target: Vector2 = _target - position
        var step: float = current_speed * delta
        if to_target.length() <= step:
            position = _target
            _has_target = false
        else:
            _facing = to_target.normalized()
            position += _facing * step
```

This is the one subtle part of the script, so let us slow down.

- `_target - position` is the vector *from Pip to the target*. Its `length()` is the distance remaining.
- `step` is how far Pip may travel in this frame.
- If the remaining distance is no more than one step, we simply **snap onto the target** and stop. Otherwise Pip would step *past* the target, turn around next frame, step past again, and jitter forever. This snapping rule appears in almost every "move towards" code you will ever write.
- Otherwise, we move one step along the direction to the target.

> **Note:** Godot has a built-in that does the same job: `position = position.move_toward(_target, step)`. We wrote it out by hand once so you can see the idea. Feel free to use `move_toward` in your own scripts.

To make the destination visible, draw a ring at the target, at the bottom of `_draw()`:

```
    if _has_target:
        draw_arc(to_local(_target), 10.0, 0.0, TAU, 24, Color("3b82c4"), 2.0)
```

`_draw()` draws in Pip's own local space, so `to_local()` converts the world position of the target into that space. `TAU` is a full circle in radians (6.283...). Each `draw_*` call is a built-in function from Chapter 6.

Stage 7: a signal for the dash

The Pip script should not know about the on-screen text. Instead, it announces "I dashed!" through a signal, and whoever is interested listens (you learned this pattern in Chapter 5). At the top of `pip.gd`:

```
signal dashed  # emitted once each time the dash button is first pressed
```

and in `_unhandled_input`, after the mouse handling:

```
    if event.is_action_pressed("dash"):
        dashed.emit()
```

Here we use event input for the *single press* of dash (to count dashes) and polling for the *held* dash (to go faster). Both can use the same action.

The complete script

Here is the whole `pip.gd` in one piece, matching the file in `projects/ch07-input-and-movement/scripts/`:

```
extends Node2D
## pip.gd -- attach to the Pip node (Node2D) in scenes/main.tscn.
## Pip is drawn with code, moves with the keyboard or gamepad, and walks
## to the spot where you left-click.

signal dashed  # emitted once each time the dash button is first pressed

@export var speed: float = 260.0
@export var dash_multiplier: float = 2.0
@export var radius: float = 22.0
@export var body_color: Color = Color("f4b942")

var _target: Vector2 = Vector2.ZERO
var _has_target: bool = false
var _facing: Vector2 = Vector2.RIGHT
var _bounds: Rect2 = Rect2()

func _ready() -> void:
    # The area Pip may stand in: the window, shrunk by Pip's radius.
    _bounds = get_viewport_rect().grow(-radius)
    _target = position

func _unhandled_input(event: InputEvent) -> void:
    if event is InputEventMouseButton and event.pressed \
            and event.button_index == MOUSE_BUTTON_LEFT:
        _target = get_global_mouse_position()
        _has_target = true
    if event.is_action_pressed("dash"):
        dashed.emit()

func _process(delta: float) -> void:
    var current_speed: float = speed
    if Input.is_action_pressed("dash"):
        current_speed *= dash_multiplier

    # get_vector returns a vector of length 0 to 1, so diagonals are not faster.
    var direction: Vector2 = Input.get_vector(
            "move_left", "move_right", "move_up", "move_down")

    if direction != Vector2.ZERO:
        _has_target = false            # the keyboard cancels a mouse trip
        _facing = direction.normalized()
        position += direction * current_speed * delta
    elif _has_target:
        var to_target: Vector2 = _target - position
        var step: float = current_speed * delta
        if to_target.length() <= step:
            position = _target
            _has_target = false
        else:
            _facing = to_target.normalized()
            position += _facing * step

    position = position.clamp(_bounds.position, _bounds.end)
    queue_redraw()

func _draw() -> void:
    draw_circle(Vector2(4, 8), radius, Color(0, 0, 0, 0.25))     # shadow
    draw_circle(Vector2.ZERO, radius, body_color)                # body
    var eye_center: Vector2 = _facing * 8.0
    for side in [-1.0, 1.0]:
        var offset: Vector2 = _facing.orthogonal() * 7.0 * side
```

```
        draw_circle(eye_center + offset, 4.5, Color.WHITE)
        draw_circle(eye_center + offset + _facing * 1.5, 2.2, Color("222222"))
    if _has_target:
        # The target marker is drawn in Pip's local space, so convert to it.
        draw_arc(to_local(_target), 10.0, 0.0, TAU, 24, Color("3b82c4"), 2.0)
```

A read-out for the player (and for you)

`main.gd` is attached to `Main`. It centres Pip, listens to his signal, and prints a status line each frame:

```
extends Node2D
## main.gd -- attach to the Main node (Node2D) in scenes/main.tscn.
## Shows a live read-out of Pip's position and the frame rate.

@onready var pip: Node2D = $Pip
@onready var info: Label = $Info

var _dash_count: int = 0

func _ready() -> void:
    pip.position = get_viewport_rect().size / 2.0
    pip.dashed.connect(_on_pip_dashed)

func _process(_delta: float) -> void:
    info.text = "Pip at (%d, %d)   FPS: %d   Dashes: %d\nMove: WASD / arrows / left stick.  Dash: Shift / pad A.  Click to walk." % [
        pip.position.x, pip.position.y,
        Engine.get_frames_per_second(), _dash_count]

func _on_pip_dashed() -> void:
    _dash_count += 1
```

The `%` operator after a string fills in the `%d` placeholders ("d" for whole numbers) from the array on the right, in order. `Engine.get_frames_per_second()` reports the frame rate. `@onready` delays the assignment until the node is in the scene tree and its children exist (Chapter 5).

> **Why it works:** `Pip` knows nothing about `Main` or the label. It only emits `dashed`. `Main` connects that signal to its own function. You could delete the label and Pip would still work, and you could connect a sound effect to the same signal later. Keeping nodes ignorant of each other is what makes a game easy to change.

7.7 Gamepads, the mouse, and thinking about players

A few practical points before the project.

Gamepads need no extra code

Because we bound the left stick to the move actions, plugging in a controller just works. The stick gives analog values, and `get_vector` turns half-tilts into half speed. You can list connected pads with `Input.get_connected_joypads()` and make a controller rumble with `Input.start_joy_vibration(device, weak, strong, duration)`. Try both only if you own a controller; the chapter does not depend on them.

Mouse-only and keyboard-only play

Some players cannot use a mouse; some cannot use a keyboard. Whenever you can, let every important thing be done with more than one device. Our Pip can be driven by keys, a stick, or the mouse, and that is an accessibility feature as much as a convenience.

Pressing two opposite keys

What if the player holds Left and Right together? `get_axis` and `get_vector` add them: -1 plus +1 is 0, so Pip stands still. That is a sensible result, and you get it for free.

Tuning "feel"

Speed values are not magic. 260 pixels per second feels brisk in a window 960 pixels wide: it crosses the screen in under four seconds. Make a habit of exporting such numbers (`@export`) and adjusting them while the game runs. Choosing good numbers is half of game design.

Looking ahead

Pip currently passes through everything because he has no physical body. In Chapter 8 we will switch him to a `CharacterBody2D`, replace the position arithmetic with `velocity` and `move_and_slide()`, and give him walls, crates, and coins to meet. The input half of the script (`get_vector`, `speed`, dash) will carry over unchanged.

Project 7: Pip on the move

Goal: Build the input-and-movement scene from this chapter, from an empty project. Allow 40 to 60 minutes.

The finished project is in `projects/ch07-input-and-movement/`. If you get stuck, open it and compare. Its files are:

```
ch07-input-and-movement/
├── project.godot
├── icon.svg
├── scenes/
│   ├── main.tscn
│   ├── delta_demo.tscn
│   └── input_explorer.tscn
└── scripts/
    ├── main.gd
    ├── pip.gd
    ├── delta_demo.gd
    └── input_explorer.gd
```

Steps

1. **Create the project.** In the Project Manager click **New Project**, name it `ch07-input-and-movement`, pick a folder, choose the **Forward+** or **Compatibility** renderer (either works), and click **Create**.
2. **Set the window size.** Open **Project > Project Settings**, search for "viewport". Under **Display > Window**, set **Viewport Width** to `960` and **Viewport Height** to `540`. Set **Stretch > Mode** to `canvas_items`. Close the window.
3. **Create the actions.** Open **Project Settings > Input Map**. Add `move_left`, `move_right`, `move_up`, `move_down`, and `dash` and bind them as in the table in section 7.2 (physical keys for the keyboard). Check the deadzone values stay at 0.2.

4. **Create the main scene.** Click **Scene > New Scene**, choose **2D Scene** (a `Node2D` root), rename the root to `Main`, and save as `scenes/main.tscn`. Make the folder if the dialog offers it.
5. **Add the background.** Add a `ColorRect` child, rename it `Background`. In the Inspector set its **Size** to `960 × 540` (under **Layout > Transform**) and its **Color** to a dark blue-grey such as `#212936`.
6. **Add Pip.** Add a `Node2D` child of `Main`, rename it `Pip`. Create `scripts/pip.gd` (right-click the node, **Attach Script**, set the path) and type in the complete script from section 7.6.
7. **Add the read-out.** Add a `Label` child of `Main`, rename it `Info`, and place it in the top-left corner (position `12, 8`). Attach `scripts/main.gd` to the `Main` node and type in the script from the previous section.
8. **Set the main scene.** In **Project Settings > Application > Run > Main Scene** choose `scenes/main.tscn` (or run the scene once and let Godot offer to set it).
9. **Run it** with **F5**. Test every item in the checklist below.
10. **Delta demo.** Create a second scene with a `Node2D` root named `DeltaDemo`, a dark `ColorRect` background, two `ColorRect` squares (50 × 50) named `PerFrame` (red, at y = 200) and `WithDelta` (green, at y = 300), and a `Label` named `Label`. Attach `delta_demo.gd` (from the project folder). Run it with **F6** and press Space.
11. **Input Explorer (optional).** Create a `Control` root named `InputExplorer` with two `Label` children, `Title` and `Log`, attach `input_explorer.gd`, run with **F6**, and press things.

Test checklist

- [ ] W, A, S, D move Pip; the arrow keys do too.
- [ ] Moving diagonally is not faster than moving straight.
- [ ] Holding Shift doubles Pip's speed, and the "Dashes" counter goes up once per press (not once per frame).
- [ ] Pip cannot leave the window; his edge touches the border.
- [ ] Left-click anywhere: a blue ring appears and Pip walks to it, then stops exactly there.
- [ ] Pressing a movement key while walking to a click cancels the walk.
- [ ] In the delta demo, the red square slows down at 20 FPS and the green one does not.
- [ ] (If you have a controller) the left stick moves Pip, and pushing it halfway moves him at about half speed.

Expected result

A dark window with a golden round Pip in the middle. He glides smoothly in any direction, turns his eyes toward where he is heading, dashes when you hold Shift, and walks to wherever you click, marking the destination with a blue ring. The text at the top left shows his coordinates, the frame rate, and the dash count, updating live.

Chapter summary

- A key press travels from the device through the operating system into Godot as an `InputEvent`, then through the Input Map, which names it as an action.
- **Actions** describe intent ("move_left"), can have many triggers, and keep your code independent of devices.
- **Polling** (`Input.is_action_pressed`, `get_axis`, `get_vector`) asks about the current state every frame. **Events** (`_input`, `_unhandled_input`) tell you when something happens and carry details such as the mouse position.
- `is_action_pressed` is true while held; `is_action_just_pressed` is true for one frame only. Choose by whether the behaviour is continuous or one-off.

- For 2D movement use `Input.get_vector`; it keeps diagonals from being faster and handles deadzones and analog sticks.
- Multiply speeds by `delta`, the seconds since the last frame, so movement is measured in pixels per second and is the same on slow and fast computers.
- Use `_unhandled_input` for gameplay events so that clicks on the user interface do not leak into the game. Use `_physics_process` for physics movement (Chapter 8) and `_process` for plain per-frame changes.
- When moving towards a target, snap onto it once you are within one step, or the character will jitter around it.

Review questions

1. In your own words, what is the difference between *polling* input and *event-driven* input? Give one example of each where it is the better choice.

2. **Multiple choice.** Which call is `true` for exactly one frame when the player taps the jump key and holds it for a second? a) `Input.is_action_pressed("jump")` b) `Input.is_action_just_pressed("jump")` c) `Input.is_action_just_released("jump")` (while the key is still held) d) `Input.get_axis("jump", "jump")`

3. **True or false.** An action can only be bound to one input at a time.

4. A character built with `Vector2(Input.get_axis("move_left", "move_right"), Input.get_axis("move_up", "move_down"))` moves faster diagonally than straight. Roughly how much faster, and what built-in function avoids the problem?

5. **Multiple choice.** In `Input.get_vector("a", "b", "c", "d")`, which argument is the action that makes `y` *negative* (up on screen)? a) "a" b) "b" c) "c" d) "d"

6. What does `delta` contain, and what are its units? Roughly what value does it have at 60 FPS?

7. A script contains `position.x += 5.0` in `_process`. Explain what goes wrong on a computer running at 30 FPS compared with one running at 120 FPS, and rewrite the line to fix it.
8. **True or false.** `_unhandled_input()` receives an event even if a button on the screen has already used it.
9. Why does the "walk to the click" code check `to_target.length() <= step` before moving?
10. **Short answer.** Why is it better to bind movement keys as *physical* keys than as label keys?

Exercises

1. **Practice (easy): Change the controls.** Without editing any script, open the Input Map and add the **I, J, K, L** keys as a third way to move Pip. Add the **Space** bar to the `dash` action as well. Run the game to prove it works. Then change `speed` in the Inspector (select the `Pip` node) while the game runs to find a speed you like.
2. **Practice (easy): A speedometer.** Add a second line to the `Info` label that shows how many pixels per second Pip is currently moving. Hint: remember the position at the end of each frame and divide the distance moved by `delta`.
3. **Apply (medium): Stamina.** Give Pip a `stamina` variable from 0 to 100. Dashing drains 40 points per second; not dashing refills 20 points per second; at zero Pip cannot dash. Show the stamina on the label. You will need `clampf()` (a function that keeps a number between two limits) or `clamp()`.
4. **Apply (medium): Toggle sprint.** Change dash so that a *single press* (use `just_pressed`) switches dashing on and off, instead of holding. Print a message when it turns on or off.

5. **Challenge (hard): Smooth acceleration.** Instead of reaching full speed instantly, make Pip speed up and slow down gradually. Keep a `velocity` variable of type `Vector2` and each frame move it towards `direction * current_speed` with `velocity = velocity.move_toward(wanted, acceleration * delta)`, where `acceleration` is an exported number of pixels per second squared (try 1200). Then add `position += velocity * delta`. Experiment with larger and smaller acceleration values and describe how the "feel" changes.

Common mistakes

- **Symptom:** Pressing the key does nothing, and the Output panel shows an error like "The InputMap action "move_left" doesn't exist." -> **Cause:** The action name in the script does not match the Input Map exactly (spelling, capitals, underscores), or the action was never added. -> **Fix:** Open Project Settings > Input Map and compare character by character; copy the name from there.
- **Symptom:** Pip crawls or seems to teleport, and behaves differently on another computer. -> **Cause:** Missing `* delta`, or multiplying by it twice. -> **Fix:** Write movement as `direction * speed * delta` with `speed` in pixels per second.
- **Symptom:** A one-time action (opening a menu, shooting) fires many times for one press. -> **Cause:** You used `is_action_pressed` in `_process`, which is true on every frame the key is held. -> **Fix:** Use `is_action_just_pressed`, or handle it in `_unhandled_input` with `event.is_action_pressed`.
- **Symptom:** Clicking a button in the menu also moves the character. -> **Cause:** The gameplay code is in `_input()`, which sees the event before the button can consume it. -> **Fix:** Move it to `_unhandled_input()`.
- **Symptom:** The character is faster along diagonals. -> **Cause:** The direction was built from two separate axes without limiting its length. -> **Fix:** Use `Input.get_vector`, or call `.normalized()` / `.limit_length(1.0)` on the result.
- **Symptom:** A character walking to a target shakes back and forth around it. -> **Cause:** The step is bigger than the remaining distance, so the character overshoots every frame. -> **Fix:** If the distance is less than or equal to one step, set the position to the target and stop (or use `move_toward`).

Further reading

- Using InputEvent: https://docs.godotengine.org/en/stable/tutorials/inputs/inputevent.html
- Input examples: https://docs.godotengine.org/en/stable/tutorials/inputs/input_examples.html
- Controllers, gamepads, and joysticks: https://docs.godotengine.org/en/stable/tutorials/inputs/controllers_gamepads_joypads.html
- Mouse and input coordinates: https://docs.godotengine.org/en/stable/tutorials/inputs/mouse_and_input_coordinates.html
- Idle and physics processing: https://docs.godotengine.org/en/stable/tutorials/scripting/idle_and_physics_processing.html
- The `Input` class reference: https://docs.godotengine.org/en/stable/classes/class_input.html
- The `InputMap` class reference: https://docs.godotengine.org/en/stable/classes/class_inputmap.html

Chapter 8: Physics and Collisions

Learning objectives

By the end of this chapter you will be able to:

- Choose the right physics node for a job: `StaticBody2D`, `CharacterBody2D`, `RigidBody2D`, or `Area2D`.
- Give any physics node a body by adding a `CollisionShape2D` with a suitable shape, and see the shapes while the game runs.
- Move a `CharacterBody2D` with `velocity` and `move_and_slide()` so that it stops at, and slides along, walls.
- Detect overlaps with an `Area2D` and react to its `body_entered` and `body_exited` signals.
- Use collision **layers** and **masks** to decide exactly which things hit which, and name the layers in Project Settings.
- Push a `RigidBody2D` from code, and tune its mass, damping, gravity, and bounciness.

Key terms

- **Physics engine**: The part of Godot that moves bodies, finds out when they touch, and decides what happens next.
- **Body**: A node that takes part in physics and can be touched, such as a wall, a crate, or the player.
- **Collision shape**: The invisible outline (rectangle, circle, capsule, and so on) the physics engine uses as the real edge of a body. It is separate from what you see.
- **Collision**: The moment two shapes would overlap, and the engine's response.
- **Static body**: A body that never moves by itself: walls, floors, platforms.
- **Character body**: A body that you move yourself with code, but which stops against solid things.
- **Rigid body**: A body the physics engine moves for you, using forces, gravity, mass, and bounce.
- **Area**: A region that notices when things enter or leave it but never pushes anything.
- **Collision layer**: The "I am on this team" label of an object. A number from 1 to 32.
- **Collision mask**: The "I pay attention to these teams" list of an object.
- **Impulse**: A sudden push that changes a rigid body's velocity at once, like a kick.
- **Normal**: A direction of length 1 that points straight out of the surface that was hit.

8.1 Why we need physics, and the four kinds of nodes

In Chapter 7 Pip could walk anywhere, including through the middle of anything you placed on the screen. Setting `position` yourself is simple, but the engine has no idea Pip is supposed to be solid. This chapter gives your game a sense of touch.

A game has to answer many questions about touching:

- Did Pip walk into a wall? Then he must stop.
- Did Pip step on a coin? Then pick it up.
- Did a crate get pushed into another crate? Then both should move in a believable way.
- Did a ball fall on the floor? Then it should bounce.

Godot's physics engine answers these. You describe the world with special nodes and Godot does the hard geometry. But the engine needs to know the **role** of each object, because a wall, a player, and a crate are treated very differently. Godot offers four node types for this, shown in Figure 8.1.

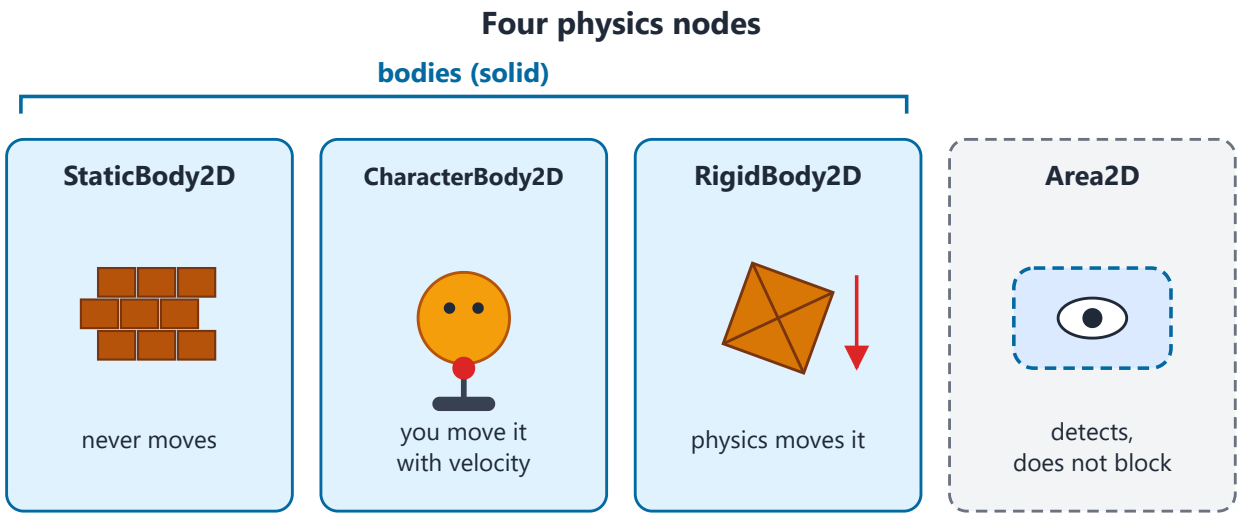


Figure 8.1: `StaticBody2D` never moves. `CharacterBody2D` is moved by your code. `RigidBody2D` is moved by the physics engine. `Area2D` is not solid at all; it only watches.

Node	Who moves it?	Is it solid?	Typical use
`StaticBody2D`	Nobody (it stays put)	Yes	Walls, floors, platforms, fixed obstacles
`CharacterBody2D`	**Your code**, using `velocity`	Yes	Player, enemies that walk, anything you steer
`RigidBody2D`	**The physics engine** (forces, gravity)	Yes	Crates, balls, rubble, anything that tumbles
`Area2D`	Whoever you like	**No**; it detects, it does not block	Coins, danger zones, doors, triggers, hit boxes

The first three are **bodies**; they take up space. An `Area2D` is more like an invisible tripwire laid on the floor. A fifth type, `AnimatableBody2D`, is a static body that you move with animation (a moving platform); we will meet it when we need it.

Which one should you pick? Ask two questions:

1. *Does it need to block other things?* If no, use an `Area2D`.
2. *Who is in charge of its movement?* If nobody, `StaticBody2D`. If you, `CharacterBody2D`. If physics, `RigidBody2D`.

> **Note:** All physics bodies inherit from `CollisionObject2D`, the shared parent that provides layers, masks, and collision shapes. When a class reference mentions `CollisionObject2D`, it applies to all four nodes.

8.2 Collision shapes: giving a body an edge

A body by itself is just a point. To be touched it needs an outline, which Godot calls a **shape**. You give it one by adding a `CollisionShape2D` child and choosing a shape in the Inspector.

The most important rule of the chapter

> **Warning:** A physics body with no `CollisionShape2D` (or with a `CollisionShape2D` that has no shape resource assigned) does nothing at all. It is invisible to physics. The editor shows a yellow warning triangle next to the node when this happens. Always read the warning.

Adding a shape

1. Add a `StaticBody2D` (or any physics node) to a scene.
2. With the body selected, click the **Add Node** button and add a `CollisionShape2D` as its child.
3. Select the `CollisionShape2D`. In the Inspector, click the empty **Shape** property, and choose **New RectangleShape2D** (or **New CircleShape2D**).
4. Click the shape in the Inspector to expand it and set its size (a rectangle) or radius (a circle). You can also drag the orange handles in the viewport.

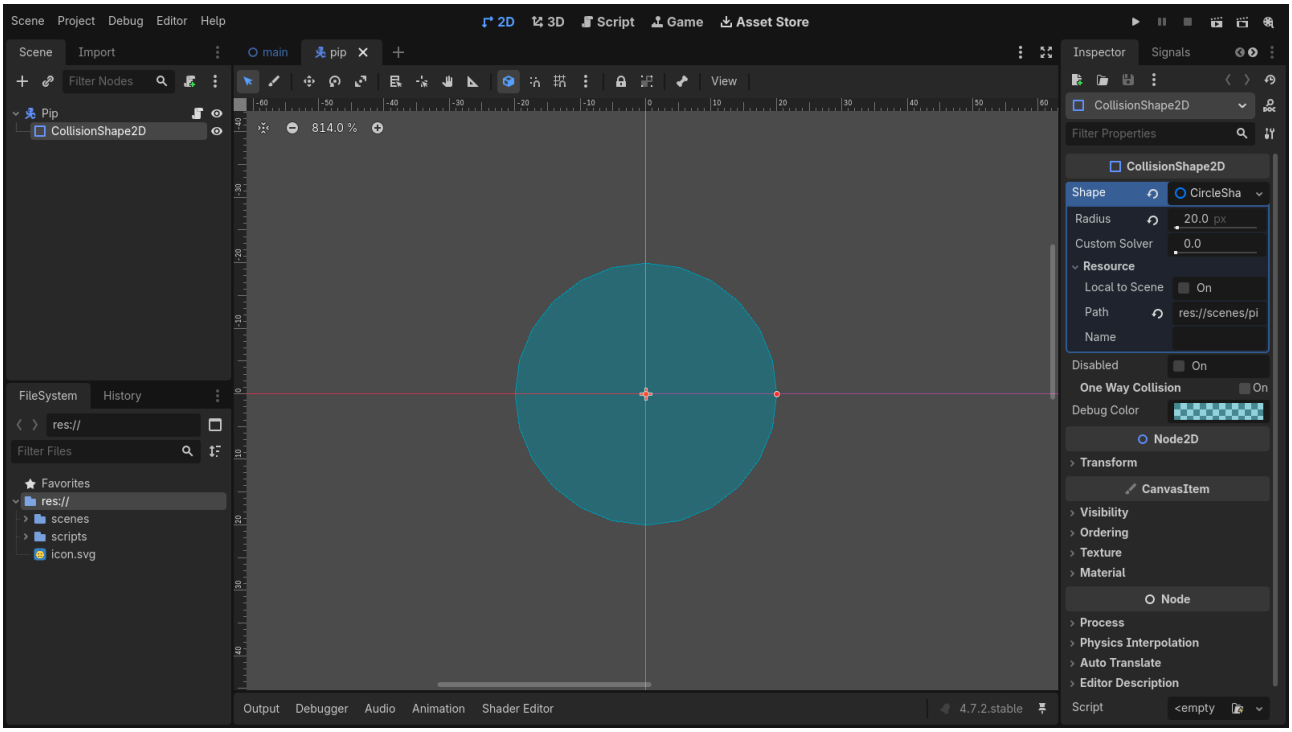


Figure 8.2: The collision shape (blue outline) is separate from the picture. Here the shape is a circle of radius 20, with handles for resizing.

The shape is **a separate thing from what you draw**. A wall might be drawn with a fancy picture but collide as a simple rectangle. The shape should be a little simpler than the picture and should match the part that matters for gameplay. A player drawn as a circle can have a circular shape; a tall character is often a capsule.

Common shapes

Shape	Good for	Notes
`RectangleShape2D`	Walls, crates, platforms	Set `size`.
`CircleShape2D`	Balls, coins, round characters, pickups	Set `radius`. Cheap and smooth.
`CapsuleShape2D`	Upright characters	A rectangle with round ends; slides nicely over small bumps.

Shape	Good for	Notes
`ConvexPolygonShape2D` / `ConcavePolygonShape2D`	Irregular outlines	Advanced; try to avoid for now.
`SegmentShape2D`, `WorldBoundaryShape2D`	Lines and endless floors	Special cases.

Shapes are centred on their `CollisionShape2D` node. The node is positioned relative to its parent body, so moving the *body* moves the shape with it.

Seeing the shapes while the game runs

Physics shapes are invisible by default, which makes mistakes hard to find. Turn them on: in the editor's top menu choose **Debug > Visible Collision Shapes**. Then press F5. Every shape is now drawn as a translucent blue overlay, and shapes that are colliding turn a different colour. When something collides "in the wrong place", this is the first tool to reach for.

Do not scale the shape

Resist the urge to stretch a `CollisionShape2D` by changing its `scale`. It leads to odd behaviour. Change the shape's own size or radius instead.

8.3 CharacterBody2D: a body you steer

`CharacterBody2D` is the node for the player. You tell it how fast to go; the engine handles walls.

The idea

A `CharacterBody2D` has a property called **`velocity`**, a `Vector2` in pixels per second. You set it according to the input. Then you call **`move_and_slide()`**. That method moves the body by `velocity` for this physics step, stops it when it hits something solid, and **slides** it along the surface instead of stopping dead, so you can run along a wall without getting stuck on it (Figure 8.3).

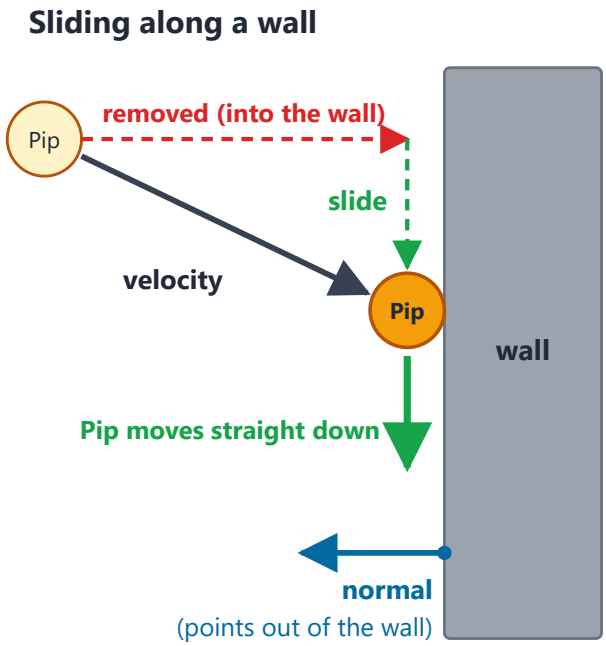


*Figure 8.3: When the body hits a wall, the part of the movement that goes into the wall is removed, and the part along the wall remains. The arrow perpendicular to the wall is the **normal**.*

Compare this with Chapter 7. There we wrote `position += direction * speed * delta`. Now we write:

```
velocity = direction * speed
move_and_slide()
```

Notice that we do **not** multiply by `delta`. `move_and_slide()` already knows how long the physics step is and applies it for us. This is the one place where the `delta` rule from Chapter 7 has an exception; it is easy to double up by mistake.

Why `_physics_process`

Physics works in fixed steps (60 per second by default). Bodies that you move with `move_and_slide()` should therefore be moved from `_physics_process()`, which runs once per physics step, rather than from `_process()`:

```
func _physics_process(_delta: float) -> void:
    # set velocity, then move
```

Pip as a CharacterBody2D

The scene for Pip is:

```
Pip (CharacterBody2D)       <- scripts/pip.gd
└── CollisionShape2D        (CircleShape2D, radius 20)
```

Pip draws himself with `_draw()`, as before, so there is no picture node. Here is the movement part of `pip.gd`:

```
extends CharacterBody2D

@export var speed: float = 240.0

func _physics_process(_delta: float) -> void:
```

```
    var direction: Vector2 = Input.get_vector(
            "move_left", "move_right", "move_up", "move_down")
    velocity = direction * speed
    move_and_slide()
```

Four lines of logic: read the input as in Chapter 7, set the velocity, move. The actions `move_left`, `move_right`, `move_up`, `move_down` are in the project's Input Map exactly as before.

Two movement styles: top-down and platformer

`CharacterBody2D` has a property **Motion Mode** with two settings:

- **Grounded** (the default) assumes gravity pulls toward the floor. It gives you `is_on_floor()`, `is_on_wall()` and slopes handling, which you will use in Chapter 13 for a platformer.
- **Floating** is for top-down games with no "down". Pip in this chapter walks around a floor seen from above, so we set **Motion Mode** to **Floating** in the Inspector (in the saved scene the property is `motion_mode = 1`).

If you leave it on Grounded in a top-down game, walking into walls can feel sticky and odd. If your character does strange things at wall corners, check this setting first.

Asking what we bumped into

After `move_and_slide()`, the body remembers every collision that happened during the move. You can ask about them:

```
for i in get_slide_collision_count():
    var hit: KinematicCollision2D = get_slide_collision(i)
    print("Bumped into ", hit.get_collider().name)
```

- `get_slide_collision_count()` says how many collisions there were (often 0 or 1; a corner gives 2).
- `get_slide_collision(i)` returns a `KinematicCollision2D`, an information object.
- `hit.get_collider()` is the node that was hit. `hit.get_normal()` is the direction pointing out of the surface that was hit, toward your body.

`for i in some_number:` loops with `i` going from 0 up to one less than the number (Chapter 4).

The other way: `move_and_collide`

If you want to move by an exact amount and just learn what you hit, use `move_and_collide(motion)`. It stops at the first collision and returns a `KinematicCollision2D` (or `null` if nothing was hit). It does not slide. It suits bullets and balls that should bounce, and it takes a distance, so there you *do* multiply by `delta`. For characters, prefer `move_and_slide()`.

Pushing crates

A `CharacterBody2D` does **not** push `RigidBody2D` crates by itself. It just treats them as walls. If you want pushing, you do it yourself with an *impulse* (section 8.5). Here is the loop we will use, a combination of the idea above and the rigid-body push:

```
for i in get_slide_collision_count():
    var hit: KinematicCollision2D = get_slide_collision(i)
    var other: Object = hit.get_collider()
```

```
    if other is RigidBody2D:
        other.apply_central_impulse(-hit.get_normal() * push_force)
```

The normal points *toward Pip*, away from the crate, so to push the crate away from Pip we use its negative. `is` checks that the thing we hit is a rigid body; walls (static bodies) lack `apply_central_impulse` and would cause an error.

8.4 StaticBody2D: the world

Walls and floors are `StaticBody2D`. They have no script; all they need is a `CollisionShape2D` and something to look at. A typical wall in our arena:

```
TopWall (StaticBody2D)        position (480, 10)
├── CollisionShape2D          (RectangleShape2D, size 960 x 20)
└── Visual (Polygon2D)        polygon from (-480,-10) to (480,10)
```

`Polygon2D` draws a flat shape from a list of points. We use it as a quick way to colour a rectangle that moves with its parent. (`ColorRect` is for user interface layouts; for things in the game world a `Polygon2D` is more convenient because it positions like any other `Node2D`.) The *visual* and the *shape* are siblings; the physics engine ignores the polygon, and the screen ignores the shape.

A static body can still have a **position**; it just does not move on its own. You may move it from code, but anything resting on it will not be carried along. For a moving platform, use `AnimatableBody2D`.

> **Try it:** In a new scene, add a `StaticBody2D` with a rectangular `CollisionShape2D`, but do **not** add a `Polygon2D`. Run with **Debug > Visible Collision Shapes** on. You see the wall only because the debug overlay draws it. That is a clean way to remember that "what you see" and "what is solid" are different things.

8.5 RigidBody2D: let the physics engine move it

A `RigidBody2D` is controlled by the physics engine. You do not set its position. You give it a starting place, and from then on forces, gravity, collisions, and friction decide where it goes. It falls, spins, bounces off walls, and tumbles.

Properties worth knowing

Property	What it does
`mass`	How heavy it is. A heavier body moves less for the same push.
`gravity_scale`	How strongly gravity affects it. 1 is normal, 0 is weightless.
`linear_damp`	A drag on straight-line motion. Bigger means it slows down faster.
`angular_damp`	A drag on spinning.
`physics_material_override`	A `PhysicsMaterial` with `bounce` (0 to 1) and `friction`.
`freeze`	Lock it in place when `true`.
`lock_rotation`	Stop it from spinning.

The default gravity comes from **Project Settings > Physics > 2D > Default Gravity** (980 pixels per second squared, pointing down). That is why a `RigidBody2D` falls as soon as the game starts, unless you set its `gravity_scale` to 0.

Top-down: no gravity, but friction

In a game viewed from above there is no "down". We want crates that **slide when pushed and then slow down**, as if on a rough floor. The recipe:

- `gravity_scale = 0` so they do not fall off the screen.
- `linear_damp = 3` and `angular_damp = 3` so they lose speed and stop.

Our crate scene is:

```
Crate (RigidBody2D)            gravity_scale 0, linear_damp 3, angular_damp 3
├── CollisionShape2D           (RectangleShape2D, size 40 x 40)
└── Visual (Polygon2D)         a brown square
```

Side-on: gravity and bounce

For a ball dropped under gravity, leave `gravity_scale` at 1 and add a `PhysicsMaterial` in the Inspector: click **Physics Material Override**, choose **New PhysicsMaterial**, and set **Bounce** to about 0.7. A bounce of 0 means "thud", and 1 means "perfectly springy" (never use exactly 1 in a real game unless you want it bouncing forever).

```
Ball (RigidBody2D)             PhysicsMaterial bounce 0.7
└── CollisionShape2D           (CircleShape2D, radius 18)
```

with the script:

```gdscript
extends RigidBody2D
## ball.gd -- attach to the Ball node (RigidBody2D) in scenes/ball.tscn.

@export var radius: float = 18.0
@export var color: Color = Color("e85d75")

func _draw() -> void:
    draw_circle(Vector2.ZERO, radius, color)
    draw_line(Vector2.ZERO, Vector2(radius, 0), Color.WHITE, 2.0)  # shows rotation
```

The white line is a trick: a plain circle looks the same however it turns, so we draw a line from the centre to show the spin.

Applying pushes

Never move a rigid body by writing to its `position` each frame; you would fight the physics engine, which makes the body jitter. Push it with forces:

- `apply_central_impulse(vector)` gives an instant change of velocity (like a kick). The size of the speed change is the impulse divided by the mass.
- `apply_central_force(vector)` applies a push for *one physics step*; call it every step for a continuous push (like a rocket).

For a rare moment when you do need to place a rigid body (teleporting it), set `linear_velocity = Vector2.ZERO` as well and be ready for the engine to do odd things; or use `freeze`.

Pushing crates with Pip

Back to the loop in section 8.3. Each physics step in which Pip presses against a crate, we add a small impulse (8 in our project). The crate gains speed little by little while `linear_damp` pulls it back, so the crate quickly settles at a steady speed that depends on those two numbers. Make `push_force` bigger for strong Pip, smaller for tired Pip, and bigger `mass` for heavy crates. Change them while the game runs and watch the difference.

> **Why it works:** A rigid body's resting speed under a constant push balances the push against the drag. A little impulse per step plus a little damping gives a steady slide. That is why the crate does not accelerate forever.

8.6 Area2D: noticing without blocking

An `Area2D` has a shape like a body, but it is not solid. Things pass through it. What it does is *notice* them and send **signals** (Chapter 5). Use it for coins, goal zones, spike traps, speed pads, doors you walk into, and "this is the range of my sword".

The signals

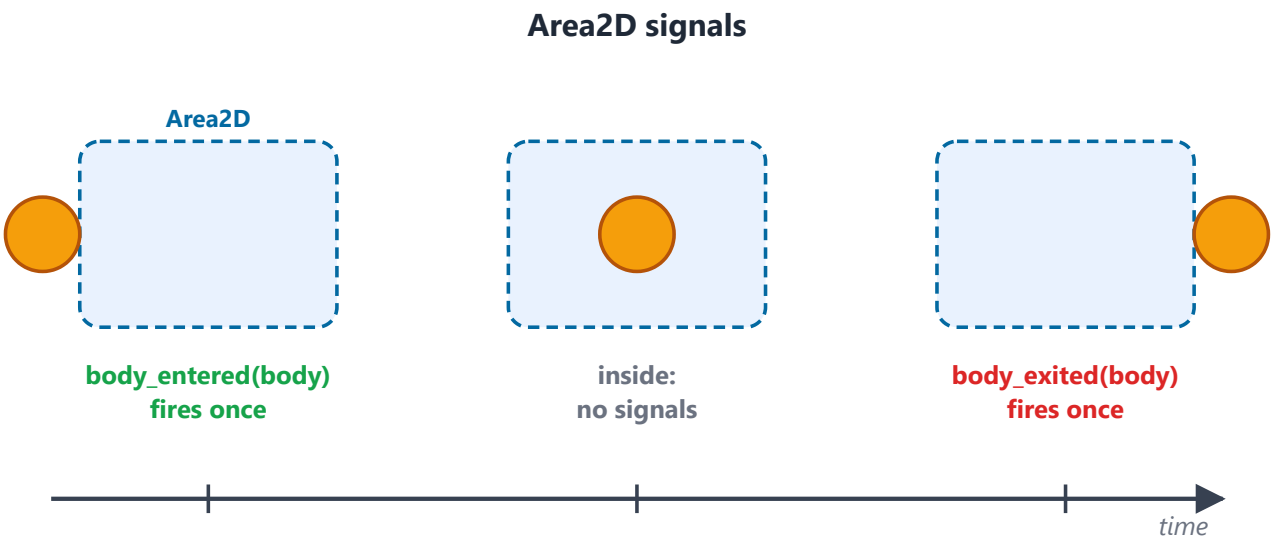


Figure 8.4: When a body's shape starts to overlap the area, `body_entered` fires once. When it stops overlapping, `body_exited` fires once. Areas also have `area_entered` and `area_exited` for other areas.

Signal	Fires when	Argument
`body_entered(body)`	a physics body starts to overlap	the body node
`body_exited(body)`	a physics body stops overlapping	the body node
`area_entered(area)`	another area starts to overlap	the area node
`area_exited(area)`	another area stops overlapping	the area node

Each fires **once** at the start or end of an overlap, not on every frame. If you need "while inside", remember the state yourself (as we do with the speed pad) or call `get_overlapping_bodies()`.

A coin

A coin is an `Area2D` that waits for the player:

```
Coin (Area2D)                  layer 4, mask 2, group "coins"
└── CollisionShape2D           (CircleShape2D, radius 12)
```

```gdscript
extends Area2D
## coin.gd -- attach to the Coin node (Area2D) in scenes/coin.tscn.

signal collected

func _ready() -> void:
    body_entered.connect(_on_body_entered)

func _on_body_entered(_body: Node2D) -> void:
    collected.emit()
    queue_free()

func _draw() -> void:
    draw_circle(Vector2.ZERO, 12.0, Color("ffd54a"))
    draw_arc(Vector2.ZERO, 12.0, 0.0, TAU, 24, Color("b8860b"), 2.0)
```

In `_ready()` the coin connects its own `body_entered` signal to its own function. (You can also connect signals in the editor's **Signals** tab; connecting in code keeps everything visible in one place.) When anything it is allowed to see touches it, it tells the world through its own custom signal, `collected`, and removes itself with `queue_free()`, which deletes a node safely at the end of the frame.

Why not decide inside the coin that "the score goes up"? A coin should not know about scores, so the level listens to `collected` and keeps the score. Each piece does one job.

A speed pad: using enter and exit

```gdscript
extends Area2D
## pad.gd -- attach to the SpeedPad node (Area2D).

@export var boost_amount: float = 1.8

func _ready() -> void:
    body_entered.connect(_on_body_entered)
    body_exited.connect(_on_body_exited)

func _on_body_entered(body: Node2D) -> void:
    if body.has_method("set_boost"):
        body.set_boost(boost_amount)

func _on_body_exited(body: Node2D) -> void:
    if body.has_method("set_boost"):
        body.set_boost(1.0)
```

`has_method("set_boost")` asks whether the thing that entered has such a function, so the pad can ignore bodies that cannot be boosted. Pip's script defines `set_boost(amount)`, which stores a multiplier that is applied in `velocity = direction * speed * boost`.

> **Warning:** Do not change an area's or a body's collision settings (turning `monitoring` on or off, adding a new `Area2D`, enabling a shape) directly inside one of these physics signals. Godot is in the middle of processing physics, and it will print `Can't change this state while flushing queries. Use call_deferred() or set_deferred() to change monitoring state instead.` Do it a moment later with `set_deferred("monitoring", false)` or `call_deferred("function_name")`. You will see this again in Chapter 9.

8.7 Layers and masks: who touches whom

So far every body has touched every other body. As a game grows, that is a problem. You do not want a coin to stop Pip. You do not want enemies' bullets to hit other enemies. You want Pip to walk through a purple "ghost wall" that blocks crates. Godot solves all of this with two settings on every physics object.

- **Layer**: "which team(s) am I on?" (what *I am*)
- **Mask**: "which team(s) do I notice?" (what *I look for*)

There are 32 layers, numbered 1 to 32. In the Inspector, under **Collision**, you see two rows of small checkboxes: **Layer** and **Mask**. Tick the numbers you want.

The rule, in plain words:

> Object **A notices** object **B** if one of A's **mask** layers is one of B's **layer** layers.

For two solid bodies to bump, it is enough that *either* one has the other in its mask. For an `Area2D` to detect a body, the **area's mask** must include the **body's layer**.

Naming the layers

Numbers are hard to remember. Give them names in **Project Settings > Layer Names > 2D Physics**. The checkboxes in the Inspector then show the names on hover. Our arena project defines:

Layer	Name	Who is on it
1	`world`	walls and the pillar
2	`player`	Pip
3	`crates`	the crates
4	`pickups`	coins
5	`ghost_walls`	the purple wall

Layers 1 to 5 are the first five checkboxes.

The collision matrix

Planning on paper first is the clearest way to avoid confusion. A row says what that object **notices** (mask); a column says what it **is** (layer):

(notices) \ (is)	world	player	crates	pickups	ghost_walls
Pip (layer: player)	yes		yes		

(notices) \ (is)	world	player	crates	pickups	ghost_walls
Crate (layer: crates)	yes	yes	yes		yes
Coin (layer: pickups)		yes			
Speed pad (no layer)		yes			
Walls (layer: world)					
Ghost wall (layer: ghost_walls)					

Read the table row by row:

- Pip collides with walls and crates. He ignores coins and the ghost wall as solids (the coin is an area; the ghost wall is simply not in his mask), so he walks straight through the ghost wall.
- Crates notice walls, Pip, other crates, and the ghost wall. The ghost wall blocks crates because *crates* look for it.
- Coins look only for the player. The speed pad does too, so a crate can slide across a pad without boosting anything.
- Walls and the ghost wall look for nothing; they are passive, and others notice them.

How the numbers are stored

Behind the checkboxes, a layer set is stored as one number: layer 1 is worth 1, layer 2 is 2, layer 3 is 4, layer 4 is 8, layer 5 is 16, and so on (powers of two, doubling each time). Add up the ticked layers.

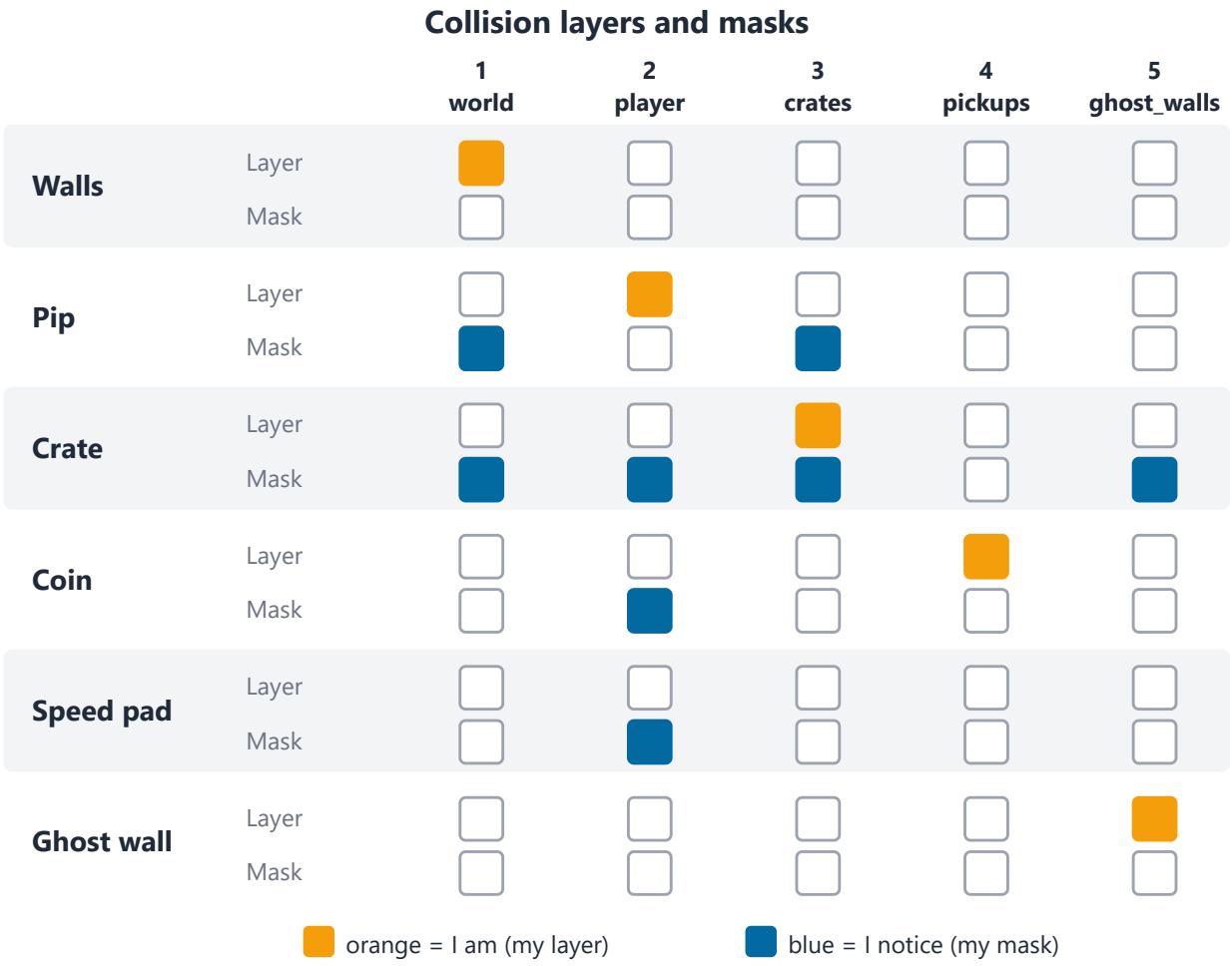


Figure 8.5: The arena's layers and masks. Each object has one row of "layer" boxes and one row of "mask" boxes. Pip: layer 2, mask 1 and 3 (value 5). Crate: layer 3, mask 1, 2, 3, 5 (value 23).

Object	collision_layer	collision_mask
Walls, pillar (default)	1	1 (default, unused)
Pip	2	5 (= 1 + 4: world, crates)
Crate	4	23 (= 1 + 2 + 4 + 16: world, player, crates, ghost)
Coin	8	2 (player)
Speed pad	0	2 (player)
Ghost wall	16	0

You will see these numbers when you open `scenes/*.tscn` files. You rarely type them; you tick boxes. But reading them is a useful skill.

Changing layers from code

In code you almost never touch the big numbers. Use the helper methods that take a layer *number* from 1 to 32:

```
set_collision_layer_value(2, true)    # put this object on layer 2
set_collision_mask_value(3, false)    # stop noticing layer 3
```

These are fragments for use inside any script that extends a physics node.

A habit that saves hours

Whenever "two things refuse to collide" or "two things collide that should not", open both objects' layer and mask rows and apply the rule in plain words: *does A's mask include B's layer?* Almost every collision bug is answered by that sentence.

Project 8: The Physics Arena

Goal: Build a small top-down room where Pip walks, bumps into walls, pushes crates, collects coins, runs over a speed pad, and finds out about the ghost wall. Then build a second scene with bouncing balls. Allow 60 to 90 minutes.

The finished project is in `projects/ch08-physics-and-collisions/`:

```
ch08-physics-and-collisions/
├── project.godot          (input actions + layer names)
├── icon.svg
├── scenes/
│   ├── main.tscn          (the arena)
│   ├── pip.tscn
│   ├── crate.tscn
│   ├── coin.tscn
│   ├── ball.tscn
│   └── bounce_demo.tscn
└── scripts/
    ├── arena.gd
    ├── pip.gd
    ├── coin.gd
    ├── pad.gd
    ├── ball.gd
    └── bounce_demo.gd
```

Part A: setup

1. Create a new project named `ch08-physics-and-collisions`. In Project Settings set the viewport to `960 × 540`, **Stretch > Mode** to `canvas_items`.
2. In **Input Map** add `move_left`, `move_right`, `move_up`, and `move_down` with WASD, the arrow keys, and the left stick, as in Chapter 7.
3. In **Project Settings > Layer Names > 2D Physics**, name layers 1 to 5: `world`, `player`, `crates`, `pickups`, `ghost_walls`.

Part B: Pip

4. **Scene > New Scene**, choose **Other Node**, and pick `CharacterBody2D`. Rename it `Pip`. Save as `scenes/pip.tscn`.
5. Add a `CollisionShape2D` child. Give it a **New CircleShape2D** with radius `20`.
6. Select `Pip`. In the Inspector, set **Motion Mode** to `Floating`. Under **Collision**, set **Layer** to only box 2 (`player`) and **Mask** to boxes 1 and 3 (`world`, `crates`).
7. Attach `scripts/pip.gd`:

```gdscript
extends CharacterBody2D
## pip.gd -- attach to the Pip node (CharacterBody2D) in scenes/pip.tscn.
## Top-down movement with move_and_slide(), plus pushing RigidBody2D crates.

@export var speed: float = 240.0
@export var push_force: float = 8.0   # impulse added to a crate every physics frame
```

```
@export var radius: float = 20.0

var boost: float = 1.0

func set_boost(amount: float) -> void:
    boost = amount
    queue_redraw()

func _physics_process(_delta: float) -> void:
    var direction: Vector2 = Input.get_vector(
            "move_left", "move_right", "move_up", "move_down")
    velocity = direction * speed * boost
    move_and_slide()

    # After sliding, ask what we bumped into.
    for i in get_slide_collision_count():
        var hit: KinematicCollision2D = get_slide_collision(i)
        var other: Object = hit.get_collider()
        if other is RigidBody2D:
            # The normal points from the crate toward Pip, so push the opposite way.
            other.apply_central_impulse(-hit.get_normal() * push_force)

func _draw() -> void:
    var color: Color = Color("f4b942") if boost == 1.0 else Color("ffe08a")
    draw_circle(Vector2.ZERO, radius, color)
    draw_circle(Vector2(7, -4), 4.0, Color.WHITE)
    draw_circle(Vector2(-7, -4), 4.0, Color.WHITE)
    draw_circle(Vector2(8, -4), 2.0, Color("222222"))
    draw_circle(Vector2(-6, -4), 2.0, Color("222222"))
```

The one-line `Color("f4b942") if boost == 1.0 else Color("ffe08a")` is a **conditional expression**: it picks the first colour when the test is true and the second when false. Pip looks paler while boosted.

Part C: crate and coin

8. New scene, root `RigidBody2D` named `Crate`, saved as `scenes/crate.tscn`. Add a `CollisionShape2D` with a **RectangleShape2D** of size `40 × 40`, and a `Polygon2D` named `Visual` with polygon points `(-20,-20) (20,-20) (20,20) (-20,20)` and a brown colour. In the Inspector set **Gravity Scale** `0`, **Linear > Damp** `3`, **Angular > Damp** `3`. Collision: **Layer** box 3, **Mask** boxes 1, 2, 3, 5.
9. New scene, root `Area2D` named `Coin`, saved as `scenes/coin.tscn`. Add a `CollisionShape2D` with a **CircleShape2D** of radius `12`. Open the **Groups** tab (next to the Inspector) and add the group `coins`. Collision: **Layer** box 4, **Mask** box 2. Attach `scripts/coin.gd` (the script in section 8.6).

Part D: the arena

10. New scene with root `Node2D` named `Main`, saved as `scenes/main.tscn`. Attach `scripts/arena.gd`:

```
extends Node2D
## arena.gd -- attach to the Main node (Node2D) in scenes/main.tscn.

@onready var pip: CharacterBody2D = $Pip
@onready var hud: Label = $HUD

var _total: int = 0
var _collected: int = 0

func _ready() -> void:
    for coin in get_tree().get_nodes_in_group("coins"):
        _total += 1
        coin.collected.connect(_on_coin_collected)
    _update_hud()

func _on_coin_collected() -> void:
    _collected += 1
```

```
func _process(_delta: float) -> void:
    _update_hud()

func _update_hud() -> void:
    var state: String = "FAST" if pip.boost > 1.0 else "normal"
    hud.text = "Coins: %d / %d   Speed: %s\nWASD / arrows / stick. Push the crates. The purple wall stops
crates, not Pip." % [
        _collected, _total, state]
```

`get_tree().get_nodes_in_group("coins")` returns every node in the group `coins`; the arena connects to each coin's `collected` signal.

11. Build the scene tree below. Instance `pip.tscn`, `crate.tscn`, and `coin.tscn` with **Instantiate Child Scene** (the chain-link icon), as in Chapter 2.

```
Main (Node2D)                 <- scripts/arena.gd
├── Floor (ColorRect)         size 960 x 540, dark green
├── SpeedPad (Area2D)         position (300, 420), layer none, mask 2, <- scripts/pad.gd
│   ├── CollisionShape2D      (RectangleShape2D 100 x 60)
│   └── Visual (Polygon2D)    green, semi-transparent
├── TopWall (StaticBody2D)    position (480, 10),  shape 960 x 20
├── BottomWall (StaticBody2D) position (480, 530), shape 960 x 20
├── LeftWall (StaticBody2D)   position (10, 270),   shape 20 x 540
├── RightWall (StaticBody2D)  position (950, 270),  shape 20 x 540
├── Pillar (StaticBody2D)     position (480, 270),  shape 60 x 120
├── GhostWall (StaticBody2D)  position (740, 330), layer 5, mask none, shape 16 x 200
├── Crate1, Crate2, Crate3    (instances of crate.tscn)
├── Coin1 ... Coin4           (instances of coin.tscn)
├── Pip                       (instance of pip.tscn) at (120, 270)
└── HUD (Label)               position (30, 28)
```

Each wall gets a `CollisionShape2D` and a `Polygon2D` named `Visual`, like the `TopWall` example in section 8.4. The speed pad uses `scripts/pad.gd` from section 8.6.

12. In **Project Settings > Application > Run > Main Scene**, choose `scenes/main.tscn`.
13. Turn on **Debug > Visible Collision Shapes** and press **F5**.

Part E: the bounce demo

14. Create `scenes/ball.tscn` (a `RigidBody2D` named `Ball` with a **CircleShape2D** radius 18, a **PhysicsMaterial** with bounce 0.7, and `scripts/ball.gd` from section 8.5).
15. Create `scenes/bounce_demo.tscn`: a `Node2D` root with a dark `ColorRect`, a `StaticBody2D` floor at `(480, 520)` with a 960 × 20 rectangle shape, and a `Label`. Attach `scripts/bounce_demo.gd`:

```
extends Node2D
## bounce_demo.gd -- attach to the root Node2D of scenes/bounce_demo.tscn.
## Run with F6. Click anywhere to drop a bouncing ball (gravity is ON here).

const BallScene: PackedScene = preload("res://scenes/ball.tscn")

func _unhandled_input(event: InputEvent) -> void:
    if event is InputEventMouseButton and event.pressed \
            and event.button_index == MOUSE_BUTTON_LEFT:
        drop_ball(get_global_mouse_position())

func drop_ball(where: Vector2) -> RigidBody2D:
    var ball: RigidBody2D = BallScene.instantiate()
    ball.position = where
    ball.color = Color.from_hsv(randf(), 0.6, 0.95)
    add_child(ball)
    return ball
```

`preload()` loads a scene file ahead of time into a `PackedScene`. `instantiate()` makes a fresh copy (the code version of dropping a scene into a tree), and `add_child()` puts it into the game. You will use this exact pattern to spawn coins in Chapter 9.

16. Run with **F6** and click many times.

Test checklist

- [ ] Pip stops at the walls and slides along them when you press diagonally.
- [ ] Pip cannot go through the pillar in the middle.
- [ ] Walking into a crate pushes it; it slows to a stop after you let go.
- [ ] Pushing a crate to the right stops it at the purple ghost wall, but Pip can walk through the same wall.
- [ ] Touching a coin makes it vanish and raises the count in the label.
- [ ] Standing on the green pad makes Pip faster (and paler); leaving it restores normal speed.
- [ ] Crates sliding across the pad do not change Pip's speed.
- [ ] In the bounce demo, each click drops a coloured ball that bounces a few times and rests on the floor.

Expected result

A dark green room with a grey pillar and walls. Pip, a golden circle, moves with WASD or the stick, pushes three brown crates around, collects four gold coins, speeds up on a green pad, and finds that a purple wall stops the crates but lets him through. A label at the top-left counts coins and shows his speed state. In the second scene, balls fall under gravity and bounce.

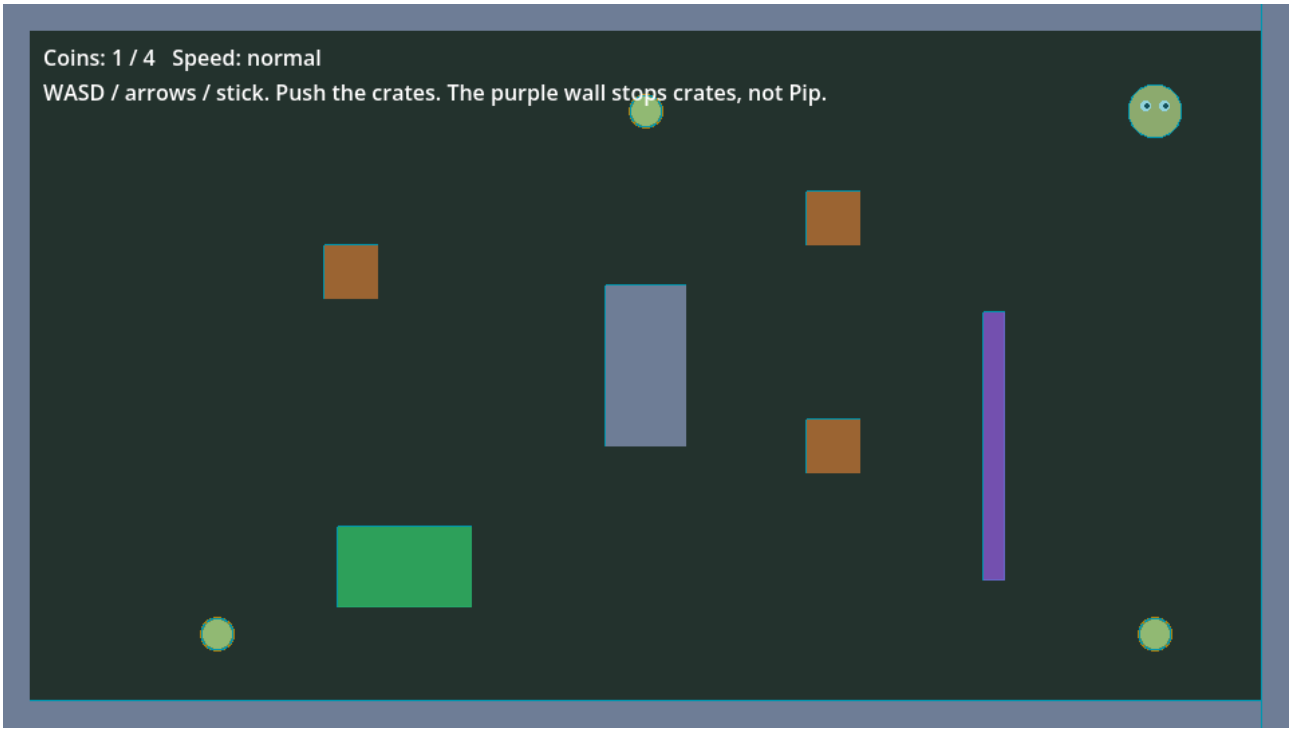


Figure 8.6: The finished arena. The blue overlays are the collision shapes; notice that the coins and the speed pad are shapes too, even though they are not solid.

Chapter summary

- Godot has four main 2D physics nodes: `StaticBody2D` (fixed), `CharacterBody2D` (moved by your code), `RigidBody2D` (moved by the physics engine), and `Area2D` (detects, does not block).

- Every physics node needs a `CollisionShape2D` with a shape resource, or it does nothing. Use **Debug > Visible Collision Shapes** to see them.
- For a `CharacterBody2D`, set `velocity` and call `move_and_slide()` in `_physics_process`. Do not multiply the velocity by `delta`; the method does that.
- `get_slide_collision()` tells you what the body hit and the surface normal, which is how you push rigid bodies with `apply_central_impulse()`.
- Rigid bodies are controlled with forces, mass, damping, gravity scale, and a `PhysicsMaterial`; do not write to their position every frame.
- An `Area2D` sends `body_entered`, `body_exited`, `area_entered`, and `area_exited` once at the start and end of an overlap. Do not change physics state inside those callbacks; use `set_deferred()` or `call_deferred()`.
- Layers say what an object *is*, masks say what it *notices*. A notices B if A's mask includes one of B's layers. Name layers in Project Settings and plan them in a table.

Review questions

1. Which of the four physics nodes would you choose for (a) a floor, (b) the player, (c) a coin, (d) a barrel that rolls downhill?
2. **True or false.** A `RigidBody2D` with no `CollisionShape2D` child will still collide with walls.
3. In a `_physics_process` function, you write `velocity = direction * speed * delta` and then call `move_and_slide()`. What is wrong, and what will the player notice?
4. **Multiple choice.** What does `get_normal()` of a `KinematicCollision2D` return? a) The distance to the wall b) A direction of length 1 pointing out of the surface that was hit c) The speed of the other body d) The name of the node that was hit
5. Why does Pip need the `for` loop with `apply_central_impulse` to push a crate, rather than just walking into it?
6. **Multiple choice.** A coin has layer 4 and mask 2, and Pip has layer 2. Which statement is correct? a) The coin can see Pip, because the coin's mask includes Pip's layer. b) The coin can see Pip only if Pip's mask includes layer 4. c) They can never interact. d) The coin is solid and blocks Pip.
7. How many times does `body_entered` fire if Pip stands on a speed pad for ten seconds?
8. **True or false.** Setting a `RigidBody2D`'s `gravity_scale` to 0 makes it ignore all forces and impulses.
9. In the arena, why can Pip walk through the ghost wall while the crates cannot?
10. What error message do you get if you switch an area's `monitoring` on directly inside a `body_entered` function, and what is the fix?

Exercises

1. **Practice (easy): Heavier crates.** Select one crate and set its `mass` to `5` in the Inspector. Push it, then push a crate with mass 1. Describe the difference. Then try `linear_damp` values of 0.5 and 8 and say which feels like ice and which like mud.
2. **Practice (easy): Show the shapes.** Turn on **Debug > Visible Collision Shapes**, run the arena, and take note of every shape's colour. Then deliberately make Pip's collision circle much bigger than his drawing (radius 40) and describe what changes in play. Put it back to 20.

3. **Apply (medium): A slippery floor.** Add a second `Area2D` named `IcePatch`. When Pip is inside, make him accelerate and decelerate slowly. Hint: when `body_entered` fires, call a new Pip method that sets a flag; in `_physics_process`, replace `velocity = direction * speed * boost` with `velocity = velocity.move_toward(direction * speed * boost, 400.0 * delta)` when the flag is on (use the `delta` parameter there, since this is a calculation of your own).
4. **Apply (medium): A new layer.** Add an enemy slime: a `CharacterBody2D` on a new layer 6 named `enemies`, standing still. Decide Pip's mask, the crates' mask, and the slime's mask so that: Pip is blocked by slimes, crates are blocked by slimes, and slimes are not blocked by coins. Write the collision matrix first, then set the checkboxes.
5. **Challenge (hard): Push the crates onto targets.** Add three target `Area2D` zones (rings on the floor) on the right side of the room. Count how many crates are fully inside a target (use `body_entered` and `body_exited` and a counter) and show "Crates home: 2 / 3" in the label. When all three are home, print "You win!". Remember the crates are on layer 3, so the targets' masks must include it.

Common mistakes

- **Symptom:** Pip walks straight through the wall. -> **Cause:** The wall has no `CollisionShape2D`, its shape is empty, or Pip's mask does not include the wall's layer. -> **Fix:** Look for the yellow warning triangle; turn on Visible Collision Shapes; compare Pip's mask with the wall's layer.
- **Symptom:** The character is too fast, jittery, or moves strangely. -> **Cause:** `velocity` was multiplied by `delta` before `move_and_slide()`, or `position` was changed directly together with `move_and_slide()`. -> **Fix:** Set only `velocity`, as pixels per second, then call `move_and_slide()` once per physics step.
- **Symptom:** Crates fall off the bottom of the screen. -> **Cause:** In a top-down game the rigid bodies still have normal gravity. -> **Fix:** Set `gravity_scale` to 0 (and add `linear_damp` so they slow down).
- **Symptom:** A coin or pad never reacts. -> **Cause:** The area's mask does not include the other object's layer, or the signal was never connected. -> **Fix:** Check the mask and add a `print()` in the connected function to prove it is called.
- **Symptom:** An error says "Can't change this state while flushing queries." -> **Cause:** A physics setting was changed inside a collision callback. -> **Fix:** Use `set_deferred("property", value)` or `call_deferred("method")`.
- **Symptom:** The character gets stuck on tile joints or tiny corners, or slides weirdly in a top-down game. -> **Cause:** Motion Mode is still Grounded. -> **Fix:** Set Motion Mode to Floating for top-down movement, and use a circle or capsule shape for the character.

Further reading

- Physics introduction: https://docs.godotengine.org/en/stable/tutorials/physics/physics_introduction.html
- Using CharacterBody2D: https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html
- Rigid body: https://docs.godotengine.org/en/stable/tutorials/physics/rigid_body.html
- Using Area2D: https://docs.godotengine.org/en/stable/tutorials/physics/using_area_2d.html
- Collision shapes (2D): https://docs.godotengine.org/en/stable/tutorials/physics/collision_shapes_2d.html
- The `CharacterBody2D` class reference: https://docs.godotengine.org/en/stable/classes/class_characterbody2d.html

Chapter 9: Mini-Game 1, Coin Dash

Learning objectives

By the end of this chapter you will be able to:

- Plan a small game on paper: its rules, its game loop, its scenes, and the signals that connect them, before writing code.
- Split a game into small reusable scenes (player, coin, hazard, HUD) and combine them in a main scene.
- Spawn many copies of a scene at random positions from code with `preload()` and `instantiate()`.
- Track score, level, and a countdown, and show them on a heads-up display (HUD).
- Manage the phases of a game (title, playing, game over) with an `enum` and a simple state variable.
- Test, tune, and debug a finished game, including the "flushing queries" error that every Godot developer meets once.

Key terms

- **Game loop**: The cycle a game repeats: take input, update the world, draw the result. Also used for the larger cycle "start, play, lose, start again".
- **Game state**: Which phase the game is in right now (for example title, playing, game over).
- **Enum** (`enum`): A named list of choices, such as `TITLE`, `PLAYING`, `GAME_OVER`, stored as numbers behind the scenes. It makes states readable.
- **Spawning**: Creating a new object in the running game.
- **Instance / instantiate**: A fresh copy of a scene made from the saved scene file, ready to be added to the tree.
- **HUD** (heads-up display): The score, timer, and messages drawn over the game.
- **CanvasLayer**: A node that draws its children on a separate layer that does not move with the camera, which makes it ideal for a HUD.
- **Timer**: A node that counts down and sends a `timeout` signal when its wait time has passed.
- **Group**: A named label you can attach to nodes so you can find or test them together (for example `coins`).
- **Playtesting**: Playing your own game (or watching someone else play it) specifically to find problems and to tune the difficulty.

9.1 The game: Coin Dash

You now know how to read input, move things, and detect touches. This chapter puts everything together into a complete small game, with a title screen, a score, a goal, a way to lose, and a reason to play again. It is short, but it is a *finished* game, and finishing things is a skill of its own.

The story. Pip is in a small field full of coins. Slimes drift around. The clock is ticking.

The rules.

1. Pip starts with **20 seconds**.
2. Each coin is worth **10 points**.
3. When Pip collects **all** the coins, the next **level** begins: more coins, one more slime (up to five), and **5 bonus seconds**.
4. Touching a slime ends the game at once.

5. When the time reaches zero, the game ends.
6. The game remembers the best score of this session (saving it to disk is Chapter 17).

That is a game. Notice how small the rule list is, and how many interesting decisions come from it: do you grab the safe coins first or risk the one next to the slime?

Plan before you build

Experienced developers sketch before they code. A good first sketch answers three questions: *what are the pieces*, *what does each piece know*, and *how do the pieces talk*?

Figure 9.1 shows the phases of the game. The game is always in exactly one of three states.

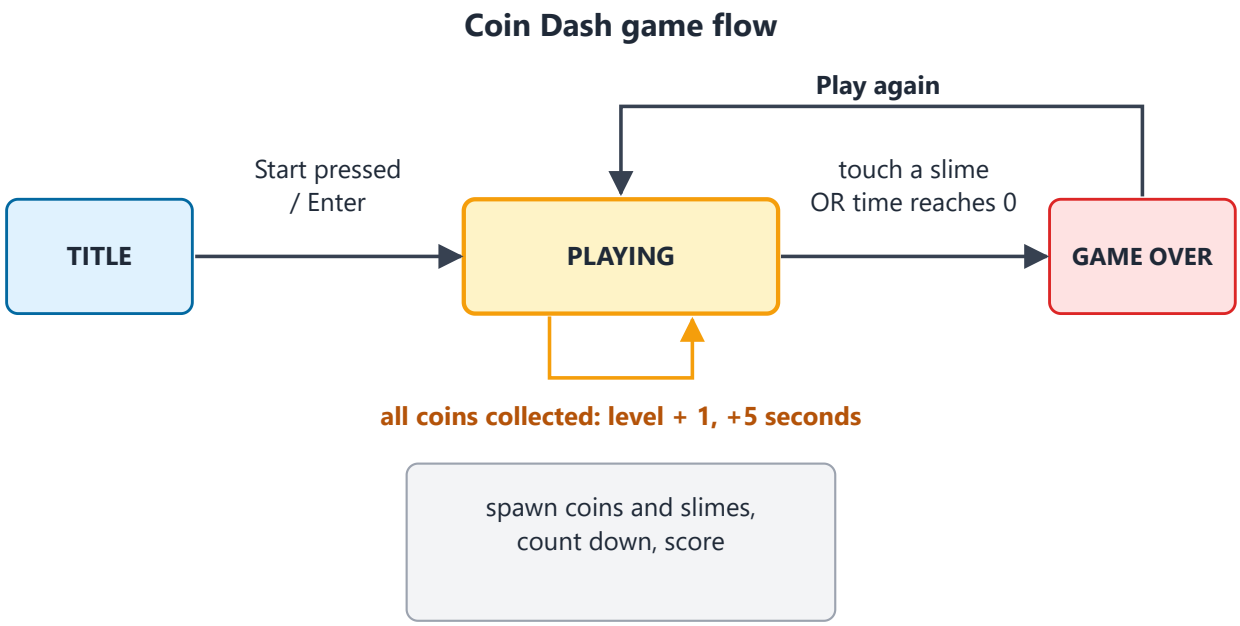


Figure 9.1: Title, Playing, and Game Over. Pressing Start moves from Title or Game Over to Playing. Touching a slime or running out of time moves from Playing to Game Over.

Figure 9.2 shows the pieces and their conversations.

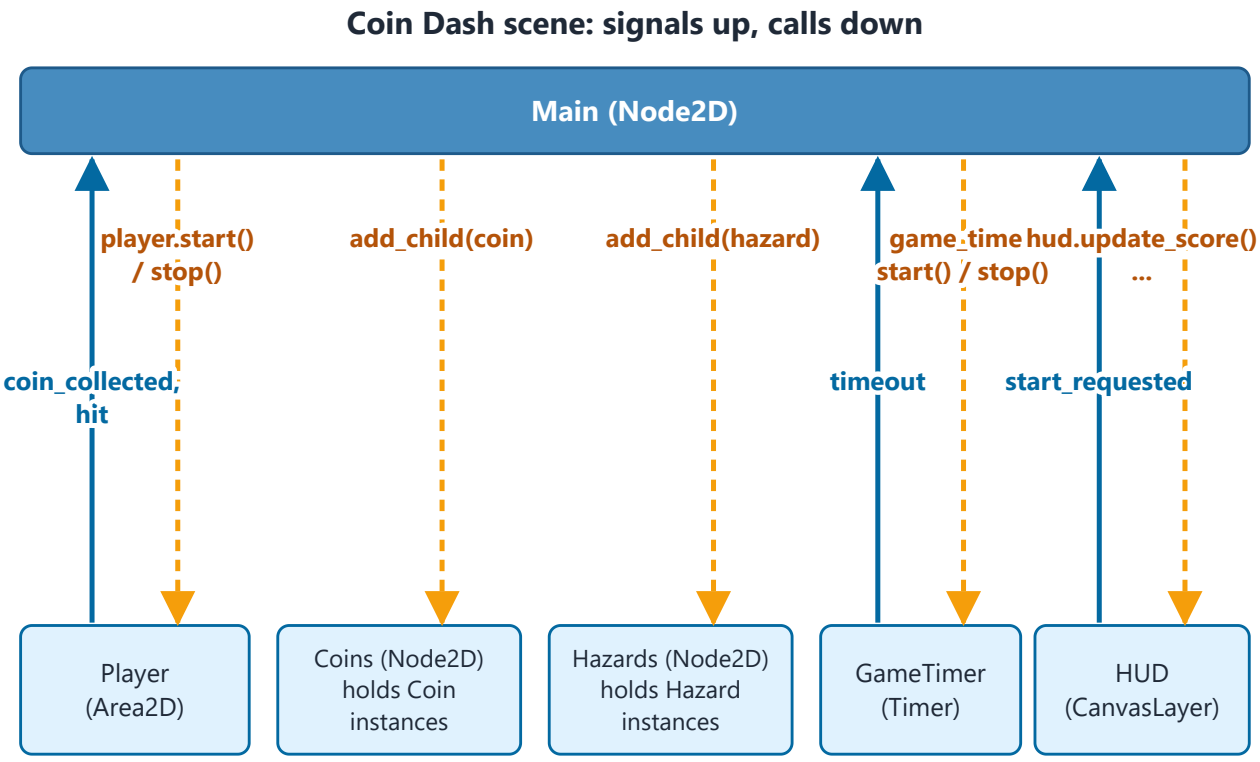


Figure 9.2: Main owns the rules. Player, Coin, Hazard, and HUD are separate scenes. Arrows show signals travelling from the small pieces up to Main; Main calls functions downward.

Scene	Node type	Job	Knows about
`player.tscn`	`Area2D`	Moves Pip; reports touches	Nothing else. It emits `coin_collected` and `hit`.
`coin.tscn`	`Area2D`	Sits there; pulses; disappears when picked up	Nothing else
`hazard.tscn`	`Area2D`	A slime that drifts and bounces off the edges	Nothing else
`hud.tscn`	`CanvasLayer`	Shows score, level, time, and the title / game-over message	Nothing else. It emits `start_requested`.
`main.tscn`	`Node2D`	Owns the rules: state, score, timer, spawning	All of the above

A useful slogan: **signals go up, function calls go down.** A child announces what happened ("I was hit!") without knowing who is listening. The parent, which knows its children, tells them what to do ("start moving!"). This arrangement means you can change or reuse any small scene without breaking the rest. You will use this slogan in every game from now on.

Why `Area2D` for everything?

In Chapter 8 Pip was a `CharacterBody2D` and coins were areas. Here Pip is also an `Area2D`. Why the change? Because in Coin Dash nothing is solid: there are no walls to bump and nothing to push. Pip only needs to *notice* coins and slimes, and the screen edge is enforced by simply clamping his position, as in Chapter 7. A simpler node fits a simpler job. Choosing the least powerful node that does the job is good design.

9.2 Setting up the project

Create a project called `ch09-coin-dash`.

1. **Window.** In **Project Settings > Display > Window**, set Viewport Width `960`, Height `540`, and Stretch **Mode** to `canvas_items`.
2. **Input.** In the **Input Map**, add `move_left`, `move_right`, `move_up`, `move_down` (WASD, arrow keys, left stick), as in Chapter 7. We do not need a start action: `ui_accept` already exists (Enter, Space, and the gamepad's bottom button).
3. **Layer names.** In **Layer Names > 2D Physics** name layer 1 `player`, layer 2 `coins`, layer 3 `hazards`.

The collision plan

Object	Layer (I am)	Mask (I notice)
Player	1 (player)	2 and 3 (coins, hazards) = 6
Coin	2 (coins)	nothing
Hazard	3 (hazards)	nothing

Only the player looks; coins and hazards are passive. Because the player's mask contains the layer of each, the player's `area_entered` signal fires when it overlaps either one. In numbers: the layer values are 1, 2, and 4, and the player's mask value is 2 + 4 = 6. (Coins and hazards each have layer 2 and 4, respectively, and mask 0.)

Folder layout

```
ch09-coin-dash/
├── project.godot
├── icon.svg
├── scenes/
│   ├── main.tscn
│   ├── player.tscn
│   ├── coin.tscn
│   ├── hazard.tscn
│   └── hud.tscn
└── scripts/
    ├── main.gd
    ├── player.gd
    ├── coin.gd
    ├── hazard.gd
    └── hud.gd
```

Naming each script after its scene and keeping them in two folders is a habit that pays off as projects grow.

A new line you will see: `class_name`

Several of our scripts start with a line like `class_name Player`. It gives the script a **global name**, so that other scripts can use it as a type (`var player: Player`). It also lets the editor suggest the script's functions when you write code that uses it. A `class_name` must be unique in the project, and it must **not** be the name of a built-in Godot class. `Player`, `Coin`, `Hazard`, and `Hud` are safe; `Timer`, `Area2D`, or `Label` would not be.

9.3 The Player scene

Scene tree:

```
Player (Area2D)              layer 1, mask 6    <- scripts/player.gd
└── CollisionShape2D         (CircleShape2D, radius 18)
```

The script moves Pip the way Chapter 7 did, but it also needs to switch itself on and off, because on the title screen and the game-over screen there is no Pip to steer or to hit.

```
class_name Player
extends Area2D
## player.gd -- attach to the Player node (Area2D) in scenes/player.tscn.
## Pip moves with the keyboard or gamepad and reports what he touches.

signal coin_collected
signal hit

@export var speed: float = 320.0
@export var radius: float = 18.0

var _bounds: Rect2 = Rect2()
var _facing: Vector2 = Vector2.RIGHT

func _ready() -> void:
    _bounds = get_viewport_rect().grow(-radius)
    area_entered.connect(_on_area_entered)
    stop()

# Called by Main when a round begins.
func start(where: Vector2) -> void:
    position = where
    show()
    set_process(true)
    set_deferred("monitoring", true)   # deferred: we may be inside a physics callback

# Called by Main when the round is over.
func stop() -> void:
    hide()
    set_process(false)
    set_deferred("monitoring", false)

func _process(delta: float) -> void:
    var direction: Vector2 = Input.get_vector(
            "move_left", "move_right", "move_up", "move_down")
    if direction != Vector2.ZERO:
        _facing = direction.normalized()
    position += direction * speed * delta
    position = position.clamp(_bounds.position, _bounds.end)
    queue_redraw()

func _on_area_entered(area: Area2D) -> void:
    if area.is_in_group("coins"):
        area.pickup()
        coin_collected.emit()
    elif area.is_in_group("hazards"):
        hit.emit()

func _draw() -> void:
    draw_circle(Vector2(3, 6), radius, Color(0, 0, 0, 0.25))
    draw_circle(Vector2.ZERO, radius, Color("f4b942"))
    for side in [-1.0, 1.0]:
        var eye: Vector2 = _facing * 7.0 + _facing.orthogonal() * 6.0 * side
        draw_circle(eye, 4.0, Color.WHITE)
        draw_circle(eye + _facing * 1.5, 2.0, Color("222222"))
```

Let us go through the new parts.

Two signals, declared at the top. `signal coin_collected` and `signal hit` are custom signals, announcements that this node can make. Pip does not decide what a collected coin is worth or what a hit means. He just announces the fact. Chapter 15 studies custom signals in depth; for now, a signal is a flag Pip can raise and `.emit()` is how he raises it.

Switching on and off. `start(where)` places Pip, makes him visible with `show()`, starts his `_process` with `set_process(true)`, and starts the area detecting overlaps (`monitoring`). `stop()` does the reverse. `_ready()` calls `stop()` so that Pip is hidden until Main starts a game. `hide()` and `show()` toggle the node's `visible` property; a hidden node draws nothing.

Why `set_deferred("monitoring", ...)`? `monitoring` is the property that makes an `Area2D` look for overlaps. `stop()` will be called from inside Main's response to a `hit` signal, which itself happens inside Pip's `area_entered` physics callback. Changing physics settings during a physics callback is not allowed (Chapter 8's warning). `set_deferred` waits until it is safe, at the end of the current step. We use it in `start()` too for consistency.

`area_entered`. When Pip's shape overlaps another area's shape, and Pip's mask includes that area's layer, Godot calls `_on_area_entered(area)` with the other area. We use groups to tell coins from slimes:

- `area.is_in_group("coins")` is true if the other area has been put into the group `coins` (in its scene file, or in the editor's **Groups** tab).
- For a coin, we call `area.pickup()` (the coin's own function) and raise `coin_collected`.
- For a hazard, we raise `hit`.

Because a coin is typed only as `Area2D` here, GDScript cannot check ahead of time that `pickup()` exists. It would only crash at run time if a non-coin sat in the `coins` group. That is a reason to keep group names reliable.

Movement is exactly Chapter 7: `get_vector`, multiply by `speed` and `delta`, clamp to the window.

> **Note:** The `_draw()` function repeats the eyes trick from Chapter 7. If you prefer, give Pip a `Sprite2D` and the `icon.svg` instead; the rest of the game does not change.

9.4 The Coin scene and spawning

Scene tree:

```
Coin (Area2D)                  layer 2, mask 0, group "coins"   <- scripts/coin.gd
└── CollisionShape2D           (CircleShape2D, radius 12)
```

```gdscript
class_name Coin
extends Area2D
## coin.gd -- attach to the Coin node (Area2D) in scenes/coin.tscn.
## A coin sits still, pulses a little, and disappears when picked up.

var _age: float = 0.0
var _picked: bool = false

func _ready() -> void:
    _age = randf() * TAU            # so coins do not all pulse in step

func _process(delta: float) -> void:
    _age += delta * 4.0
    var s: float = 1.0 + 0.12 * sin(_age)
    scale = Vector2(s, s)

func pickup() -> void:
    if _picked:
        return                      # never count the same coin twice
    _picked = true
    queue_free()

func _draw() -> void:
```

```
    draw_circle(Vector2.ZERO, 12.0, Color("ffd54a"))
    draw_arc(Vector2.ZERO, 12.0, 0.0, TAU, 24, Color("b8860b"), 2.0)
    draw_line(Vector2(0, -6), Vector2(0, 6), Color("b8860b"), 2.0)
```

A few ideas to notice:

- **A little life.** Static coins look dead. We keep a running timer `_age`, and each frame set the coin's `scale` to `1 + 0.12 * sin(_age)`. The `sin` function swings smoothly between -1 and 1, so the coin grows and shrinks by 12 percent. This is a taste of animation; Chapter 10 does it properly with tweens. The line `_age = randf() * TAU` gives each coin a different starting point: `randf()` returns a random decimal between 0 and 1, and `TAU` is a full circle in radians.
- **A guard against double counting.** `pickup()` checks `_picked` first. `queue_free()` does not delete the coin instantly; it schedules deletion at the end of the frame, and in the meantime the coin still exists. If two overlaps were reported in that window, the score would go up twice. The flag makes `pickup()` safe to call more than once.
- **The coin knows nothing about the score.** It disappears. Pip announces the pick-up; Main counts it.

The collision shape for a pulsing coin

Because the coin scales, its collision shape scales too (it is a child). That is fine here; the shape grows by at most 12 percent. (Earlier we warned against scaling shapes because of odd physics for solid bodies. For a small overlap-only area it is harmless.)

Spawning coins from code

The big new skill of this chapter is making things appear while the game runs. In Main we will write:

```
const CoinScene: PackedScene = preload("res://scenes/coin.tscn")
```

`preload()` reads a scene file when the script loads and gives you a **`PackedScene`**: a saved blueprint of a scene. By convention, constants that hold scenes begin with a capital letter, like a class. (`const` means the value never changes.)

To make a copy:

```
var thing: Node2D = CoinScene.instantiate()   # build a new copy from the blueprint
thing.position = Vector2(300, 200)            # set it up BEFORE adding it
$Coins.add_child(thing)                       # now it becomes part of the game
```

`instantiate()` builds the whole scene (a coin and its shape and script) in memory. Nothing appears until `add_child()` puts it into the scene tree, and only then do its `_ready()` and `_process()` start. It is good practice to set properties such as `position` before `add_child()`.

This is exactly what the Instantiate Child Scene button does in the editor, but now your *code* decides when and where.

Random positions

Coins should appear at random places, but not just anywhere. Figure 9.3 shows the two rules we use.

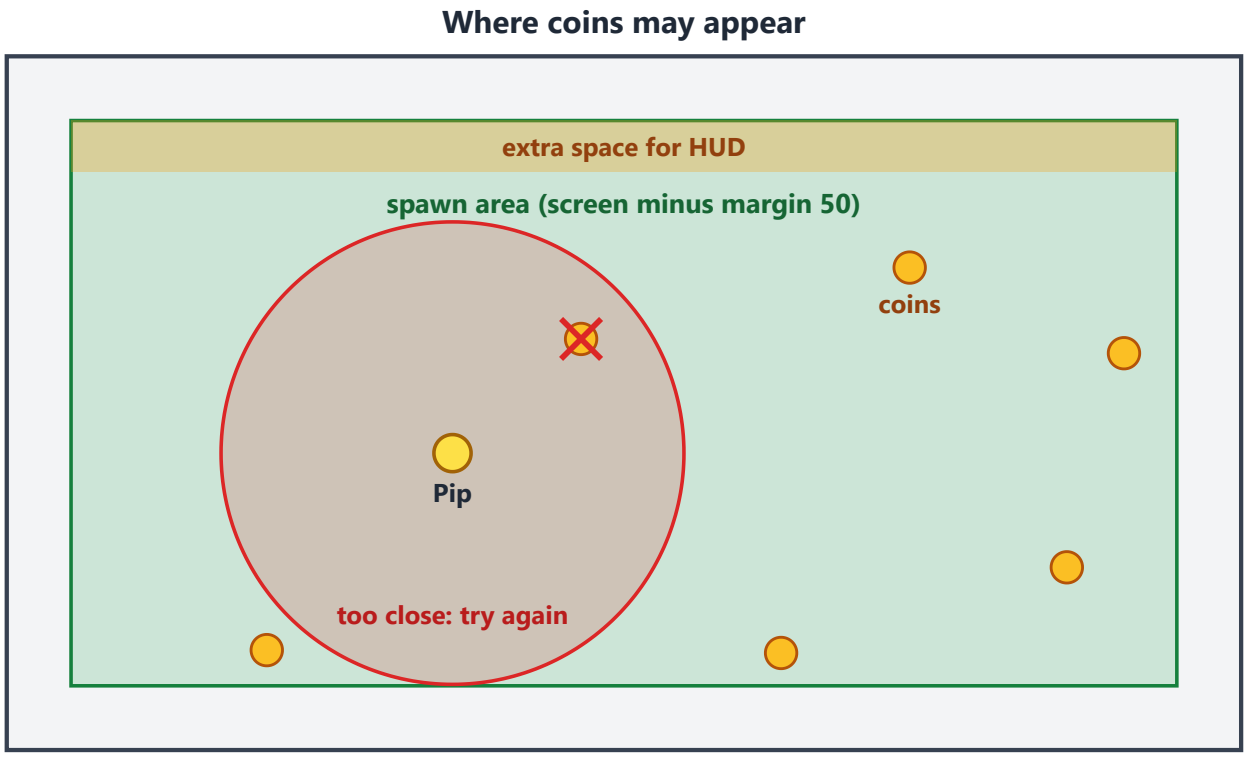


Figure 9.3: A coin may appear anywhere inside the green rectangle (the screen shrunk by a margin) but not inside the red circle around Pip. We try up to 30 random points and take the first one that works.

1. **Stay away from the edges**, so no coin is half hidden or impossible to reach.
2. **Stay away from Pip**, so a new level does not start with coins already on top of him.

```
const MARGIN: float = 50.0          # keep spawns away from the screen edge
const SAFE_DISTANCE: float = 180.0  # keep spawns away from Pip

# Pick a random point that is not too close to Pip.
func _random_spot() -> Vector2:
    var size: Vector2 = get_viewport_rect().size
    var spot: Vector2 = Vector2.ZERO
    for attempt in 30:
        spot = Vector2(
                randf_range(MARGIN, size.x - MARGIN),
                randf_range(MARGIN + 40.0, size.y - MARGIN))
        if spot.distance_to(player.position) >= SAFE_DISTANCE:
            break
    return spot
```

`randf_range(a, b)` returns a random decimal between `a` and `b`. The top margin is bigger by 40 pixels so that coins do not hide under the HUD text. The loop tries up to 30 times and uses `break` to leave the loop as soon as a good spot is found. If, by pure bad luck, all 30 tries fail, the last random spot is used anyway; with a 960 × 540 field and a 180-pixel circle, that practically never happens. A simple method that nearly always works is better than a complicated one that always does.

Why `randomize()`? In Godot 4 the random number generator is seeded randomly at start-up already, so calling `randomize()` is not required. We still call it once in `_ready()` as a harmless reminder that random numbers need a seed, and it makes the intent obvious to readers.

9.5 The Hazard scene: a bouncing slime

A game with only coins is a to-do list. A slime adds risk. Our slime is deliberately simple: it picks a random direction and drifts in a straight line, bouncing off the screen edges.

Scene tree:

```
Hazard (Area2D)               layer 3, mask 0, group "hazards"   <- scripts/hazard.gd
└── CollisionShape2D          (CircleShape2D, radius 17)
```

```gdscript
class_name Hazard
extends Area2D
## hazard.gd -- attach to the Hazard node (Area2D) in scenes/hazard.tscn.
## A slime that drifts in a straight line and bounces off the screen edges.

@export var speed: float = 70.0
@export var radius: float = 20.0

var _velocity: Vector2 = Vector2.ZERO
var _bounds: Rect2 = Rect2()

func _ready() -> void:
    _bounds = get_viewport_rect().grow(-radius)
    _velocity = Vector2.from_angle(randf() * TAU) * speed

func _process(delta: float) -> void:
    position += _velocity * delta
    if position.x < _bounds.position.x or position.x > _bounds.end.x:
        _velocity.x = -_velocity.x
    if position.y < _bounds.position.y or position.y > _bounds.end.y:
        _velocity.y = -_velocity.y
    position = position.clamp(_bounds.position, _bounds.end)

func _draw() -> void:
    draw_circle(Vector2.ZERO, radius, Color("5fbf4a"))
    draw_circle(Vector2(-7, -4), 5.0, Color.WHITE)
    draw_circle(Vector2(7, -4), 5.0, Color.WHITE)
    draw_circle(Vector2(-7, -3), 2.5, Color("222222"))
    draw_circle(Vector2(7, -3), 2.5, Color("222222"))
```

The new function is `Vector2.from_angle(angle)`, which builds a direction of length 1 from an angle in radians. A random angle between 0 and `TAU` is a random direction; multiplying by `speed` makes it a velocity.

Bouncing is one of the nicest tricks in game programming, because it is so cheap. When the slime passes the left or right edge, we flip the sign of the x part of its velocity (it now moves the other way horizontally). Passing the top or bottom flips the y part. A mirror reflection is just a sign change. The final `clamp` makes sure that a slime that stepped outside by a pixel is placed back at the border.

Notice that Pip's `_on_area_entered` only reports overlaps while Pip is monitoring. The slime moves on its own; the physics engine notices that its shape now overlaps Pip's, and tells Pip.

9.6 The HUD: Score, level, time, and messages

A **HUD** shows information that is not part of the world. Its natural home is a `CanvasLayer`, which draws on top of the game and ignores any camera. We have no camera now, but we will add one in later games, and the HUD should not scroll away.

Scene tree:

```
HUD (CanvasLayer)              <- scripts/hud.gd
├── ScoreLabel (Label)         top-left,   "Score: 0"
├── LevelLabel (Label)         top-centre, "Level 1"
├── TimeLabel (Label)          top-right,  "Time: 20"
├── Message (Label)            centre, big, "COIN DASH"
└── StartButton (Button)       below the message, "Start"
```

User interface nodes (the `Control` family: `Label`, `Button`, and others) are covered fully in Chapter 14. Here are the three things you need to know now.

1. **A `Label` shows text.** Set its `text` property in code or in the Inspector. To change its size, open **Theme Overrides > Font Sizes > Font Size** in the Inspector. We use 28 for the numbers and 36 for the message.
2. **Anchors pin a Control to part of the screen.** Select a Control and use the **Layout** menu in the toolbar above the viewport. Choose **Top Right** for `TimeLabel`, **Top Center** (or `Top Wide`) for `LevelLabel`, and **Center** for `Message` and `StartButton`. Then drag them to size. Anchors keep the label in its corner if the window changes size. Set `TimeLabel`'s **Horizontal Alignment** to `Right`, and `LevelLabel`'s and `Message`'s to `Center`.
3. **A `Button` sends a `pressed` signal** when clicked, or when you press Space or Enter while it has focus.

```gdscript
class_name Hud
extends CanvasLayer
## hud.gd -- attach to the HUD node (CanvasLayer) in scenes/hud.tscn.
## Shows score, level, time and the title / game-over message. It only
## displays things; Main decides what is true.

signal start_requested

@onready var score_label: Label = $ScoreLabel
@onready var level_label: Label = $LevelLabel
@onready var time_label: Label = $TimeLabel
@onready var message: Label = $Message
@onready var start_button: Button = $StartButton

func _ready() -> void:
    start_button.pressed.connect(func() -> void: start_requested.emit())

func update_score(value: int) -> void:
    score_label.text = "Score: %d" % value

func update_level(value: int) -> void:
    level_label.text = "Level %d" % value

func update_time(value: int) -> void:
    time_label.text = "Time: %d" % value

func show_title() -> void:
    message.text = "COIN DASH\nGrab every coin. Dodge the slimes!"
    start_button.text = "Start"
    _show_menu(true)

func show_game_over(score: int, best: int) -> void:
    message.text = "GAME OVER\nScore: %d    Best: %d" % [score, best]
    start_button.text = "Play again"
    _show_menu(true)

func hide_menu() -> void:
    _show_menu(false)

func _show_menu(visible_now: bool) -> void:
    message.visible = visible_now
    start_button.visible = visible_now
    if visible_now:
        start_button.grab_focus()
```

Points worth a second look:

- **The HUD is dumb on purpose.** It has functions such as `update_score(value)` that just put a number on screen. It never decides anything. Main calls these functions; the HUD only raises one signal, `start_requested`, when the button is pressed. This is "signals up, calls down" again.
- `func() -> void: start_requested.emit()` is an anonymous function, or **lambda**: a tiny function with no name, written on the spot. It is handy for a one-line reaction. Here the button's `pressed` signal is connected to it so that pressing the button raises our own `start_requested` signal. Main does not need to know that a button exists, only that someone asked to start.
- `"Score: %d" % value` fills the whole-number placeholder `%d` with `value`.
- `\n` **inside a string** is a line break, so the message has two lines.
- `grab_focus()` gives the button keyboard focus, so Space or Enter presses it immediately without clicking. A gamepad's bottom button works too.

9.7 Main: the rules and the states

Main is the director. It owns the score, the timer, the level, and the current state. It is the longest script in the project, but each function is short.

Scene tree:

```
Main (Node2D)                  <- scripts/main.gd
├── Background (ColorRect)     size 960 x 540, dark blue
├── Coins (Node2D)             an empty container for spawned coins
├── Hazards (Node2D)           an empty container for spawned slimes
├── Player (instance of player.tscn)
├── GameTimer (Timer)          wait time 1 second, not one-shot, no autostart
└── HUD (instance of hud.tscn)
```

The empty `Coins` and `Hazards` nodes are *containers*: spawning goes into them, and clearing a level is just "delete all their children". Without containers the coins would be scattered among the other children of Main.

The order matters for drawing: nodes lower in the list draw on top. The HUD is last, so it is on top.

States with an enum

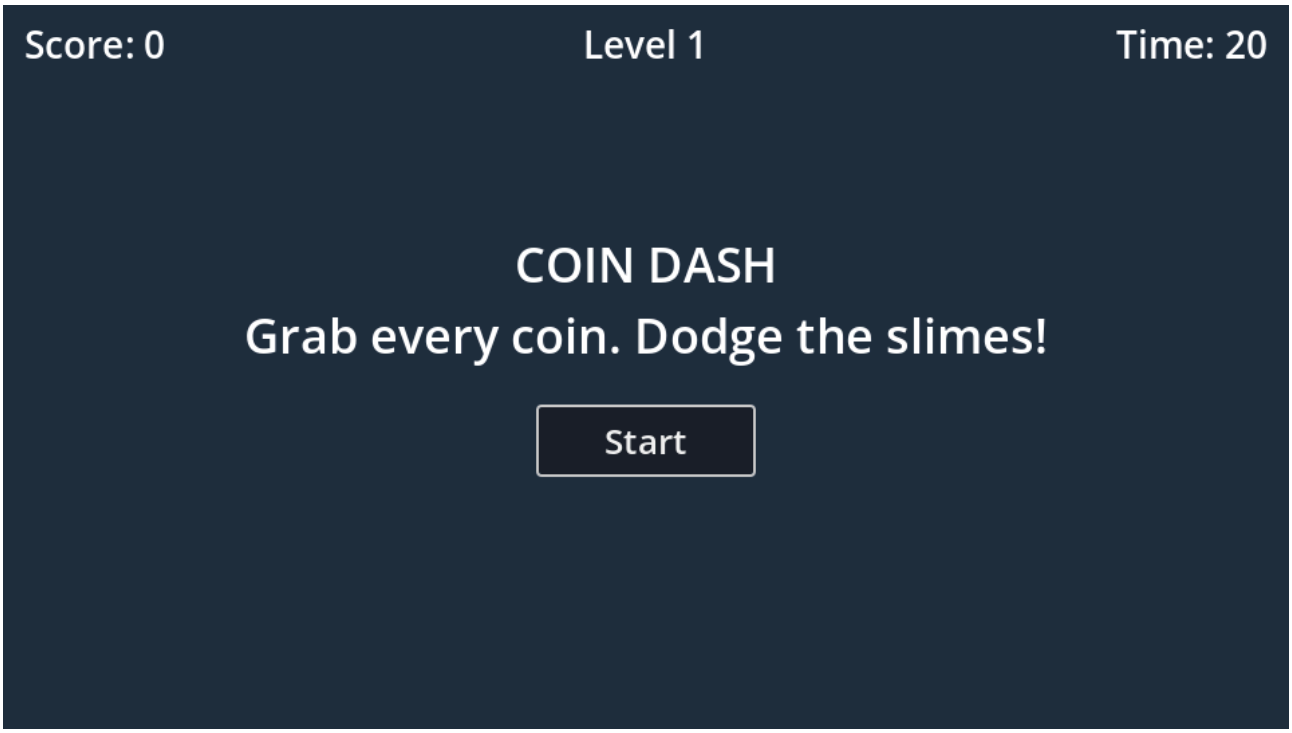


Figure 9.4: The title state. Nothing moves, Pip is hidden, and the Start button has focus.

A game that is "sometimes showing a menu and sometimes running" needs to remember which. We could use a boolean `is_playing`, but we have three states, so an **enum** reads better:

```
enum State { TITLE, PLAYING, GAME_OVER }

var state: State = State.TITLE
```

`State.TITLE`, `State.PLAYING`, and `State.GAME_OVER` are now valid values, with names that explain themselves. Compare `if state == State.PLAYING:` with `if state == 1:`. The first needs no comment.

Checks against the state guard our functions against being called at the wrong time. For example `start_game()` does nothing if we are already playing, and `game_over()` does nothing unless we are playing. Without such guards, a slime and a timer running out in the same moment would both end the game, and the best score or the game-over message could be written twice.

The complete script

```
extends Node2D
## main.gd -- attach to the Main node (Node2D) in scenes/main.tscn.
## Runs the whole game: states, spawning, score, timer, game over.

enum State { TITLE, PLAYING, GAME_OVER }

const CoinScene: PackedScene = preload("res://scenes/coin.tscn")
const HazardScene: PackedScene = preload("res://scenes/hazard.tscn")

const START_TIME: int = 20        # seconds on the clock at the start
const BONUS_TIME: int = 5         # seconds added for each cleared level
const COIN_POINTS: int = 10
const MARGIN: float = 50.0        # keep spawns away from the screen edge
const SAFE_DISTANCE: float = 180.0  # keep spawns away from Pip

@onready var player: Player = $Player
```

```
@onready var coins: Node2D = $Coins
@onready var hazards: Node2D = $Hazards
@onready var game_timer: Timer = $GameTimer
@onready var hud: Hud = $HUD

var state: State = State.TITLE
var score: int = 0
var best: int = 0
var level: int = 1
var time_left: int = 0
var coins_left: int = 0

func _ready() -> void:
    randomize()
    player.coin_collected.connect(_on_coin_collected)
    player.hit.connect(_on_player_hit)
    game_timer.timeout.connect(_on_game_timer_timeout)
    hud.start_requested.connect(start_game)
    hud.update_score(0)
    hud.update_level(1)
    hud.update_time(START_TIME)
    hud.show_title()

func _unhandled_input(event: InputEvent) -> void:
    if state != State.PLAYING and event.is_action_pressed("ui_accept"):
        start_game()

func start_game() -> void:
    if state == State.PLAYING:
        return
    state = State.PLAYING
    score = 0
    level = 1
    time_left = START_TIME
    hud.update_score(score)
    hud.update_level(level)
    hud.update_time(time_left)
    hud.hide_menu()
    player.start(get_viewport_rect().size / 2.0)
    _start_level()
    game_timer.start()

func game_over() -> void:
    if state != State.PLAYING:
        return
    state = State.GAME_OVER
    game_timer.stop()
    player.stop()
    best = max(best, score)
    _clear_children(coins)
    _clear_children(hazards)
    hud.show_game_over(score, best)

func _start_level() -> void:
    _clear_children(coins)
    _clear_children(hazards)
    coins_left = 4 + level * 2
    for i in coins_left:
        _spawn(CoinScene, coins)
    for i in mini(level, 5):
        _spawn(HazardScene, hazards)

func _spawn(scene: PackedScene, parent: Node2D) -> void:
    var thing: Node2D = scene.instantiate()
    thing.position = _random_spot()
    parent.add_child(thing)

# Pick a random point that is not too close to Pip.
func _random_spot() -> Vector2:
    var size: Vector2 = get_viewport_rect().size
    var spot: Vector2 = Vector2.ZERO
    for attempt in 30:
```

```
        spot = Vector2(
                randf_range(MARGIN, size.x - MARGIN),
                randf_range(MARGIN + 40.0, size.y - MARGIN))
        if spot.distance_to(player.position) >= SAFE_DISTANCE:
            break
    return spot

func _clear_children(parent: Node) -> void:
    for child in parent.get_children():
        child.queue_free()

func _on_coin_collected() -> void:
    if state != State.PLAYING:
        return
    score += COIN_POINTS
    coins_left -= 1
    hud.update_score(score)
    if coins_left <= 0:
        level += 1
        time_left += BONUS_TIME
        hud.update_level(level)
        hud.update_time(time_left)
        # We are inside a physics callback, so build the next level a moment later.
        _start_level.call_deferred()

func _on_player_hit() -> void:
    game_over()

func _on_game_timer_timeout() -> void:
    time_left -= 1
    hud.update_time(time_left)
    if time_left <= 0:
        game_over()
```

Reading it top to bottom

Constants. All the numbers that define the game's feel sit at the top in capital letters: starting time, bonus time, points per coin. Changing the difficulty means changing one line, not hunting through code. These are *magic numbers* made visible.

`_ready()`. This is the wiring. Main connects the signals it cares about:

- `player.coin_collected` -> `_on_coin_collected`
- `player.hit` -> `_on_player_hit`
- `game_timer.timeout` -> `_on_game_timer_timeout`
- `hud.start_requested` -> `start_game`

then sets the HUD to its starting values and shows the title. All of Main's children are ready *before* Main's own `_ready()` runs, which is why Main can use them here.

`_unhandled_input()`. Besides clicking the button, the player may press Enter or Space (`ui_accept`) to start or restart. This only works when we are not already playing. If the button has focus it handles the press itself first, so `_unhandled_input` is a backup for when focus is elsewhere.

`start_game()` resets the numbers, hides the menu, places Pip in the middle, builds level 1, and starts the clock. `game_timer.start()` makes the `Timer` node count one second, send `timeout`, and repeat.

The countdown. `_on_game_timer_timeout()` runs once per second. It subtracts one second, updates the HUD, and ends the game at zero. We use a whole-number counter of seconds rather than reading the Timer's fractional time because it is simple, and the display jumps exactly once per second. (A smooth countdown bar would need `game_timer.time_left` and a `ProgressBar`; try it as an extension.)

`game_over()`. Records the best score, stops the clock, hides Pip, deletes the leftover coins and slimes, and shows the message with the final and best scores. Note `best = max(best, score)`: `max` returns the larger of two numbers.

`_start_level()`. Clears anything old, decides how many coins this level needs, and spawns that many: `4 + level * 2`, so level 1 has 6 coins, level 2 has 8, and so on. It spawns `mini(level, 5)` slimes: `mini` returns the smaller of two whole numbers, which caps the slimes at five, so the game never becomes impossible. The `for i in number:` loop runs that many times; we don't use `i` itself.

`_spawn()`. The recipe from section 9.4: `instantiate()`, set the position, `add_child()`. It is written once and used for both coins and slimes. Its parameters are the scene to copy and the container to put it in. This is **reuse by passing a different argument**, a core idea of functions from Chapter 4.

`_clear_children()`. Loops over the container's children and calls `queue_free()` on each.

`_on_coin_collected()`. Adds points, counts one coin off, updates the HUD, and if no coins remain, advances the level: one more level number, extra seconds, and fresh spawns.

Why `call_deferred`?

The last line of `_on_coin_collected()` is the one that matters most. When we first built this project and called `_start_level()` directly, Godot printed a long list of errors:

```
ERROR: Can't change this state while flushing queries. Use call_deferred() or set_deferred() to change
monitoring state instead.
```

Think about the chain of events. Pip's `area_entered` signal fired. That happens *inside* the physics step. It called `_on_coin_collected`, which called `_start_level`, which created new `Area2D` nodes and added them to the tree. Adding a new area means changing the physics world, which is forbidden while the physics step is still busy ("flushing queries").

The cure is `call_deferred`. Writing `_start_level.call_deferred()` asks Godot: "run this function when it is safe, at the end of the current frame." The effect for the player is invisible (it happens a fraction of a second later), and the errors vanish.

This pattern is a rule of thumb you will use again and again:

> **Whenever a collision signal needs to add or remove physics objects, or change `monitoring`, a shape, or a layer, do it deferred.**

We already did the same for `monitoring` in Pip's `start()` and `stop()`.

> **Why it works:** Physics objects interact in a carefully ordered step. Changing them midway would be like replacing a player's chair while they are still sitting in it. Deferring waits until everybody stands up.

`queue_free()` is already deferred by nature, which is why `_clear_children` does not need special care.

9.8 Playtesting, tuning, and testing without hands

You can now press **F5** and play. The first thing to do is *play*, many times, and watch yourself.

Playtesting questions

- Is the first level too easy or too hard? A typical first success takes about 12 to 15 seconds in level 1. If it feels trivial, shorten `START_TIME`; if it feels stressful, lengthen it.
- Is Pip too slow, so that a coin on the opposite side takes forever? Adjust `speed` in the Player's Inspector.
- Do slimes feel fair? A slime that starts close to a coin cluster feels cheap. The safe distance only keeps spawns away from Pip; you might also keep slimes away from the coins.
- Does the game give players a *reason to try again*? The best score on the game-over screen is a small version of that.

Tuning table

Constant	Where	Effect of increasing it
`START_TIME`	`main.gd`	More time at the beginning: easier
`BONUS_TIME`	`main.gd`	Bigger reward for clearing a level: easier
`COIN_POINTS`	`main.gd`	Higher scores (no effect on difficulty)
`speed` of Pip	Player Inspector	Faster Pip: easier
`speed` of Hazard	Hazard Inspector	Faster slimes: harder
`4 + level * 2`	`_start_level()`	More coins per level: longer levels
`mini(level, 5)`	`_start_level()`	More slimes allowed: harder

> **Try it:** Change one number at a time, play twice, and write down how it felt. Never change three things at once; you will not know which one made the difference.

Testing without hands

An automated check cannot press keys, but you *can* test the rules by driving the game from a small script. You can run the finished project that way, without a window, using Godot's command-line options:

```
godot --headless --path projects/ch09-coin-dash --import
godot --headless --path projects/ch09-coin-dash --quit-after 120
```

The first line lets Godot read the project and prepare its files. The second runs it for 120 frames with no window and quits; if the scripts have mistakes, they appear as errors in the output, and if there are none, the output is clean. In addition we ran a throw-away script that created the main scene, started a game, put Pip on top of each coin in turn, and checked that the score, level, bonus time, number of slimes, and game-over states were what the rules say. In an advanced chapter (22) you will learn how to write such tests yourself; for now remember the idea: *because the rules live in Main and the pieces talk through signals, they can be tested without anyone pressing a key.*

9.9 Where to go from here

Coin Dash is the seed of the finished game in Chapter 18, which adds a title scene, a pause menu, and a saved high score. Until then, you could try these extensions (several are in the exercises):

- A **sound** when you pick up a coin (Chapter 11).
- Smooth **animation** when coins appear and vanish (Chapter 10).
- A **power-up** that freezes the slimes for three seconds.

- Slimes that **chase** Pip slowly (Chapter 21).
- **Screen shake** when the game ends (Chapter 20).
- Free art: replace the drawn shapes with sprites from Kenney.nl, itch.io, or OpenGameArt. This is optional; the game is complete as it is.

Project 9: Build Coin Dash

Goal: Build the complete game from an empty project. Allow 60 to 90 minutes. If you get stuck on a step, open `projects/ch09-coin-dash/` and compare.

Part A: project setup

1. Create a new project named `ch09-coin-dash`.
2. Project Settings: viewport `960 × 540`, stretch mode `canvas_items`.
3. Input Map: `move_left`, `move_right`, `move_up`, `move_down`, bound to WASD, arrows, and the left stick.
4. Layer Names > 2D Physics: `1 = player`, `2 = coins`, `3 = hazards`.
5. Make folders `scenes` and `scripts`.

Part B: the small scenes (test each one alone with F6 if you wish)

6. **Coin.** New scene, root `Area2D` renamed `Coin`. Add a `CollisionShape2D` with a **CircleShape2D** (radius 12). In the Inspector set **Collision > Layer** to only box 2; **Mask** to none. In the **Groups** tab, add `coins`. Attach `scripts/coin.gd` (section 9.4). Save as `scenes/coin.tscn`.
7. **Hazard.** New scene, `Area2D` named `Hazard`, circle shape radius 17, **Layer** box 3, **Mask** none, group `hazards`. Attach `scripts/hazard.gd` (section 9.5). Save as `scenes/hazard.tscn`.
8. **Player.** New scene, `Area2D` named `Player`, circle shape radius 18. **Layer** box 1, **Mask** boxes 2 and 3. Attach `scripts/player.gd` (section 9.3). Save as `scenes/player.tscn`.
9. **HUD.** New scene, root `CanvasLayer` named `HUD`. Add the five children from the tree in section 9.6 with the right names (`ScoreLabel`, `LevelLabel`, `TimeLabel`, `Message`, `StartButton`), set their font sizes, anchors, and alignment. Attach `scripts/hud.gd`. Save as `scenes/hud.tscn`.

Part C: the main scene

10. New scene, root `Node2D` named `Main`. Add:
    - a `ColorRect` named `Background`, size `960 × 540`, colour dark blue (`#1f2e40`),
    - two empty `Node2D` children named `Coins` and `Hazards`,
    - an instance of `player.tscn` (name it `Player`),
    - a `Timer` named `GameTimer` (leave **Wait Time** at 1, **One Shot** off, **Autostart** off),
    - an instance of `hud.tscn` (name it `HUD`), as the last child.
11. Attach `scripts/main.gd` (section 9.7) to `Main`. Save as `scenes/main.tscn`.
12. **Project Settings > Application > Run > Main Scene:** set `scenes/main.tscn`.

Part D: play and test

13. Press **F5**. You should see the title with a Start button. Press Enter.
14. Run through this checklist:

- [ ] The title screen shows, Pip is hidden, and the Start button is focused (press Space or Enter to start).

- [ ] After starting: six coins and one slime are on screen; the HUD says `Score: 0`, `Level 1`, `Time: 20`.
- [ ] No coin and no slime starts within about 180 pixels of Pip.
- [ ] Collecting a coin raises the score by 10 and removes the coin.
- [ ] The time counts down one second at a time.
- [ ] Collecting all six coins starts Level 2: eight coins, two slimes, and the clock jumps up by 5.
- [ ] Touching a slime shows GAME OVER with the score and the best score.
- [ ] Letting the time run out also shows GAME OVER.
- [ ] Play again works and resets the score, level, and time but keeps the best score.
- [ ] The Output panel shows no red error messages during all of the above.

Expected result

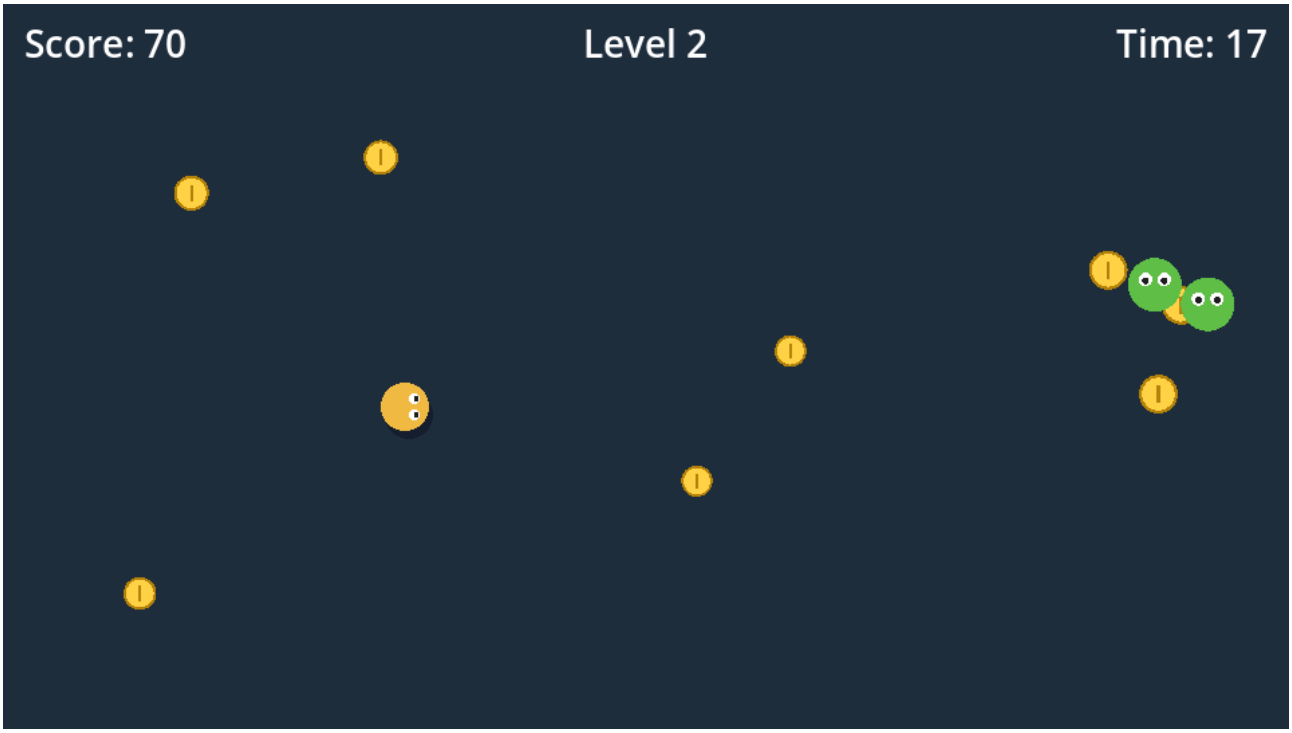


Figure 9.5: Level 2 of Coin Dash in play. Score, level, and time are at the top; coins pulse gently; a slime drifts across the field.

You can play a full round: pick coins before the clock runs out, avoid the slime, advance to harder levels, and see your score when the game ends.

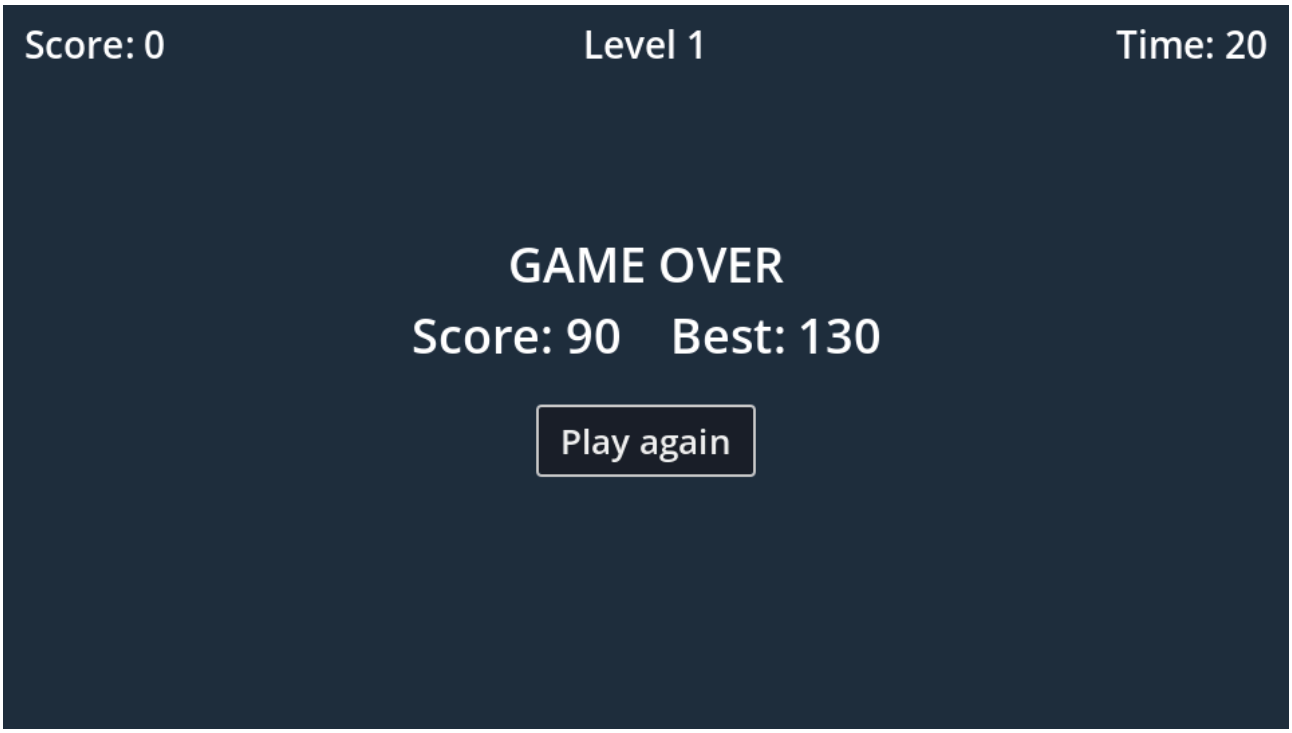


Figure 9.6: Game over. The message shows the final and best scores; the button restarts the game.

Chapter summary

- Plan first: list the rules, name the game states, divide the game into scenes, and decide how the scenes talk. **Signals go up; calls go down.**
- Small scenes (`Player`, `Coin`, `Hazard`, `HUD`) each do one job and know nothing about the others; `Main` connects them and owns the rules.
- `preload()` loads a scene as a `PackedScene`; `instantiate()` makes a copy; `add_child()` puts it in the game. Set properties such as `position` before adding.
- Random spawning uses `randf_range()`, margins from the screen edge, and a safe distance from the player.
- Use an `enum` and a `state` variable for the phases of the game, and guard functions so they run only in the right state.
- A `Timer` node with a `timeout` signal makes a countdown; a `CanvasLayer` with `Label` and `Button` nodes makes a HUD.
- Never add or change physics objects inside a collision callback. Use `call_deferred()` or `set_deferred()`.
- Playtest, change one number at a time, and keep the tuning constants together at the top of the script.

Review questions

1. Why is `Main` the only script that knows about the score, even though the coin is the thing that gets picked up?

2. **Multiple choice.** What does `preload("res://scenes/coin.tscn")` return? a) A `Coin` node already in the game b) A `PackedScene`, a blueprint you can instantiate c) A string with the file path d) The coin's script

3. In what order must you call `instantiate()`, `add_child()`, and set `position`, and why is it a good idea?

4. **True or false.** Calling `queue_free()` on a node deletes it immediately, in the middle of the current line of code.
5. What is the purpose of the `_picked` flag in `coin.gd`?
6. Why does `Player.stop()` use `set_deferred("monitoring", false)` rather than `monitoring = false`?

7. **Multiple choice.** Which statement about the `State` enum is true? a) `State.PLAYING` is a text string "PLAYING". b) It lets you name the phases of the game and compare against readable names. c) You can only have two states in an enum. d) An enum makes the game run faster.

8. At level 3, how many coins and how many slimes are spawned? Show the calculation from `_start_level()`.

9. The player's mask is 6 and the coin's layer is 2. Does the player's `area_entered` fire when Pip overlaps a coin? Explain with the bit values.
10. Explain "signals go up, function calls go down" using the HUD and Main as your example.

Exercises

1. **Practice (easy): Tuning.** Without adding any code, change `START_TIME`, `BONUS_TIME`, and Pip's speed until you can reach level 3 about half of the time. Write down your three numbers.
2. **Practice (easy): Bigger rewards.** Make each coin worth `10 * level` points, so later levels are worth more. Which line changes? Test that level 2 coins give 20 points.
3. **Apply (medium): A time bonus coin.** Create a second coin scene, `time_coin.tscn`, drawn in blue. Picking it up gives +3 seconds instead of points, and it does not count toward clearing the level. You will need a `kind` variable on the coin, a way for Pip's signal to carry it (use a signal with a parameter: `signal coin_collected(kind: String)`), and a change to `Main._on_coin_collected`.
4. **Apply (medium): Best score on the HUD.** Show `Best: N` permanently in the top-left under the score. Which scripts change? (Hint: HUD needs a new label and a function; Main calls it whenever `best` changes.)
5. **Challenge (hard): A pause key.** Add an input action `pause` (the P key). While playing, pressing it should freeze everything: set `get_tree().paused = true`, and show "PAUSED". Pressing it again resumes. You must set the `process_mode` of the node that listens for the key to `Always` (in the Inspector under **Node > Process > Mode**), or it will be paused too and never see the key. Check that the timer also stops.

Common mistakes

- **Symptom:** The score goes up by 20 for one coin. -> **Cause:** The same coin was reported twice (no guard against double pick-up). -> **Fix:** Keep a `_picked` flag, or check `is_queued_for_deletion()`.
- **Symptom:** The error "Can't change this state while flushing queries." fills the Output panel. -> **Cause:** Physics objects were added or changed inside a collision callback. -> **Fix:** Defer: `_start_level.call_deferred()` or `set_deferred("monitoring", true)`.
- **Symptom:** Pip touches coins and nothing happens. -> **Cause:** The player's mask does not include the coin's layer, the coin is not in the `coins` group, or Pip's `monitoring` is off. -> **Fix:** Check both layer and mask, the group name spelling, and that `start()` ran.
- **Symptom:** The Start button or HUD does nothing, or Godot says a node was not found. -> **Cause:** A `$NodeName` path in a script does not match the node's actual name in the scene tree (capitals matter). -> **Fix:** Rename the node to match, or fix the path, and watch for yellow warnings in the Output panel.

- **Symptom:** The game never restarts after Game Over, or the old coins remain. -> **Cause:** The old state was not reset (score, time, level) or leftover nodes were not cleared. -> **Fix:** Put all resetting in `start_game()` and `_start_level()`, and clear containers with `queue_free()`.
- **Symptom:** The text is hidden behind the game or too small. -> **Cause:** The HUD is not the last child of Main, or the font size was never overridden. -> **Fix:** Move the HUD to the bottom of the list (or put it in a `CanvasLayer`) and set font sizes under Theme Overrides.

Further reading

- Your first 2D game (the official tutorial that inspired this project): https://docs.godotengine.org/en/stable/getting_started/first_2d_game/index.html
- Instancing with code: https://docs.godotengine.org/en/stable/tutorials/scripting/instancing_with_signals.html
- Using `Timer`: https://docs.godotengine.org/en/stable/classes/class_timer.html
- `CanvasLayer`: https://docs.godotengine.org/en/stable/classes/class_canvaslayer.html
- Groups: https://docs.godotengine.org/en/stable/tutorials/scripting/groups.html
- Using signals: https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

Chapter 10: Animation

Learning objectives

By the end of this chapter you will be able to:

- Explain how a game creates the illusion of movement from still pictures, and choose between `AnimatedSprite2D`, `AnimationPlayer` and a `Tween` for a given job.
- Cut a sprite sheet into frames and play them with an `AnimatedSprite2D` and a `SpriteFrames` resource.
- Build an animation in the `AnimationPlayer` with tracks and keyframes, and trigger it from a script.
- Create a `Tween` in code that moves, scales and fades nodes, with easing, chaining, parallel steps and callbacks.
- Apply two simple animation principles (anticipation and squash and stretch) to make a jump feel alive.
- Keep animation separate from game logic so that your scripts stay short and your characters stay easy to change.

Key terms

- **Frame**: one still picture in a sequence. Showing frames quickly one after another looks like motion.
- **Frames per second (FPS)**: how many animation frames are shown each second. Not the same as the game's frame rate.
- **Sprite sheet**: one image that holds many frames arranged in a grid or a row.
- **SpriteFrames**: a resource that stores named animations (such as "idle" and "run"), each a list of frames, for an `AnimatedSprite2D`.
- **AnimatedSprite2D**: a node that plays the animations stored in a `SpriteFrames` resource.
- **AnimationPlayer**: a node that stores animations that change any property of any node over time.
- **Track**: one row in an animation that controls one property (for example `modulate`) of one node.
- **Keyframe (key)**: a value placed on a track at a specific time. The engine fills in the values between keys.
- **Interpolation**: calculating the in-between values between two keys or two end points.
- **Tween**: a small helper object, created in code, that changes a property from its current value to a target value over a set time.
- **Easing**: how the speed changes during a movement (slow start, slow end, overshoot, bounce).
- **Squash and stretch**: an animation principle where an object flattens or lengthens to show weight and speed.

10.1 How animation works

Still pictures, fast

A film is a ribbon of still photographs. When the pictures change quickly enough, about 12 times a second or faster, your brain stops seeing separate pictures and sees one thing moving. This effect has a name, persistence of vision, and every game relies on it.

Your game already does this at two levels:

1. The screen redraws the whole scene about 60 times each second. When you move a node a tiny bit each frame (as you did in Chapter 7 with `delta`), the node appears to glide.

2. A character can also change its *picture*. A walking character shows leg-up, legs-together, other-leg-up, legs-together, and then starts over. That is a **frame animation**, and it can run slower than the screen's 60 redraws. Hand-drawn walk cycles are often 8 to 12 pictures per second.

These two levels are independent. A game can draw 60 screens a second while the hero's walk cycle plays at 10 pictures a second. In this chapter we call the second number **FPS** (frames per second) *of the animation*, and we call the first number the *game's frame rate*.

Three tools, three jobs

Godot gives you three main ways to animate. Learn what each is *for* and you will rarely pick the wrong one.

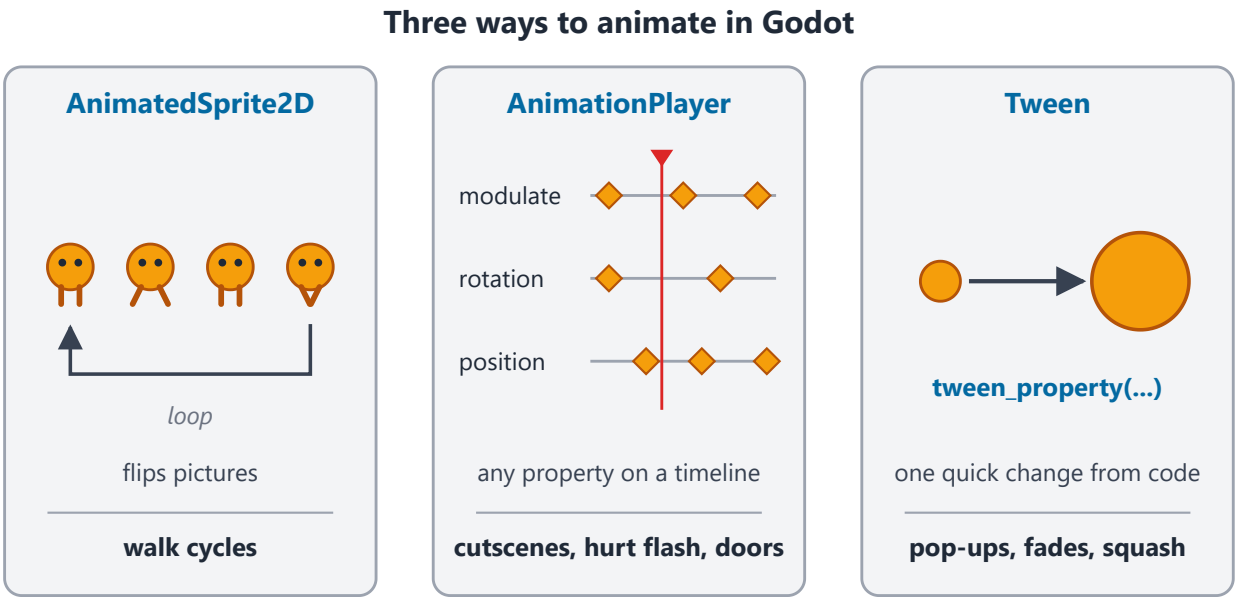


Figure 10.1: AnimatedSprite2D flips pictures, AnimationPlayer changes any property on a timeline, and a Tween makes one quick change from code.

Tool	What it does	Best for
`AnimatedSprite2D`	Shows a list of pictures one after another	Walk, run, idle, attack cycles drawn frame by frame
`AnimationPlayer`	Changes any property of any node along a timeline you design with keyframes	Cutscenes, doors opening, a hurt flash, menus sliding in, anything with several parts moving together
`Tween`	Changes one property from where it is now to a new value, from code, then disappears	Quick effects: pop in, fade out, squash on landing, a button growing when hovered

A rule of thumb that works well:

- If the animation is **a set of drawn pictures**, use `AnimatedSprite2D`.
- If the animation is **designed once and reused many times** with several properties together, use `AnimationPlayer`.
- If the animation is **small, one-off, and the target value is only known while the game runs** (for example "slide to wherever the player clicked"), use a `Tween`.

You will often use all three in one character, and the project at the end of this chapter does exactly that.

> **Note:** A fourth node, `AnimationTree`, blends and switches between many `AnimationPlayer` animations using a graph. It is powerful and mostly used for 3D characters. You do not need it yet, and this book does not use it.

10.2 Frame animation with AnimatedSprite2D

Sprite sheets

Artists rarely save each frame as a separate file. They pack all frames of a character into one image, a **sprite sheet**. Every frame has the same size, so the sheet is a tidy grid. Godot's job is to cut the sheet into frames and show them in order.

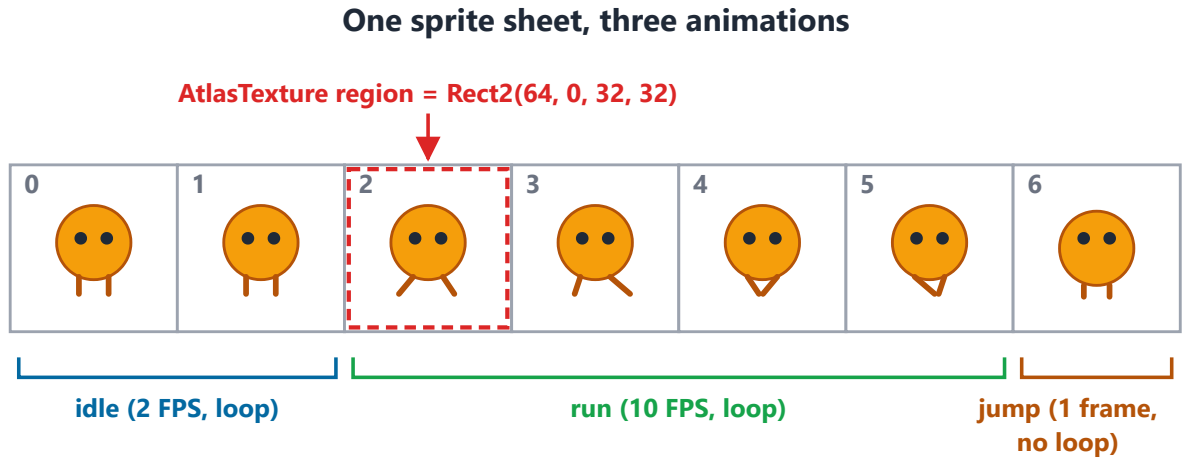


Figure 10.2: One image, seven equal cells. Cell 0 and 1 form "idle", cells 2 to 5 form "run", cell 6 is "jump".

Rule 9 of this book says our projects need no outside art. So Pip's sprite sheet is drawn **by code**: a function paints seven 32 by 32 pixel pictures side by side into one `Image` object. The code is in the project at the end of the chapter. What matters for now is that the result is exactly like a sheet you would download or draw yourself: a single image with equal-sized cells.

> **Note:** If you want real art, free sprite sheets are easy to find on Kenney.nl, OpenGameArt and itch.io. Everything in this section works the same with a downloaded PNG. Appendix F lists places to look.

The editor way: SpriteFrames

Here is the standard workflow with an image file. Assume you have `res://pip_sheet.png`, seven frames in one row, each 32 by 32.

1. Add an `AnimatedSprite2D` node to your scene.
2. In the Inspector, click the empty **Sprite Frames** property and choose **New SpriteFrames**.
3. Click the new `SpriteFrames` resource. The **SpriteFrames panel** opens at the bottom of the editor.
4. In the left list is one animation called `default`. Rename it (double-click) to `idle`. Use the **Add Animation** button to make `run` and `jump`.
5. Select `idle`. Click the **Add frames from sprite sheet** button. Pick your PNG. A window shows the image with a grid.

6. Set **Horizontal** to 7 and **Vertical** to 1. Click the cells you want (here cells 0 and 1), then click **Add 2 Frame(s)**.
7. Set the **FPS** box (speed) and the **Loop** toggle for each animation. Idle is calm, so try 2 FPS and loop on. Run might use 10 FPS and loop on. Jump has one frame, so loop off.
8. Back in the Inspector, set **Animation** to `idle`, and click **Autoplay on Load** (the small play icon in the SpriteFrames panel) if you want it to start by itself.

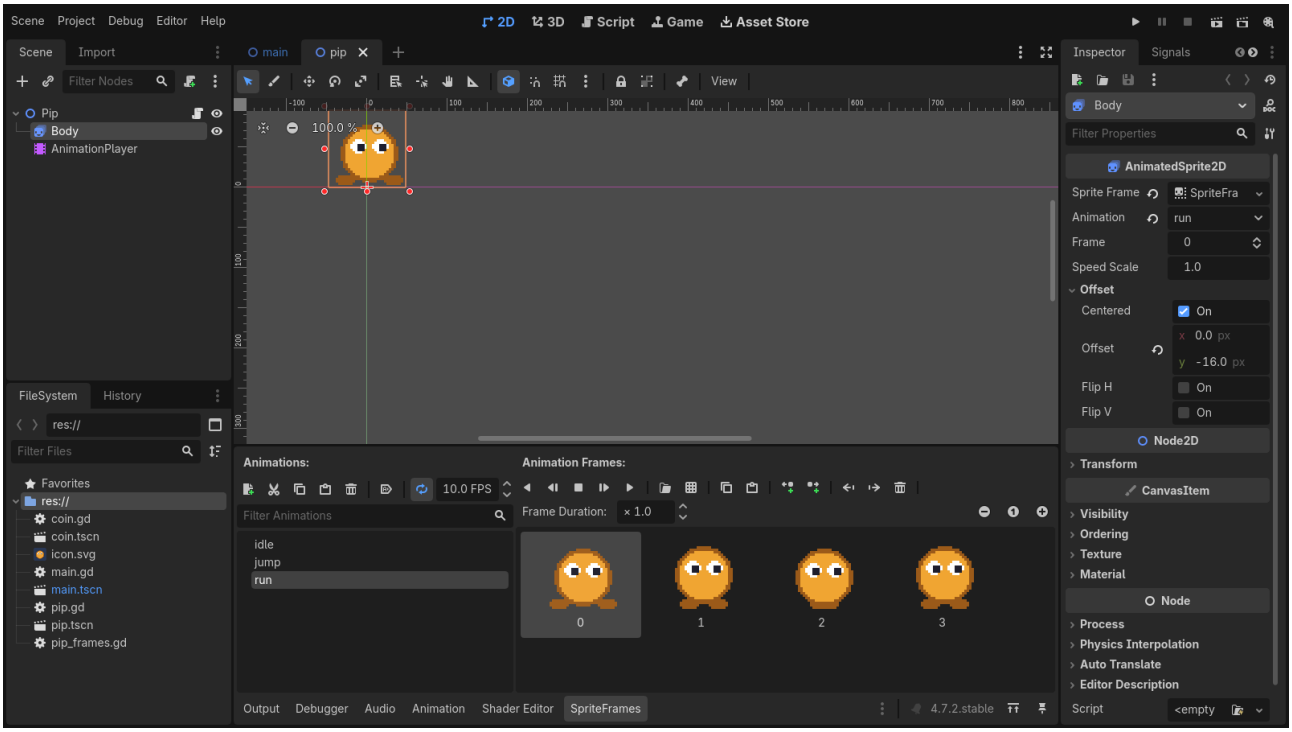


Figure 10.3: The SpriteFrames panel. Animations on the left, frames on the right, speed and loop above.

Playing animations from a script

An `AnimatedSprite2D` is controlled from GDScript with a few methods and properties:

Member	Meaning
`play("run")`	Start the animation called `run`
`pause()` / `stop()`	Pause (keep the frame) or stop (go back to frame 0)
`animation`	Name of the animation that is current
`frame`	Number of the frame being shown (0 is the first)
`speed_scale`	Multiplies the speed: 2.0 is twice as fast, 0.5 half
`flip_h` / `flip_v`	Mirror the picture sideways or upside down
`animation_finished` (signal)	Sent when a **non-looping** animation reaches its last frame
`frame_changed` (signal)	Sent every time the shown frame changes

Notice `flip_h`. Artists draw a character once, facing right. When the player walks left you do not need another set of frames. You set `flip_h = true` and the same pictures show mirrored.

Here is a **fragment** that you would place inside a script on a character whose child is an `AnimatedSprite2D` named `Body`:

```gdscript
@onready var body: AnimatedSprite2D = $Body

func _process(_delta: float) -> void:
    var dir: float = Input.get_axis("ui_left", "ui_right")
    if dir != 0.0:
        body.flip_h = dir < 0.0     # face the way we are walking
        body.play("run")
    else:
        body.play("idle")
```

Calling `play("run")` again while `run` is already playing does **not** restart it, so this is safe to call every frame. That is a handy property: you can describe what the character *should* be doing and let the sprite sort out the rest.

> **Why it works:** `body.flip_h = dir < 0.0` stores a true/false answer in the property. `dir < 0.0` is a comparison (Chapter 4), and a comparison is itself a value that is either `true` or `false`.

Reacting when an animation ends

For a non-looping animation such as an attack, you often want something to happen at the end. Connect the `animation_finished` signal (Chapter 5 introduced signals):

```gdscript
func _ready() -> void:
    body.animation_finished.connect(_on_body_animation_finished)

func attack() -> void:
    body.play("attack")

func _on_body_animation_finished() -> void:
    # This runs when a non-looping animation reaches its last frame.
    body.play("idle")
```

This is a **fragment**: it assumes `body` is declared as above and that the sprite has animations named `attack` and `idle`.

A simpler cousin: Sprite2D with hframes

If your sheet is a single row or grid and you only need one animation, you can skip `SpriteFrames`. A plain `Sprite2D` has properties **Hframes**, **Vframes** and **Frame**. Set Hframes to 7 and the sprite shows only one seventh of the image, and the property `frame` picks *which* seventh. Change `frame` over time (from a script or from an `AnimationPlayer`, as in section 10.3) and you have an animation.

> **Try it:** In a new scene, add a `Sprite2D`, drag `res://icon.svg` onto its Texture, and set Hframes to 2. You will see half of the icon. Change Frame between 0 and 1. Which half appears for each?

Building frames in code

You do not have to use the editor panel. Everything it does is available in code, and our project builds Pip's animations that way because there is no image file to import. A fragment to show the idea:

```gdscript
var frames := SpriteFrames.new()
frames.remove_animation("default")       # a new SpriteFrames comes with one animation, "default"
frames.add_animation("run")
frames.set_animation_speed("run", 10.0)  # FPS
frames.set_animation_loop("run", true)
```

```
frames.add_frame("run", some_texture)    # repeat once per picture
$Body.sprite_frames = frames
```

The pictures come from `AtlasTexture`, a resource that means "this region of that big image":

```
var piece := AtlasTexture.new()
piece.atlas = big_image_texture        # the whole sheet
piece.region = Rect2(64, 0, 32, 32)    # x, y, width, height of one cell
```

That is all slicing is: a rectangle cut out of the sheet. Cell number 2 starts at x = 2 times 32 = 64.

10.3 Animating any property with AnimationPlayer

`AnimatedSprite2D` can only change pictures. What if you want a door to swing open, a light to pulse, a character to flash red and wobble when hit, and a sound to play at the same moment? You need something that can change *any* property of *any* node, on a timeline you control. That is the `AnimationPlayer`.

The idea: tracks and keyframes

An animation is a set of **tracks**. Each track controls one property of one node. On a track you place **keyframes**: "at 0.0 seconds the value is X, at 0.3 seconds it is Y." Godot computes the values *between* the keys (that is **interpolation**) so you only place the important moments.

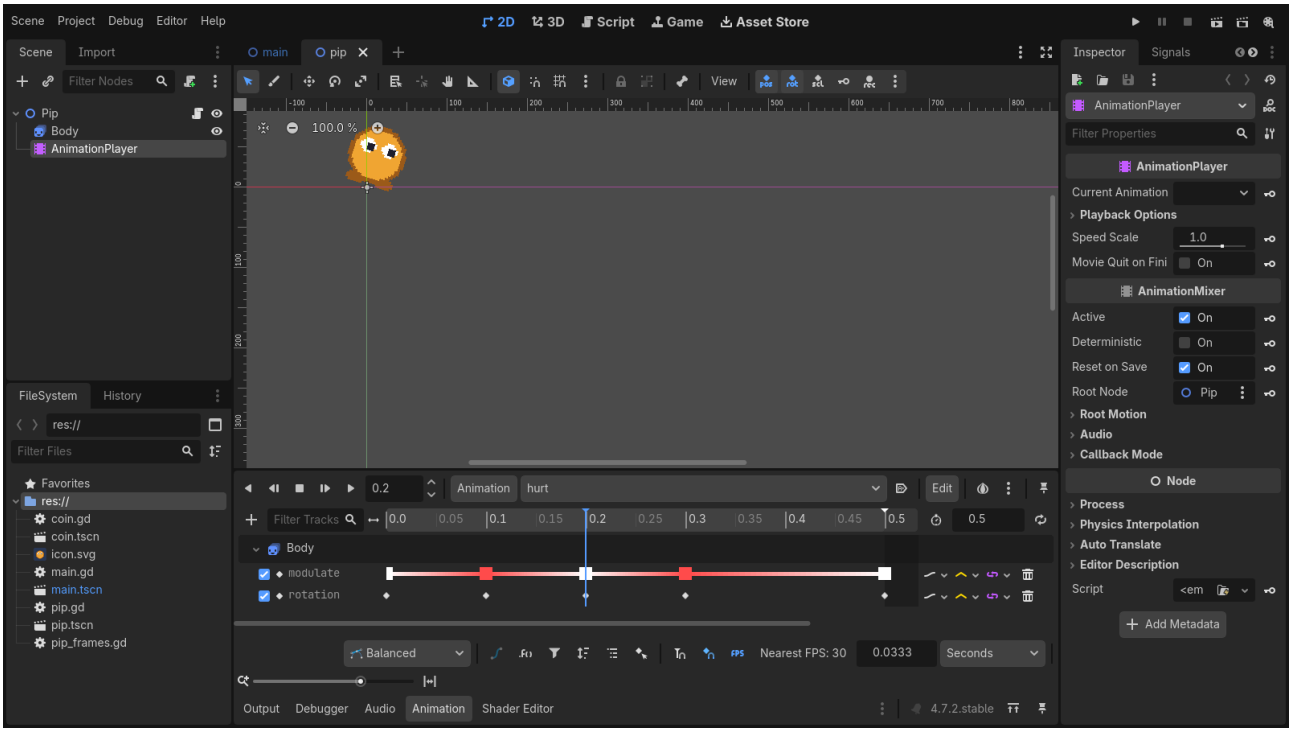


Figure 10.4: An animation called "hurt" with a `Body:modulate` track and a `Body:rotation` track. Each diamond is a keyframe.

Here is a two-key track in plain words: *rotation of Body: at 0.0 s it is 0, at 0.5 s it is 1.57 (a quarter turn, in radians).* Godot fills in every value between, so the node turns smoothly.

The editor way

1. Add an **AnimationPlayer** node. Put it next to the nodes it will animate. By default it controls nodes *relative to its parent*.

2. Select it. The **Animation panel** opens at the bottom.
3. Click **Animation**, then **New**. Name it `hurt`.
4. Set the length (the box with the clock icon, in seconds), for example 0.5.
5. Select the node you want to animate, for example `Body`. In the Inspector, find a property such as **Modulate** (under CanvasItem, Visibility) and click the small **key icon** to its left. Godot asks to create a track. Say **Create**.
6. Move the red time marker to a new time, change the property's value, and click the key icon again. A second keyframe appears.
7. Press the **Play** button in the panel to preview.

Two settings are worth knowing:

- **Loop**: the button with two arrows near the length box. Off: the animation plays once. On: it repeats. (There is also ping-pong, which plays forward then backward.)
- **Autoplay on Load**: the button that looks like a play triangle with an A. Use it for scenery that should move from the start.

Interpolation and update mode

Each track has two small settings you can change by clicking the icons at the right end of the track:

- **Interpolation** decides how Godot fills the gaps. *Nearest* jumps (no smoothing). *Linear* is a straight line from key to key. *Cubic* makes a smooth curve through the keys.
- **Update mode** (for property tracks) decides *when* the property is written. *Continuous* writes a new value every frame, so numbers and colors glide. *Discrete* writes only at the keyframes, which is correct for things that cannot blend, like a sprite's frame number or a boolean such as `visible`.

If you animate `frame` with Continuous mode, Godot may try to produce frame 2.4, which makes no sense. Use **Discrete** for frame numbers.

Other kinds of track

Besides property tracks, the *Add Track* menu offers:

- **Call Method Track**: at a given time, call a function on a node. Use it to spawn a dust puff at the instant a foot touches the ground, or to tell the game "the hit landed now."
- **Audio Playback Track**: start a sound at a given time (Chapter 11 introduces audio).
- **Animation Playback Track**: start another `AnimationPlayer`'s animation.
- **Bezier Curve Track**: a property animated with hand-shaped curves, for finer control.

The RESET animation

If you animate a property and then leave the animation half-finished in the editor, the node in your scene may be saved in a strange state (for example half-faded). Godot helps with a special animation named `RESET`. It stores the *normal* value of every property you animate. When you add your first track the editor offers to create it. Keep it: the editor uses it to restore the scene when you save or close the animation.

Controlling an AnimationPlayer from a script

```
@onready var anim: AnimationPlayer = $AnimationPlayer

func get_hurt() -> void:
    anim.play("hurt")
```

The members you will use most:

Member	Meaning
`play("name")`	Start an animation. To force it to begin again from the start, call `stop()` first.
`queue("name")`	Play this one when the current one finishes.
`stop()` / `pause()`	Stop (back to start) or freeze in place.
`is_playing()`	`true` while something is playing.
`current_animation`	The name of the playing animation, or an empty string.
`speed_scale`	Playback speed multiplier. 2.0 is double speed. A negative value plays backwards.
`seek(seconds)`	Jump to a time inside the animation.
`animation_finished(anim_name)` (signal)	Sent when an animation reaches its end. The name is passed in.

```
func _ready() -> void:
    anim.animation_finished.connect(_on_animation_finished)

func _on_animation_finished(anim_name: StringName) -> void:
    if anim_name == &"hurt":
        print("Ouch is over")
```

This is a **fragment** that goes in the same script as the one above. Notice `&"hurt"`: the `&` in front of a string makes a **StringName**, a text type that is fast to compare. Animation names are StringNames, so we compare with one.

Animations built in code

The editor panel is the friendliest way to author an animation, but it helps to know what it builds, because the same objects exist in code. An animation is an `Animation` resource; a player holds `Animation`s inside an `AnimationLibrary`. Our project builds the "hurt" animation in a script so that you can read every step. Here is the core of it:

```
var anim := Animation.new()
anim.length = 0.5                                  # seconds

var flash: int = anim.add_track(Animation.TYPE_VALUE)   # returns the track number
anim.track_set_path(flash, NodePath("Body:modulate"))   # which node:property
anim.track_insert_key(flash, 0.0, Color.WHITE)          # time, value
anim.track_insert_key(flash, 0.1, Color(1, 0.3, 0.3))
anim.track_insert_key(flash, 0.5, Color.WHITE)

var library := AnimationLibrary.new()
library.add_animation(&"hurt", anim)
$AnimationPlayer.add_animation_library(&"", library)    # the empty name is the default library
```

Read the track path `Body:modulate` as "the node named Body, its property modulate". The part before the colon is a node path **relative to the AnimationPlayer's root node**, which by default is the AnimationPlayer's parent. That is why the player should be a sibling of the nodes it animates, or at least placed where you remember the paths.

> **Warning:** If a node you animate is renamed or moved in the scene, the track path no longer finds it and the track silently does nothing. The editor shows such tracks with a red marker. Rename nodes with the Animation panel open, or fix the path afterwards.

Animating a Sprite2D's frame

The sprite sheet also works with `AnimationPlayer`. Make a track on `Sprite2D:frame`, set the update mode to Discrete, and place keys 2, 3, 4, 5 at 0.0, 0.1, 0.2, 0.3 seconds. Set the length to 0.4 and Loop on. You get the same run cycle as an `AnimatedSprite2D`, and now you can add other tracks to the same animation (the shadow, a footstep sound, a dust puff). The trade-off: `AnimatedSprite2D` is simpler for pure frame animation, and `AnimationPlayer` wins when several things must stay in step.

10.4 Quick effects with Tweens

What a Tween is

Sometimes you do not want to design an animation in advance. You want to say, in one line of code: "make this node's scale grow to 1.5 over a quarter of a second." The `Tween` does that. You create it, tell it what to change, and it does the work over the next frames and then deletes itself.

```
func grow() -> void:
    var tween: Tween = create_tween()
    tween.tween_property(self, "scale", Vector2(1.5, 1.5), 0.25)
```

Read it aloud: *make a tween; tween the property `scale` of `self` to (1.5, 1.5) taking 0.25 seconds.* The tween starts by itself on the next frame. It reads the **current** value of the property as the starting point.

`create_tween()` is a method every node has. A tween made this way is *bound* to that node: if the node is freed, the tween stops by itself, which avoids errors.

The four parts of `tween_property(object, "property", final_value, duration)`:

1. **object**: which node (or resource) to change.
2. **property**: the property name as text. A sub-property uses a colon: `"position:y"`, `"modulate:a"` (just the alpha, the see-through part, of the color).
3. **final_value**: where it should end.
4. **duration**: seconds.

Steps in order, steps together

By default each new `tween_property` call starts **after the previous one finishes**. That makes a sequence easy to write:

```
func pop_in() -> void:
    scale = Vector2.ZERO
    var tween: Tween = create_tween()
    tween.tween_property(self, "scale", Vector2(1.2, 1.2), 0.15)   # step 1
    tween.tween_property(self, "scale", Vector2.ONE, 0.1)          # step 2, after step 1
```

To run a step at the **same time** as the one before it, put `.parallel()` in front:

```
    tween.tween_property(self, "position:y", -50.0, 0.3)
    tween.parallel().tween_property(self, "modulate:a", 0.0, 0.3)   # at the same time
```

Or call `tween.set_parallel(true)` once after creating the tween to make *everything* parallel, and use `.chain()` when you want a step to wait for the previous ones.

Other useful tween steps:

Call	Meaning
`tween.tween_interval(1.0)`	Wait 1 second.
`tween.tween_callback(func_name)`	Call a function at this point in the sequence.
`tween.tween_method(callable, from, to, duration)`	Call a function repeatedly with a number that runs from `from` to `to`. Good for things that are not simple properties.
`tween.set_loops(3)`	Repeat the whole sequence 3 times. `set_loops()` with no number repeats forever.
`tween.kill()`	Stop the tween right now.

A common pattern is the fade-out-then-delete:

```
var tween: Tween = create_tween()
tween.tween_property(self, "modulate:a", 0.0, 0.5)
tween.tween_callback(queue_free)      # when the fade is done, remove this node
```

Note that we pass `queue_free` *without parentheses*. We are handing over the function itself so the tween can call it later. With parentheses it would run immediately.

Easing: how the speed changes

Left alone, a tween moves at a constant speed, which looks robotic. Real things speed up and slow down. Two settings fix that:

- **Transition** (`set_trans`): the *shape* of the speed curve. Common shapes: `Tween.TRANS_LINEAR` (constant), `TRANS_SINE` (gentle), `TRANS_QUAD` and `TRANS_CUBIC` (stronger), `TRANS_BACK` (overshoot and settle), `TRANS_ELASTIC` (spring wobble), `TRANS_BOUNCE` (bounce like a ball).
- **Ease** (`set_ease`): *which end* of the movement the curve is applied to. `Tween.EASE_IN` (slow at the start), `EASE_OUT` (slow at the end), `EASE_IN_OUT` (slow at both).

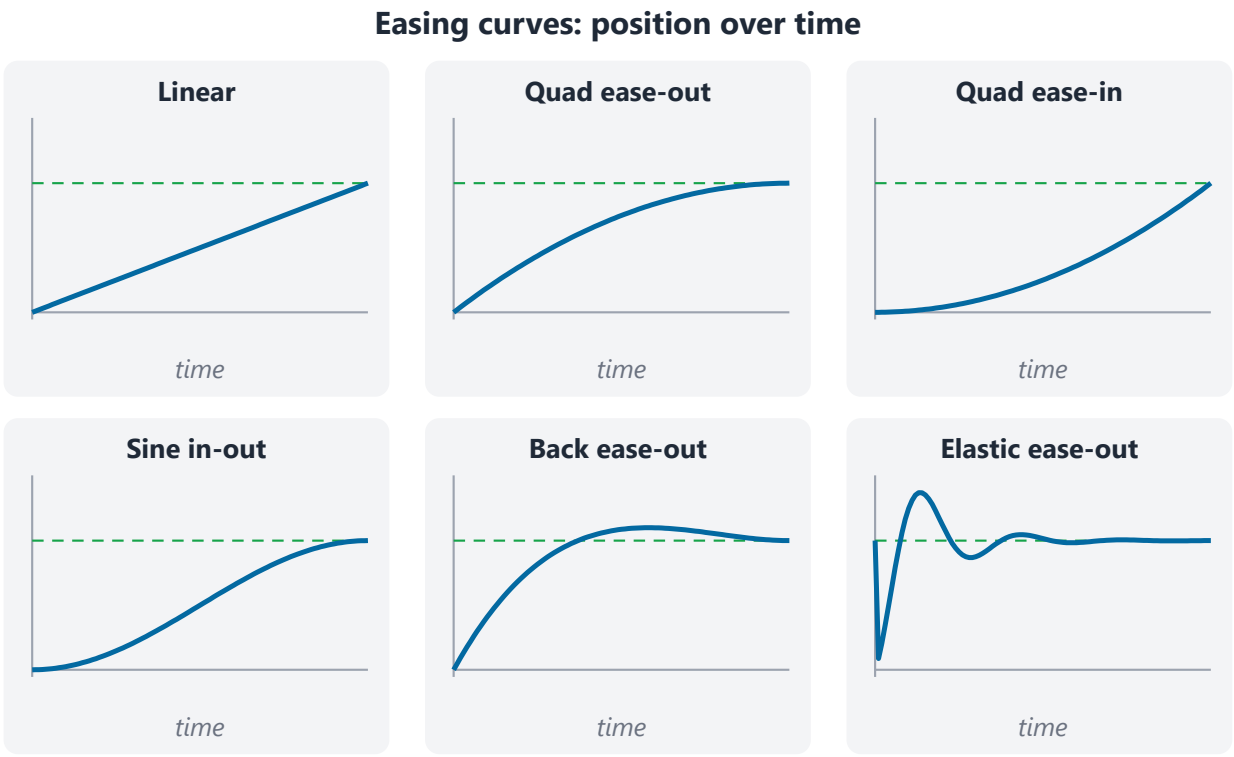


Figure 10.5: Linear, quad ease-out, quad ease-in, sine in-out, back ease-out (overshoot) and elastic ease-out (wobble).

```
tween.tween_property(self, "position:y", 0.0, 0.3) \
    .set_trans(Tween.TRANS_QUAD).set_ease(Tween.EASE_IN)
```

The backslash `\` at the end of a line means "this statement continues on the next line." It only makes long lines readable. Each `set_...` call changes the step that came right before it, which is why they hang on the end of `tween_property`.

Rules of thumb that you can lean on:

- Something **leaving** (falling, a door slamming) feels natural with `EASE_IN`: it starts slowly and gains speed.
- Something **arriving** (landing, a popup appearing) feels natural with `EASE_OUT`: it arrives fast and settles.
- `TRANS_BACK` with `EASE_OUT` overshoots slightly and settles: the classic "pop".
- `TRANS_ELASTIC` with `EASE_OUT` gives a springy wobble at the end.

Two tweens fighting

A property can only have one master at a time. If you start a new tween on the same property while an old tween is still running, both write to it every frame and the result jumps around. The cure is to remember the old tween and kill it first:

```
var hover_tween: Tween

func animate_to(target: Vector2) -> void:
    if hover_tween:
        hover_tween.kill()
    hover_tween = create_tween()
    hover_tween.tween_property(self, "scale", target, 0.15)
```

Our project's hover button uses exactly this pattern.

Tween or AnimationPlayer?

Situation	Choose
The target value is known only at run time (to the mouse position, to the enemy's position)	Tween
One or two properties, short effect, no designer needed	Tween
Several properties and nodes in step, or a non-programmer should edit it in the editor	AnimationPlayer
An effect that has to play the same way every time	AnimationPlayer
An effect that must be triggered in many places with different values	Tween (in a function with parameters)

> **Note:** Tweens are single-use. When a tween finishes, it is finished. To run the effect again, create a new tween. You never store a tween to "restart" it; you write a function that builds one.

10.5 Making motion feel good

Technical animation is only half the job. The other half is making motion feel believable. Disney animators wrote down twelve principles in the 1930s. Two of them give you the most for the least work.

Anticipation

Before a big action, the character prepares: a runner crouches, then sprints. In a jump the crouch tells the player "the jump is coming" and gives the jump power. In code it is just one extra, short step before the main step.

Squash and stretch

A rubber ball flattens when it lands and stretches when it flies fast. Even for characters that are not rubber, a little squash and stretch shows weight and speed. Scale makes it easy: to squash, make the object *wider and shorter* (`Vector2(1.3, 0.7)`); to stretch, make it *narrower and taller* (`Vector2(0.85, 1.2)`). Keep the area about the same: when width goes up, height goes down.

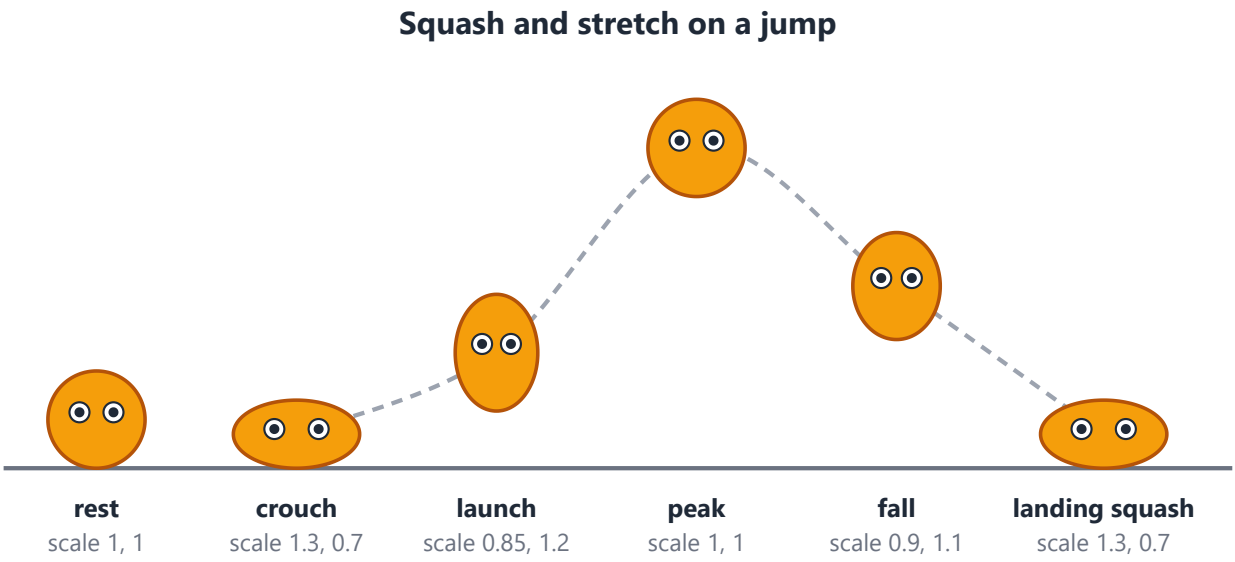


Figure 10.6: The jump in poses. The numbers under each pose are the scale used (width, height).

The whole jump, written as a tween (this is the real code from the project):

```
func jump() -> void:
    jumping = true
    body.play("jump")
    var tween := create_tween()
    tween.tween_property(body, "scale", Vector2(1.3, 0.7), 0.08)     # anticipation: crouch
    tween.tween_property(body, "scale", Vector2(0.85, 1.2), 0.1)     # stretch as we launch
    tween.parallel().tween_property(body, "position:y", -JUMP_HEIGHT, 0.28) \
        .set_trans(Tween.TRANS_QUAD).set_ease(Tween.EASE_OUT)        # rise, slowing near the top
    tween.tween_property(body, "position:y", 0.0, 0.24) \
        .set_trans(Tween.TRANS_QUAD).set_ease(Tween.EASE_IN)         # fall, speeding up
    tween.parallel().tween_property(body, "scale", Vector2.ONE, 0.24)
    tween.tween_property(body, "scale", Vector2(1.3, 0.7), 0.06)     # landing squash
    tween.tween_property(body, "scale", Vector2.ONE, 0.3) \
        .set_trans(Tween.TRANS_ELASTIC).set_ease(Tween.EASE_OUT)     # spring back
    tween.tween_callback(_on_landed)
```

This is a **fragment**: it lives in `pip.gd` (shown in full in the project) where `body`, `jumping`, `JUMP_HEIGHT` and `_on_landed` are defined.

Why the rise uses `EASE_OUT` and the fall uses `EASE_IN`: gravity slows Pip at the top of the arc and speeds him up on the way down. With `TRANS_QUAD` the speed follows the same curve that gravity does, so the jump feels physically right without any physics code.

> **Why it works:** The sprite's pivot (the point it scales around) sits at Pip's feet, because in the project we set the sprite's `offset` to `Vector2(0, -16)`, which lifts the picture so that the node's origin is at its feet. Squash then flattens Pip *into the ground* instead of shrinking him around his belly button.

Keeping logic and animation apart

Look at how `pip.gd` is organized. Movement code changes `position`. Animation code changes the `Body` child: its picture, its scale, its offset, its color. Pip's root node never gets squashed; only the visual child does. This separation has a big payoff: if you later give Pip a physics body (Chapter 13) the squashing will not stretch the collision shape, because the shape is not a child of the squashed node.

A good habit: **put things you want to animate on a child node** (such as `Body`) and keep the "real" node (position, collisions, scripts) clean.

10.6 Putting it together: choosing and combining

Pip uses all three tools:

Effect	Tool	Why
Idle, run and jump pictures	`AnimatedSprite2D`	Pure frame animation
Hurt flash and wobble, with a signal when done	`AnimationPlayer`	Several properties and a method call, designed once
Squash and stretch jump	`Tween`	Short, one-off, purely code
Coin pops in, waits, fades, deletes itself	`Tween`	A sequence with a callback
Button grows when hovered	`Tween`	Starts on a signal, must replace the previous tween
Sheet demo sprite cycling frames	`AnimationPlayer`	Shows how `frame` can be keyed

Combining without conflict

When two animations touch the same property, the last one to write wins, frame by frame. Our project avoids trouble by giving each tool its own properties:

- `AnimatedSprite2D` controls the picture (`animation`, `frame`).
- The jump tween controls `Body:scale` and `Body:position`.
- The hurt animation controls `Body:modulate` and `Body:rotation`.

Because those do not overlap, you can jump *and* get hurt at the same time and everything blends. If you had animated `Body:scale` in the hurt animation, the two would fight. When you design animations, list which properties each one owns.

Performance and cleanup

- Tweens and `AnimationPlayer`s are cheap. You can run hundreds of tweens.
- A tween bound to a node dies when the node dies. A tween you create from a *different* node (`get_tree().create_tween()`) lives on its own; use it only when you want the effect to outlive the node, for example a fade that continues after the object is gone.
- If an animation does not play, check the Output panel: Godot prints an error such as `Animation not found` or `Invalid track path` that tells you what went wrong.

Project 10: Pip comes alive

You will build a small stage where Pip runs, jumps and gets hurt, coins pop up where you click, and a button wobbles when the mouse touches it. It takes about 45 to 60 minutes.

Project folder: `projects/ch10-animation/`

Files you will create:

File	Type	Purpose
`pip_frames.gd`	Script (no node)	Draws the sprite sheet with code and builds the `SpriteFrames`

File	Type	Purpose
`pip.tscn` + `pip.gd`	Scene + script	Pip: frame animation, jump tween, hurt animation
`coin.tscn` + `coin.gd`	Scene + script	A coin that pops in and fades out
`main.tscn` + `main.gd`	Scene + script	The stage, a hover button, a sheet demo sprite
`icon.svg`, `project.godot`	Project files	Created by Godot when you make the project

Step 1: Create the project

1. In the Project Manager click **Create**. Name it `ch10-animation`, pick a folder, renderer **Compatibility** is fine. Click **Create & Edit**.
2. Open **Project > Project Settings > Display > Window** and set **Viewport Width** to 960 and **Viewport Height** to 540.
3. Open **Rendering > Textures** (turn on **Advanced Settings** at the top right if you do not see it). Set **Default Texture Filter** to **Nearest**. This keeps pixel art sharp when it is enlarged, instead of blurry.

Step 2: The art, by code

Create a new script that is **not** attached to any node: in the FileSystem dock right-click, **Create New > Script**. Name it `pip_frames.gd`, and in the dialog set **Inherits** to `RefCounted`. Replace the whole text with:

`pip_frames.gd`

```
class_name PipFrames
extends RefCounted
## Draws a sprite sheet of Pip with code and turns it into SpriteFrames.
## No image files are needed. A real project would load a PNG sheet instead.

const CELL := 32          # each frame is 32 x 32 pixels
const FRAME_COUNT := 7    # idle x2, run x4, jump x1

const BODY := Color(0.95, 0.65, 0.2)
const OUTLINE := Color(0.62, 0.36, 0.1)
const WHITE := Color(1, 1, 1)
const DARK := Color(0.12, 0.12, 0.18)


## Returns one Image holding all frames side by side in a single row.
static func make_sheet() -> Image:
    var img := Image.create(CELL * FRAME_COUNT, CELL, false, Image.FORMAT_RGBA8)
    img.fill(Color(0, 0, 0, 0))  # fully transparent
    # frame 0-1: idle (breathing)      args: index, body y, body ry, left foot, right foot, eye shift
    _draw_pip(img, 0, 18.0, 12.0, 0.0, 0.0, 0.0)
    _draw_pip(img, 1, 19.0, 11.0, 0.0, 0.0, 0.0)
    # frame 2-5: run (feet swap, body bobs)
    _draw_pip(img, 2, 18.0, 12.0, -4.0, 4.0, 1.0)
    _draw_pip(img, 3, 17.0, 12.0, 0.0, 0.0, 1.0)
    _draw_pip(img, 4, 18.0, 12.0, 4.0, -4.0, 1.0)
    _draw_pip(img, 5, 17.0, 12.0, 0.0, 0.0, 1.0)
    # frame 6: jump (tall body, feet tucked, eyes up)
    _draw_pip(img, 6, 16.0, 13.0, 0.0, 0.0, 0.0, true)
    return img


static func _inside(x: float, y: float, cx: float, cy: float, rx: float, ry: float) -> float:
    var dx := (x - cx) / rx
    var dy := (y - cy) / ry
    return dx * dx + dy * dy  # <= 1.0 means inside the ellipse


static func _draw_pip(img: Image, index: int, cy: float, ry: float, foot_l: float, foot_r: float, lean:
```

```
float, jump: bool = false) -> void:
    var ox := index * CELL
    var cx := 16.0 + lean
    var foot_y := 29.0 if not jump else 25.0
    for y in range(CELL):
        for x in range(CELL):
            var fx := float(x) + 0.5
            var fy := float(y) + 0.5
            var color := Color(0, 0, 0, 0)
            # feet first, so the body can cover their tops
            for foot in [[cx - 6.0 + foot_l, foot_y], [cx + 6.0 + foot_r, foot_y]]:
                if _inside(fx, fy, foot[0], foot[1], 4.0, 2.5) <= 1.0:
                    color = OUTLINE
            var d := _inside(fx, fy, cx, cy, 12.0, ry)
            if d <= 1.0:
                color = OUTLINE if d > 0.72 else BODY
                # eyes
                var eye_y := cy - 3.0 - (2.0 if jump else 0.0)
                for ex in [cx - 4.5, cx + 4.5]:
                    if _inside(fx, fy, ex, eye_y, 3.2, 3.2) <= 1.0:
                        color = WHITE
                    if _inside(fx, fy, ex + 0.8 + lean * 0.3, eye_y + (-1.0 if jump else 0.5), 1.6, 1.6)
<= 1.0:
                        color = DARK
            img.set_pixel(ox + x, y, color)


## Cuts the sheet into AtlasTextures, one per frame.
static func make_frame_textures(sheet: Texture2D) -> Array[AtlasTexture]:
    var list: Array[AtlasTexture] = []
    for i in range(FRAME_COUNT):
        var tex := AtlasTexture.new()
        tex.atlas = sheet
        tex.region = Rect2(i * CELL, 0, CELL, CELL)  # x, y, width, height
        list.append(tex)
    return list


## Builds the full SpriteFrames resource: the "idle", "run" and "jump" animations.
static func build() -> SpriteFrames:
    var sheet := ImageTexture.create_from_image(make_sheet())
    var tex := make_frame_textures(sheet)
    var frames := SpriteFrames.new()
    frames.remove_animation("default")  # a new SpriteFrames starts with one called "default"
    _add(frames, "idle", [tex[0], tex[1]], 2.0, true)
    _add(frames, "run", [tex[2], tex[3], tex[4], tex[5]], 10.0, true)
    _add(frames, "jump", [tex[6]], 1.0, false)
    return frames


static func _add(frames: SpriteFrames, anim: StringName, textures: Array, fps: float, looping: bool) ->
void:
    frames.add_animation(anim)
    frames.set_animation_speed(anim, fps)
    frames.set_animation_loop(anim, looping)
    for t in textures:
        frames.add_frame(anim, t)
```

Let us read the important parts.

- `class_name PipFrames` gives the script a global name, so any other script can write `PipFrames.build()` without loading it first. (`PipFrames` is not a built-in Godot class, so the name is free.)
- Every function is `static`. A static function belongs to the *script itself*, not to an object made from it, so we can call it directly.
- `Image.create(width, height, false, Image.FORMAT_RGBA8)` makes an empty picture in memory. `img.set_pixel(x, y, color)` paints one pixel. `_draw_pip` loops over every pixel of one 32 by 32 cell and decides: inside the body ellipse? inside an eye? inside a foot? The test `_inside(...) <= 1.0` is the

standard way to ask "is this point inside an ellipse?"

- `ImageTexture.create_from_image(...)` converts the `Image` (in memory) into a `Texture2D` that a node can show.
- `make_frame_textures` slices the sheet into `AtlasTexture`s, one per cell (the code from section 10.2).
- `build()` assembles the three animations: `idle` with frames 0 and 1 at 2 FPS, `run` with frames 2 to 5 at 10 FPS, and `jump` with frame 6, not looping.

You do not need to understand every line of the pixel drawing. Treat it like a tiny art studio that you can call. The part that matters for this chapter is `build()`.

Step 3: Pip's scene

Create a new scene with a **Node2D** as the root and name it `Pip`. Add two children:

```
Pip (Node2D)                <- script: pip.gd
├── Body (AnimatedSprite2D)
└── AnimationPlayer
```

Select `Body` and set **Offset** (under Offset in the Inspector) to `(0, -16)`. This lifts the picture so the node's origin is at Pip's feet. Select the root `Pip` and set **Scale** to `(3, 3)` so he is big enough to see. Save the scene as `pip.tscn`.

Attach a new script `pip.gd` to the root `Pip` node and paste:

`pip.gd`

```gdscript
extends Node2D
## Pip: walks with arrow keys, jumps with Space, gets hurt with H.

signal hurt_finished

const SPEED := 240.0
const JUMP_HEIGHT := 30.0  # in Pip's own (unscaled) pixels

@onready var body: AnimatedSprite2D = $Body
@onready var player: AnimationPlayer = $AnimationPlayer

var jumping: bool = false


func _ready() -> void:
    body.sprite_frames = PipFrames.build()
    body.play("idle")
    _build_hurt_animation()


func _process(delta: float) -> void:
    var dir := Input.get_axis("ui_left", "ui_right")
    if dir != 0.0:
        position.x = clampf(position.x + dir * SPEED * delta, 60.0, 900.0)
        body.flip_h = dir < 0.0
    if not jumping:
        var wanted: StringName = &"run" if dir != 0.0 else &"idle"
        if body.animation != wanted:
            body.play(wanted)
    if Input.is_action_just_pressed("ui_accept") and not jumping:
        jump()


func _unhandled_input(event: InputEvent) -> void:
    if event is InputEventKey and event.pressed and not event.echo and event.keycode == KEY_H:
        hurt()
```

```
func jump() -> void:
    jumping = true
    body.play("jump")
    var tween := create_tween()
    # 1. crouch (squash)
    tween.tween_property(body, "scale", Vector2(1.3, 0.7), 0.08)
    # 2. launch: stretch tall while rising
    tween.tween_property(body, "scale", Vector2(0.85, 1.2), 0.1)
    tween.parallel().tween_property(body, "position:y", -JUMP_HEIGHT, 0.28) \
        .set_trans(Tween.TRANS_QUAD).set_ease(Tween.EASE_OUT)
    # 3. fall
    tween.tween_property(body, "position:y", 0.0, 0.24) \
        .set_trans(Tween.TRANS_QUAD).set_ease(Tween.EASE_IN)
    tween.parallel().tween_property(body, "scale", Vector2.ONE, 0.24)
    # 4. land: squash, then spring back
    tween.tween_property(body, "scale", Vector2(1.3, 0.7), 0.06)
    tween.tween_property(body, "scale", Vector2.ONE, 0.3) \
        .set_trans(Tween.TRANS_ELASTIC).set_ease(Tween.EASE_OUT)
    tween.tween_callback(_on_landed)


func _on_landed() -> void:
    jumping = false


func hurt() -> void:
    player.play("hurt")


func _build_hurt_animation() -> void:
    var anim := Animation.new()
    anim.length = 0.5

    # Track 0: flash red and back (the path is relative to the AnimationPlayer's parent)
    var flash := anim.add_track(Animation.TYPE_VALUE)
    anim.track_set_path(flash, NodePath("Body:modulate"))
    anim.track_insert_key(flash, 0.0, Color.WHITE)
    anim.track_insert_key(flash, 0.1, Color(1, 0.3, 0.3))
    anim.track_insert_key(flash, 0.2, Color.WHITE)
    anim.track_insert_key(flash, 0.3, Color(1, 0.3, 0.3))
    anim.track_insert_key(flash, 0.5, Color.WHITE)

    # Track 1: wobble
    var wobble := anim.add_track(Animation.TYPE_VALUE)
    anim.track_set_path(wobble, NodePath("Body:rotation"))
    anim.track_insert_key(wobble, 0.0, 0.0)
    anim.track_insert_key(wobble, 0.1, -0.35)
    anim.track_insert_key(wobble, 0.25, 0.3)
    anim.track_insert_key(wobble, 0.4, -0.15)
    anim.track_insert_key(wobble, 0.5, 0.0)

    # Track 2: at the very end, call a method on the node that is the animation root
    var call_track := anim.add_track(Animation.TYPE_METHOD)
    anim.track_set_path(call_track, NodePath("."))
    anim.track_insert_key(call_track, 0.5, {"method": &"_on_hurt_done", "args": []})

    var library := AnimationLibrary.new()
    library.add_animation(&"hurt", anim)
    player.add_animation_library(&"", library)  # "" is the default library


func _on_hurt_done() -> void:
    hurt_finished.emit()
```

Read it in four pieces.

- `_ready` gives the sprite its frames (`PipFrames.build()`), starts `idle`, and builds the hurt animation.

- `_process` reads left/right with `Input.get_axis` (Chapter 7), moves Pip, flips the picture, and chooses `run` or `idle` *unless he is jumping*. The `jumping` variable stops this code from overriding the `jump` picture while the tween runs.
- `jump()` is the tween from section 10.5. The final `tween_callback(_on_landed)` sets `jumping` back to `false`.
- `_build_hurt_animation()` creates three tracks: a color flash, a rotation wobble, and a **method track** that calls `_on_hurt_done` at the end. That function emits the custom signal `hurt_finished` that you declared at the top of the script with `signal hurt_finished`. The main scene listens to that signal.

> **Note:** `ui_left`, `ui_right` and `ui_accept` are built-in actions that every new project already has (arrow keys and Space or Enter). Using them means you do not have to open the Input Map for this project. `KEY_H` is checked directly with an `InputEventKey` because we only need one extra key.

Step 4: The coin

Create a scene with a **Node2D** root named `Coin`, attach `coin.gd`, and save as `coin.tscn`.

`coin.gd`

```gdscript
extends Node2D
## A coin that pops in, waits, fades out and removes itself.


func _ready() -> void:
    scale = Vector2.ZERO
    var tween := create_tween()
    tween.tween_property(self, "scale", Vector2.ONE, 0.4) \
        .set_trans(Tween.TRANS_BACK).set_ease(Tween.EASE_OUT)
    tween.tween_interval(1.0)
    tween.tween_property(self, "modulate:a", 0.0, 0.5)
    tween.tween_callback(queue_free)


func _draw() -> void:
    draw_circle(Vector2.ZERO, 16.0, Color(1.0, 0.85, 0.2))
    draw_arc(Vector2.ZERO, 16.0, 0.0, TAU, 32, Color(0.7, 0.5, 0.0), 3.0)
    draw_line(Vector2(0, -8), Vector2(0, 8), Color(0.7, 0.5, 0.0), 3.0)
```

`_ready` starts with scale zero (invisible), pops to full size with an overshoot (`TRANS_BACK`), waits a second (`tween_interval`), fades out (`modulate:a`), and finally calls `queue_free` so the coin removes itself. The look of the coin comes from `_draw` (Chapter 6).

Step 5: The stage

Create a scene with a **Node2D** root named `Main`:

```
Main (Node2D)                    <- script: main.gd
├── Background (ColorRect)       size 960 x 540, dark blue, Mouse Filter: Ignore
├── Ground (ColorRect)           top at y = 440, size 960 x 100, green, Mouse Filter: Ignore
├── Pip (instance of pip.tscn)   position (300, 450)
├── SheetSprite (Sprite2D)       position (820, 450), scale (3, 3), offset (0, -16), Hframes 7
├── SheetPlayer (AnimationPlayer)
└── HUD (CanvasLayer)
    ├── Hint (Label)             the help text at the top
    └── HoverButton (Button)     text "Hover me", custom minimum size (160, 48), Pivot Offset (80, 24)
```

Instance `pip.tscn` by dragging it from the FileSystem dock into the scene. **Mouse Filter: Ignore** on the two `ColorRect`s matters: by default a `ColorRect` captures mouse clicks, which would stop our "click to spawn a coin" code from receiving them. The **Pivot Offset** on the button tells it to scale around its middle, not its top-left corner.

Attach `main.gd`:

`main.gd`

```gdscript
extends Node2D
## Chapter 10 demo: AnimatedSprite2D (Pip), AnimationPlayer (hurt + sheet), Tweens (jump, coins, button).

const COIN_SCENE: PackedScene = preload("res://coin.tscn")

@onready var sheet_sprite: Sprite2D = $SheetSprite
@onready var sheet_player: AnimationPlayer = $SheetPlayer
@onready var hover_button: Button = $HUD/HoverButton
@onready var pip: Node2D = $Pip

var button_tween: Tween


func _ready() -> void:
    # A Sprite2D that shows one cell of the sheet at a time through its "frame" property.
    sheet_sprite.texture = ImageTexture.create_from_image(PipFrames.make_sheet())
    sheet_sprite.hframes = PipFrames.FRAME_COUNT
    _build_sheet_animation()
    sheet_player.play("cycle")

    hover_button.mouse_entered.connect(_on_button_hover.bind(true))
    hover_button.mouse_exited.connect(_on_button_hover.bind(false))
    hover_button.pressed.connect(_spawn_coin_at.bind(Vector2(480.0, 250.0)))
    pip.hurt_finished.connect(_on_pip_hurt_finished)


func _unhandled_input(event: InputEvent) -> void:
    if event is InputEventMouseButton and event.pressed and event.button_index == MOUSE_BUTTON_LEFT:
        _spawn_coin_at(get_global_mouse_position())


func _spawn_coin_at(where: Vector2) -> void:
    var coin := COIN_SCENE.instantiate()
    coin.position = where
    add_child(coin)


func _on_pip_hurt_finished() -> void:
    print("Pip finished the hurt animation")


func _on_button_hover(is_over: bool) -> void:
    if button_tween:
        button_tween.kill()  # stop the previous tween so two do not fight
    button_tween = create_tween()
    var target := Vector2(1.15, 1.15) if is_over else Vector2.ONE
    button_tween.tween_property(hover_button, "scale", target, 0.15) \
        .set_trans(Tween.TRANS_BACK).set_ease(Tween.EASE_OUT)


func _build_sheet_animation() -> void:
    var anim := Animation.new()
    anim.length = 0.4
    anim.loop_mode = Animation.LOOP_LINEAR
    var track := anim.add_track(Animation.TYPE_VALUE)
    anim.track_set_path(track, NodePath("SheetSprite:frame"))
    anim.value_track_set_update_mode(track, Animation.UPDATE_DISCRETE)
    anim.track_set_interpolation_type(track, Animation.INTERPOLATION_NEAREST)
    anim.track_insert_key(track, 0.0, 2)
```

```
    anim.track_insert_key(track, 0.1, 3)
    anim.track_insert_key(track, 0.2, 4)
    anim.track_insert_key(track, 0.3, 5)
    var library := AnimationLibrary.new()
    library.add_animation(&"cycle", anim)
    sheet_player.add_animation_library(&"", library)
```

Notes:

- `preload("res://coin.tscn")` loads the coin scene once, when the script is read. `.instantiate()` then creates a fresh copy each time, as you learned with instancing in Chapter 2.
- `_on_button_hover` uses `.bind(true)` and `.bind(false)` to tell one function whether the mouse entered or left. `bind` attaches an extra argument to a signal connection.
- `_build_sheet_animation` is the sprite-frame animation from section 10.3: a `frame` track with Discrete update.

Set `main.tscn` as the main scene (**Project > Project Settings > Application > Run > Main Scene**, or press **F5** and choose **Select Current**) and save everything.

Step 6: Run and test

Press **F5**. Check each of these:

1. Pip breathes (a slow idle). Hold **Left** or **Right**: he runs and faces the right way.
2. Press **Space**: he crouches, springs up stretched, falls, squashes on landing and wobbles back.
3. Press **H**: he flashes red and wobbles. Look at the Output panel: after half a second it prints `Pip finished the hurt animation`.
4. Press **H** while he is mid-jump. Both effects play together without a clash.
5. Click anywhere on the stage: a coin pops in, waits, fades and disappears.
6. Move the mouse over the button: it grows with a little overshoot, and shrinks when you leave.
7. At the right, the little Pip runs forever on his own: that is the `Sprite2D` driven by `AnimationPlayer`.

Expected result

A dark-blue stage with a green floor. An orange round Pip with big eyes stands on the floor and breathes. Arrow keys make him run, Space makes him jump with visible squash and stretch, and H makes him flash red and wobble. Clicking spawns coins that pop in and fade away, a button grows when hovered, and a second tiny Pip at the right runs in place.

Making it your own (optional)

- Change `JUMP_HEIGHT` and the durations in `jump()` and feel how the personality of the jump changes.
- Change `TRANS_ELASTIC` in the final landing step to `TRANS_BOUNCE`.
- Add a fourth animation to the sprite sheet (for example a "wave") by drawing another frame in `make_sheet` and adding it in `build()`.

Chapter summary

- Animation is a sequence of changes shown fast enough to look like motion. An animation's FPS is separate from the game's frame rate.
- `AnimatedSprite2D` plays named frame animations stored in a `SpriteFrames` resource. Sprite sheets are cut into frames with `AtlasTexture` regions or the editor's "Add frames from sprite sheet" tool.

- `AnimationPlayer` animates any property of any node with tracks and keyframes. Property tracks use `Node:property` paths relative to the player's root node. Method tracks call functions at a moment in time.
- Use Discrete update mode for values that cannot blend (frame numbers, booleans), and Continuous for numbers and colors.
- `create_tween()` makes a one-shot helper. `tween_property` changes a property; steps run in order unless you use `.parallel()`; `tween_callback` runs a function; `kill()` prevents two tweens fighting over one property.
- Easing (`set_trans` and `set_ease`) turns robotic motion into natural motion. Out for arriving, in for leaving.
- Anticipation and squash and stretch make a jump feel weighty. Animate a child node so the root, its collision shape and its logic stay clean.

Review questions

1. **Short answer.** An animation plays at 10 FPS while the game runs at 60 frames per second. How many game frames pass for each animation frame, on average?

2. **Multiple choice.** Which node is the best choice for a character walk cycle made of drawn pictures? a) `Tween` b) `AnimatedSprite2D` c) `AnimationPlayer` d) `Timer`

3. **True or false.** Calling `body.play("run")` every frame while `run` is already playing restarts the animation from frame 0 each time.

4. **Short answer.** What does `body.flip_h = true` do, and why is it useful for a character that can walk left and right?

5. **Multiple choice.** You want to animate a sprite's `frame` property from the `AnimationPlayer`. Which update mode is correct? a) Continuous b) Discrete c) Capture d) It does not matter

6. **Short answer.** In the track path `Body:modulate`, what does each side of the colon mean, and relative to what is the left side measured?

7. **Multiple choice.** Which line makes a node fade to fully transparent over half a second? a) `tween.tween_property(self, "modulate:a", 0.0, 0.5)` b) `tween.tween_property(self, "alpha", 0, 0.5)` c) `tween.fade(self, 0.5)` d) `tween.tween_property(self, "visible", false, 0.5)`

8. **Short answer.** By default, how does a tween decide when the second `tween_property` call starts? What do you add to make it run at the same time as the first?

9. **True or false.** A finished `Tween` can be restarted later by calling `play()` on it, so you should store it in a variable and reuse it.
10. **Short answer.** A button grows on `mouse_entered` and shrinks on `mouse_exited`, but when you move the mouse in and out quickly it jitters. What is the likely cause and the fix?

Exercises

1. **Practice (easy).** In the project, change the run animation speed from 10 FPS to 16 FPS and the idle speed to 1 FPS. Run it and describe how each looks. Then set `body.speed_scale = 2.0` in `_ready` and observe what changes.
2. **Practice (easy).** Make the coin pop in with `TRANS_ELASTIC` instead of `TRANS_BACK`, and make it stay for 3 seconds instead of 1.

3. **Apply (medium).** Add a `bounce` animation to the `AnimationPlayer` in `pip.tscn` (built in code in `_build_hurt_animation`'s style): a value track on `Body:position:y` with keys 0.0 at time 0, -12.0 at time 0.15 and 0.0 at time 0.3. Play it when the player presses the **B** key. Does it clash with jumping? Explain why.
4. **Apply (medium).** Add a `Label` that shows the text "Landed!" for one second when Pip lands. Use a tween to fade it out. Hint: use the `_on_landed` function.
5. **Challenge (hard).** Create a "dash": when the player presses **Shift**, Pip moves 200 pixels in the direction he faces in 0.15 seconds using a tween with `TRANS_EXPO` and `EASE_OUT`, and leaves three fading "ghost" copies behind (create each ghost as a coin-like Node2D that tweens `modulate:a` to 0 and frees itself). Make sure that two dashes in a row do not fight each other.

Common mistakes

- **Symptom:** The animation does nothing and the Output panel says `Animation not found` or the track shows a red mark. **Cause:** The animation name or the track path (`Body:modulate`) does not match the real node name. **Fix:** Check spelling and capital letters; node names are case sensitive. Make the track path relative to the player's root node.
- **Symptom:** `AnimatedSprite2D` shows nothing. **Cause:** The `Sprite Frames` property is empty, no animation is playing, or the animation has no frames. **Fix:** Set `sprite_frames`, call `play("name")` or enable Autoplay, and confirm that the animation list shows frames.
- **Symptom:** The picture is blurry when scaled. **Cause:** The default texture filter is Linear. **Fix:** Project Settings > Rendering > Textures > Default Texture Filter = Nearest (or set Texture Filter on the node).
- **Symptom:** The animation restarts every frame and looks stuck on the first frame. **Cause:** `stop()` and `play()` are called together every frame, or two scripts keep switching between two different animations. **Fix:** Only call `play` when the state changes: compare with `body.animation` first, as `pip.gd` does.
- **Symptom:** Two effects make a property jitter or ignore each other. **Cause:** Two tweens, or a tween and an `AnimationPlayer`, write to the same property. **Fix:** Give each effect its own properties, or store the old tween and call `kill()` before creating a new one.
- **Symptom:** A squashed sprite shrinks into the air instead of flattening onto the ground. **Cause:** The sprite's origin is at its center. **Fix:** Offset the sprite so the node's origin is at the feet, or set the `pivot_offset` for Controls.

Further reading

- Using AnimatedSprite2D: https://docs.godotengine.org/en/stable/tutorials/2d/2d_sprite_animation.html
- Introduction to the animation features: https://docs.godotengine.org/en/stable/tutorials/animation/introduction.html
- Tween class reference: https://docs.godotengine.org/en/stable/classes/class_tween.html
- AnimationPlayer class reference: https://docs.godotengine.org/en/stable/classes/class_animationplayer.html
- SpriteFrames class reference: https://docs.godotengine.org/en/stable/classes/class_spriteframes.html

Chapter 11: Sound and Music

Learning objectives

By the end of this chapter you will be able to:

- Describe what digital sound is (samples, sample rate, amplitude, frequency) and why decibels are used for volume.
- Play sounds and music with `AudioStreamPlayer`, and set volume, pitch and looping.
- Import real audio files, and choose between WAV and Ogg Vorbis for sound effects and music.
- Create audio buses, add effects such as reverb and a low-pass filter, and control buses from a script with `AudioServer`.
- Generate sound effects and a music loop from pure code, with no audio files, using `AudioStreamWAV` and `AudioStreamGenerator`.
- Add positional (2D) sound, and apply good habits such as pitch variation, sound pools and music fades.

Key terms

- **Sound wave**: a pattern of air pressure that rises and falls. A speaker makes one; a microphone records one.
- **Sample**: one measurement of a sound wave at one instant, stored as a number.
- **Sample rate (mix rate)**: how many samples are stored or played each second, measured in hertz (Hz). CD quality is 44,100.
- **Frequency**: how many times a wave repeats each second. High frequency sounds high-pitched.
- **Amplitude**: how tall the wave is. Larger amplitude sounds louder.
- **Decibel (dB)**: a logarithmic unit used for volume. 0 dB means "unchanged", and negative numbers are quieter.
- **AudioStream**: a resource that holds sound data (a WAV file, an Ogg file, or sound made by code).
- **AudioStreamPlayer**: a node that plays an `AudioStream` through an audio bus.
- **Audio bus**: a channel that mixes sounds together, applies effects and passes the result on, finally to the speakers.
- **Master bus**: the last bus. Everything you hear goes through it.
- **Envelope**: how a sound's volume changes over its life, for example a quick fade-in and a slow fade-out.
- **Pitch scale**: a multiplier that makes a sound higher or lower (and shorter or longer) when played.

11.1 What sound is, for a computer

From air pressure to numbers

When a guitar string vibrates, it pushes the air in rhythmic waves. Your ear feels the waves as sound. A computer cannot store a smooth wave, so it does what a film camera does with motion: it takes many snapshots. Each snapshot is a **sample**: a single number that says how far the wave is from the middle at that instant. Play the numbers back in order, quickly, through a speaker, and the wave is rebuilt.

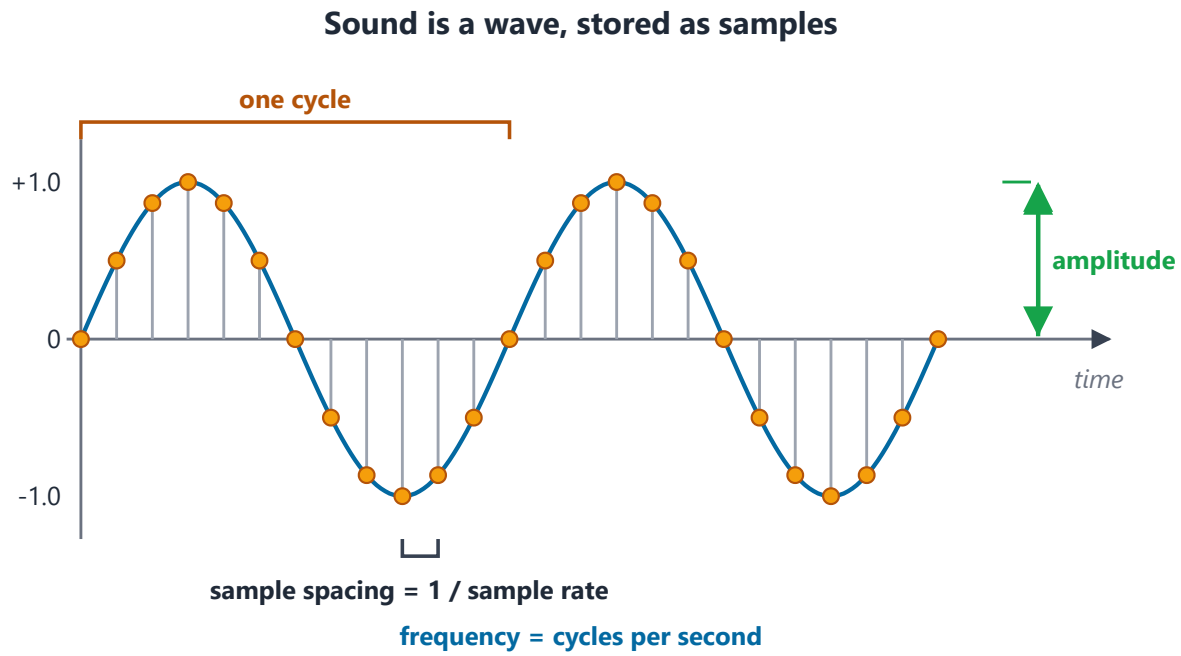


Figure 11.1: A wave stored as samples. Frequency is how many cycles fit in one second. Amplitude is the height.

Three numbers describe almost everything you need:

- **Sample rate**: samples per second. More samples capture higher frequencies. Godot's default mix rate is 44,100 Hz. A sound sampled at 22,050 Hz can contain frequencies up to about 11,000 Hz (half the sample rate), which is plenty for game blips and chiptune music.
- **Frequency** (pitch): cycles per second, in hertz. The note A above middle C is 440 Hz. A wave twice as fast, 880 Hz, is the same note one octave higher.
- **Amplitude** (loudness): the height of the wave. In Godot's generated audio, sample values range from -1.0 to 1.0.

Files and formats

Godot can import three audio file formats:

Format	Size	Good for	Notes
WAV	Large (raw samples)	Short sound effects	Plays with almost no processing, which is ideal for a jump sound that must play *instantly*.
Ogg Vorbis (`.ogg`)	Small (compressed)	Music and long sounds	Needs decoding while it plays, which is fine for a few streams.
MP3	Small (compressed)	Music	Supported, but Ogg is the usual choice in Godot.

Rule of thumb: **WAV for effects shorter than a few seconds, Ogg for music.**

Sound in the Godot node tree

Playing sound needs two things, like showing a picture needs a texture and a `Sprite2D`:

1. An `AudioStream`: the sound data. An imported file becomes an `AudioStream` resource automatically. In this chapter we also make streams with code.

2. A **player node** that plays it:
    - `AudioStreamPlayer`: plays the same everywhere. Use it for music and interface sounds.
    - `AudioStreamPlayer2D`: positional. It gets quieter with distance and moves between the left and right speakers as the node moves. Use it for sounds that come from something in the 2D world.
    - `AudioStreamPlayer3D`: the same idea in 3D space (you will meet it in Chapter 19).

11.2 Playing sounds with AudioStreamPlayer

The editor way

1. Download or make a sound file, such as `jump.wav`, and drag it into the FileSystem dock. Godot imports it.
2. Add an **AudioStreamPlayer** node to your scene.
3. Drag the file onto the player's **Stream** property in the Inspector (or click the property and choose Load).
4. Turn on **Autoplay** to start when the scene starts, or play it from a script.

> **Note:** Where to get free sounds: Kenney.nl (Audio packs), OpenGameArt.org and freesound.org (check each file's license). They are optional; this chapter's project makes all its sounds with code.

The properties that matter:

Property	Meaning
`stream`	The `AudioStream` to play.
`volume_db`	Volume in decibels. 0 is unchanged; negative is quieter.
`pitch_scale`	Speed and pitch multiplier. 1.0 normal, 2.0 an octave higher and twice as fast.
`autoplay`	Start playing when the node enters the scene.
`bus`	Name of the audio bus to play through (default `Master`).
`playing`	`true` while the sound is playing.
`stream_paused`	Pause and resume the sound without losing its position.
`max_polyphony`	How many copies of the same stream may overlap on this one player.

Playing from a script

```
@onready var jump_sound: AudioStreamPlayer = $JumpSound

func jump() -> void:
    jump_sound.play()
```

This is a **fragment**: it assumes the scene has an `AudioStreamPlayer` named `JumpSound` with a stream assigned. The most-used members of the player:

Member	Meaning
`play()`	Start from the beginning. If it is already playing, it starts again from the beginning (cutting itself off).
`play(2.5)`	Start 2.5 seconds into the sound.
`stop()`	Stop.
`seek(1.0)`	Jump to 1.0 seconds in.

Member	Meaning
`get_playback_position()`	Seconds played so far.
`finished` (signal)	Sent when the sound reaches the end. It is **not** sent for looping sounds, because they never end.

You can also load a sound file in code:

```
var jump_stream: AudioStream = preload("res://sounds/jump.wav")

func _ready() -> void:
    $JumpSound.stream = jump_stream
```

`preload` loads the file while the script is read, so it is ready when the game starts. This is a **fragment** that assumes `res://sounds/jump.wav` exists in your project. It is the fastest way to use a real file. If the path is wrong, Godot reports an error when the script loads.

Looping and the Import dock

Music should repeat. The loop setting belongs to the **stream**, not to the player, and you set it in the **Import** dock (next to the Scene dock):

1. Click the audio file in the FileSystem dock. The Import tab shows its import options.
2. For **WAV** files, choose a **Loop Mode** such as *Forward*. For **Ogg** and **MP3**, tick **Loop** (and optionally set the **Loop Offset**, the point where the loop jumps back to).
3. Click **Reimport**.

If the loop has an audible click at the join, the end of the sound does not match its beginning. Good loops are cut so that the last sample flows naturally into the first. We will meet that exact problem again when we generate our own music loop.

> **Warning:** Playing the *same* sound many times a second (footsteps in a crowd, a machine gun) can pile up and become loud mush. Section 11.6 shows how to handle it.

11.3 Loudness: decibels, not percentages

Why decibels

Ears do not hear loudness in a straight line. Doubling the energy of a sound does not sound twice as loud. Audio engineers use **decibels**, a scale that follows how people hear. Godot uses decibels for `volume_db` and for bus volumes.

The scale you need:

dB	Effect on the wave's height (amplitude)	Rough feel
+6	Twice as big	Much louder, may distort
0	Unchanged	The sound as it was made
-6	Half	Clearly quieter
-12	A quarter	Background level
-20	One tenth	Faint

dB	Effect on the wave's height (amplitude)	Rough feel
-40	One hundredth	Almost silent
-80	Practically zero	Silence (the lowest the Inspector allows)

Each 6 dB step roughly halves or doubles the amplitude. 0 dB is the "do not change it" setting.

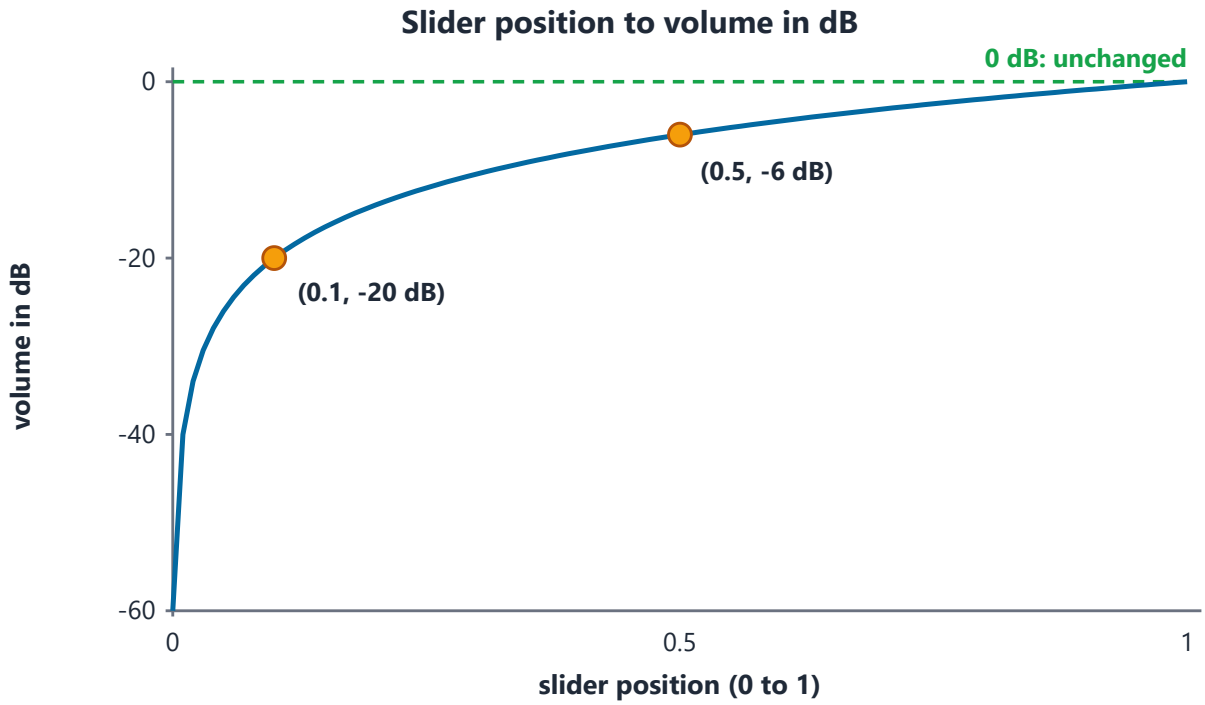


Figure 11.2: Linear slider values (0 to 1) become decibels with `linear_to_db`. Most of the change happens near the quiet end.

Converting for sliders

A volume slider that goes from 0 to 1 (0 to 100 percent) is friendly for the player, but the audio system wants dB. Two built-in functions convert:

```
var db: float = linear_to_db(0.5)       # about -6.02
var amount: float = db_to_linear(-6.0)  # about 0.5
```

`linear_to_db(0.0)` is minus infinity, meaning "silence". Setting a bus to that value works, but it is cleaner to also *mute* the bus when the slider is at the very bottom, as the project does.

> **Try it:** In the editor, add an `AudioStreamPlayer` with any sound and set `volume_db` to -6, then -12, then -20. Compare how each sounds. Notice that going from 0 to -6 sounds like a big drop, but from -34 to -40 you barely notice.

11.4 Buses: mixing and effects

The idea

If every sound went straight to the speakers, you could not make the music quieter without changing every music player, and you could not offer separate "music" and "effects" volume sliders. Instead, sounds go through **buses**. A bus is like a channel on a mixing desk: it adds up all the sounds sent to it, can change their volume, apply effects, and then sends the result onward, usually to the **Master** bus and then to your speakers.

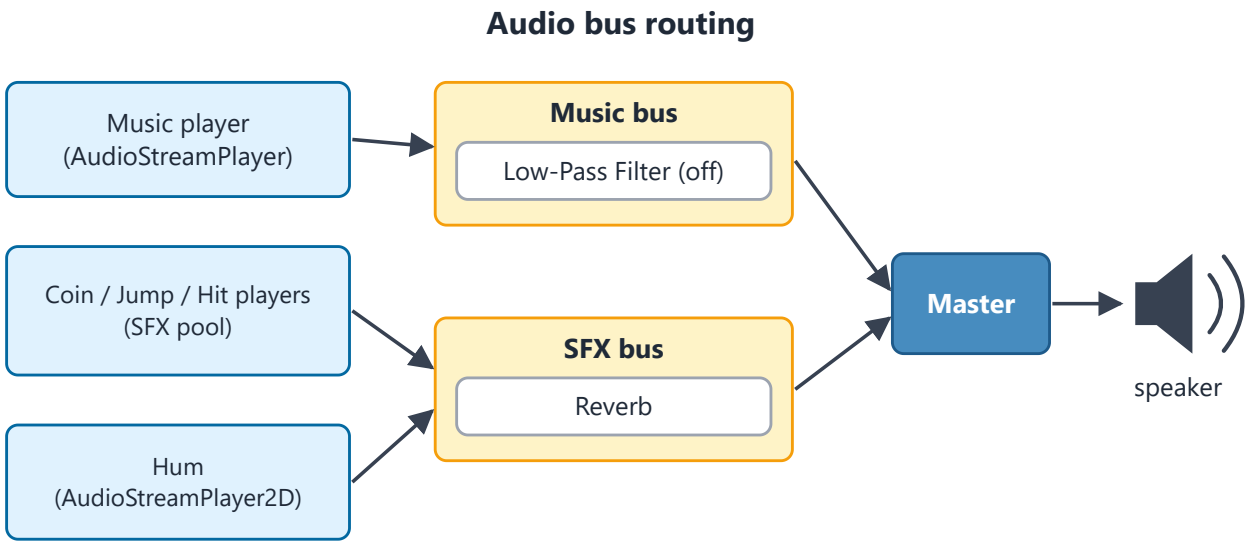


Figure 11.3: The bus layout used in the project. Each player chooses a bus; each bus sends to Master.

Common setup: `Master` (always exists), `Music`, `SFX`, and sometimes `UI` and `Voice`. Then the options menu can have a slider per bus.

The Audio panel

Open the **Audio** panel at the bottom of the editor.

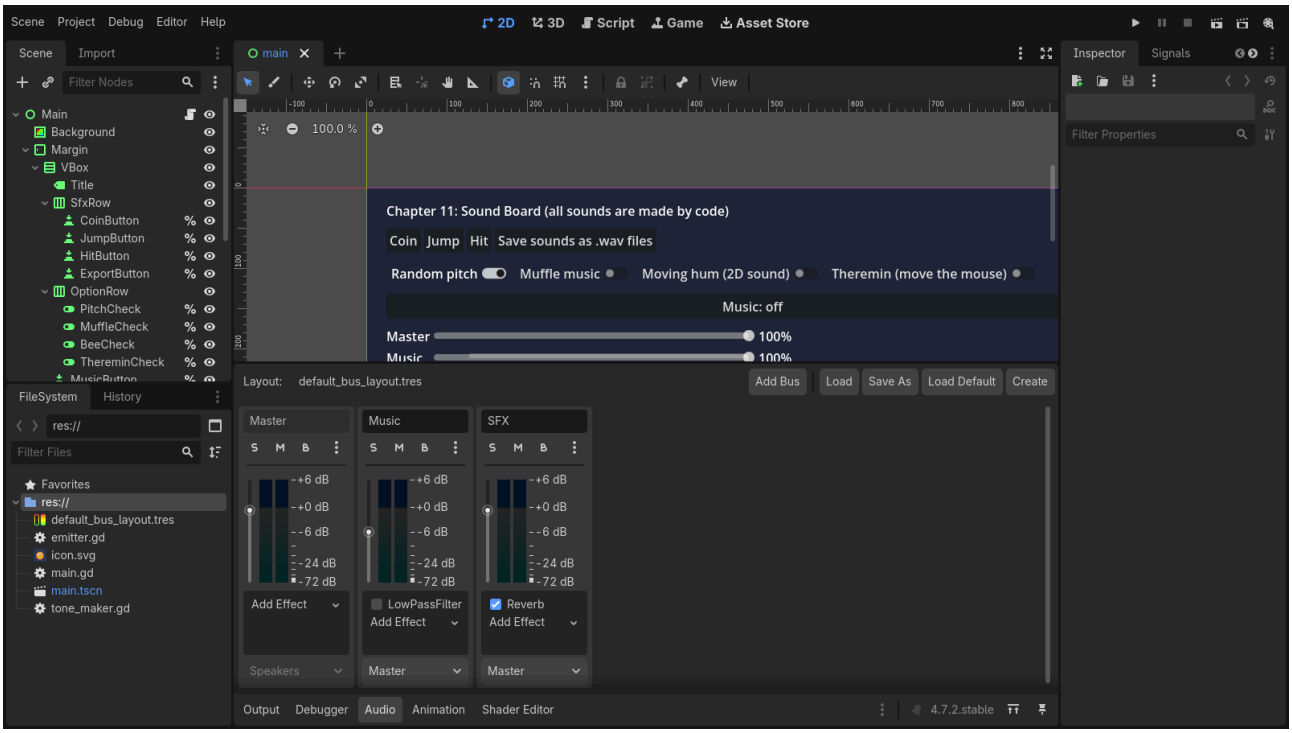


Figure 11.4: The Audio panel. Each bus has a volume fader, M (mute), S (solo) and B (bypass effects) buttons, a Send selector and an effect list.

1. Click **Add Bus** twice. Rename the new buses to `Music` and `SFX` (double-click the name).
2. Each has a **Send** drop-down at the bottom: leave it on `Master`.
3. On the `SFX` bus, click the empty effect slot and choose **Add Effect > Reverb**. Reverb adds the echo of a room. Click the effect to edit it in the Inspector; set **Wet** to about 0.15 for a light touch.
4. On the `Music` bus, add **Low-Pass Filter** and set its **Cutoff Hz** to about 900. A low-pass filter lets low sounds through and muffles high ones, like music heard through a wall. Turn the effect off with its small checkbox so that it only runs when we want it.
5. Save the layout with the **Save** icon at the top right of the panel. It goes to `res://default_bus_layout.tres`, and Godot loads that file automatically when the game starts.

Other effects you can add: Delay, Chorus, Distortion, High-Pass Filter, Compressor (evens out loud and quiet), Limiter (prevents clipping), Equalizer, Pitch Shift and more. Try them with music; they cost nothing but time.

Choosing the bus for a player

Set the **Bus** property of each `AudioStreamPlayer` (or `player.bus = &"SFX"` in code). A player whose bus does not exist falls back to `Master`, so a typo causes loud mistakes rather than silence.

Controlling buses with AudioServer

The editor panel is for design. During play, scripts use the global `AudioServer`:

```
var music_bus: int = AudioServer.get_bus_index("Music")   # buses are numbered; Master is 0
AudioServer.set_bus_volume_db(music_bus, -10.0)
AudioServer.set_bus_mute(music_bus, true)
AudioServer.set_bus_effect_enabled(music_bus, 0, true)    # effect number 0 on that bus
```

Call	Meaning
`AudioServer.get_bus_index(name)`	The number of the bus with that name, or -1 if it does not exist.
`AudioServer.get_bus_volume_db(bus)` and `set_bus_volume_db(bus, db)`	Read and set the bus volume.
`AudioServer.set_bus_mute(bus, true)`	Mute or unmute.
`AudioServer.set_bus_effect_enabled(bus, effect_index, on)`	Switch one effect on or off.
`AudioServer.bus_count`	How many buses exist.

A very common use is the **muffled pause**: when the player opens the pause menu, enable a low-pass filter on the music bus, and the music sounds dull and distant. When the game resumes, disable it. The project has a check box that does exactly this.

> **Note:** Buses can also be created in code (`AudioServer.add_bus()`, `add_bus_effect()`), but a saved layout file is simpler and keeps your design visible in the editor. Use code only if you must build a bus layout at run time.

11.5 Making sound with code

Here is the part of the chapter that needs no audio files. Since a sound is a list of numbers, we can *compute* the numbers.

A tone is a sine wave

The simplest sound is one pure note. Its samples follow the sine function, the same wave you see in Figure 11.1. If `t` is the time in seconds, then the sample at that moment is

```
sample = sin(TAU * frequency * t)
```

`TAU` is the constant 2π (a full turn in radians, Chapter 6). One second contains `frequency` full turns, so a frequency of 440 gives the note A.

Other wave shapes sound different:

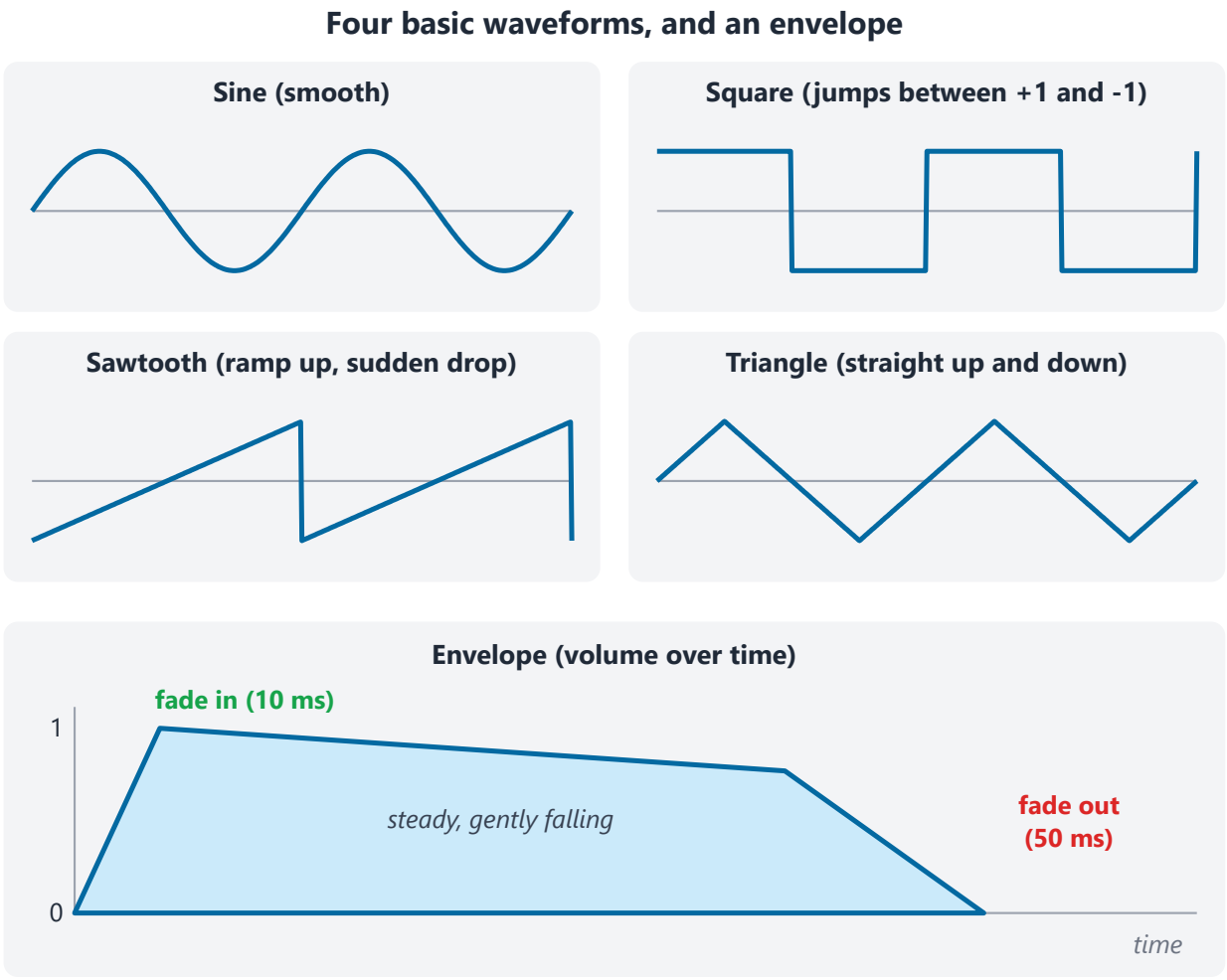


Figure 11.5: Four basic waveforms and an envelope. Square waves sound buzzy and retro; triangles are soft; saws are bright.

- **Sine**: pure and soft.
- **Square**: jumps between +1 and -1; hollow, buzzy; the classic early-console sound.
- **Sawtooth**: ramps up then drops; bright and harsh.
- **Triangle**: ramps up and down; soft like a flute.

The project's `ToneMaker.wave()` function returns one point of any of these:

```
static func wave(kind: String, phase: float) -> float:
    var p := fposmod(phase, 1.0)     # keep only the part after the last whole cycle
    match kind:
        "square":
            return 1.0 if p < 0.5 else -1.0
        "saw":
            return 2.0 * p - 1.0
        "triangle":
            return 4.0 * absf(p - 0.5) - 1.0
        _:
            return sin(TAU * p)
```

Here `phase` counts whole cycles (so `frequency * t`). `fposmod(x, 1.0)` is the "remainder after dividing by 1", which gives a number between 0 and 1 that repeats. `match` is like a row of `if`/`elif` that compares one value against several options. The `_` at the end is the "anything else" case.

From numbers to an AudioStreamWAV

An `AudioStreamWAV` is a stream whose data is raw samples. To fill one with our own samples we convert each number to a 16-bit whole number (-32768 to 32767) and write two bytes per sample into a `PackedByteArray`:

```
static func to_stream(samples: PackedFloat32Array, loop: bool = false) -> AudioStreamWAV:
    var data := PackedByteArray()
    data.resize(samples.size() * 2)             # 2 bytes per 16-bit sample
    for i in range(samples.size()):
        var v := int(clampf(samples[i], -1.0, 1.0) * 32767.0)
        data.encode_s16(i * 2, v)               # write a signed 16-bit number at this byte position
    var stream := AudioStreamWAV.new()
    stream.format = AudioStreamWAV.FORMAT_16_BITS
    stream.mix_rate = 22050
    stream.stereo = false
    stream.data = data
    if loop:
        stream.loop_mode = AudioStreamWAV.LOOP_FORWARD
        stream.loop_begin = 0
        stream.loop_end = samples.size()
    return stream
```

`clampf` keeps each number inside the allowed range; a sample beyond 1.0 would otherwise wrap around and produce a nasty crack. A `PackedFloat32Array` is an array specialized to hold decimal numbers compactly, and `PackedByteArray` holds raw bytes. We use these "packed" types because plain arrays would be slow for hundreds of thousands of values.

Envelopes: avoiding clicks

If a note starts at full volume, the speaker cone jumps instantly and you hear a click. If it ends abruptly, there is another click. An **envelope** multiplies the samples by a number that rises and falls gently:

```
var env := minf(t / 0.01, 1.0) * minf((seconds - t) / 0.05, 1.0)
```

The first factor grows from 0 to 1 in the first 10 milliseconds (fade in). The second factor falls from 1 to 0 over the final 50 milliseconds (fade out). `minf(a, b)` returns the smaller of two numbers, so each factor stays at 1.0 when it is not fading.

Sweeps, noise and notes

With the same building blocks we can make a whole library of sounds:

- **A jump sound** is a square wave whose frequency slides *up*. Instead of `frequency * t` we add up the phase little by little: `phase += freq / MIX_RATE` where `freq` changes through the sound with `lerpf(f_start, f_end, t)`.
- **A hit or explosion** is **noise**: each sample is a random number, `randf_range(-1.0, 1.0)`, scaled to fade quickly.
- **A coin** is two short square-wave notes: B then E a fourth above, the "ba-ding" you know from many games.

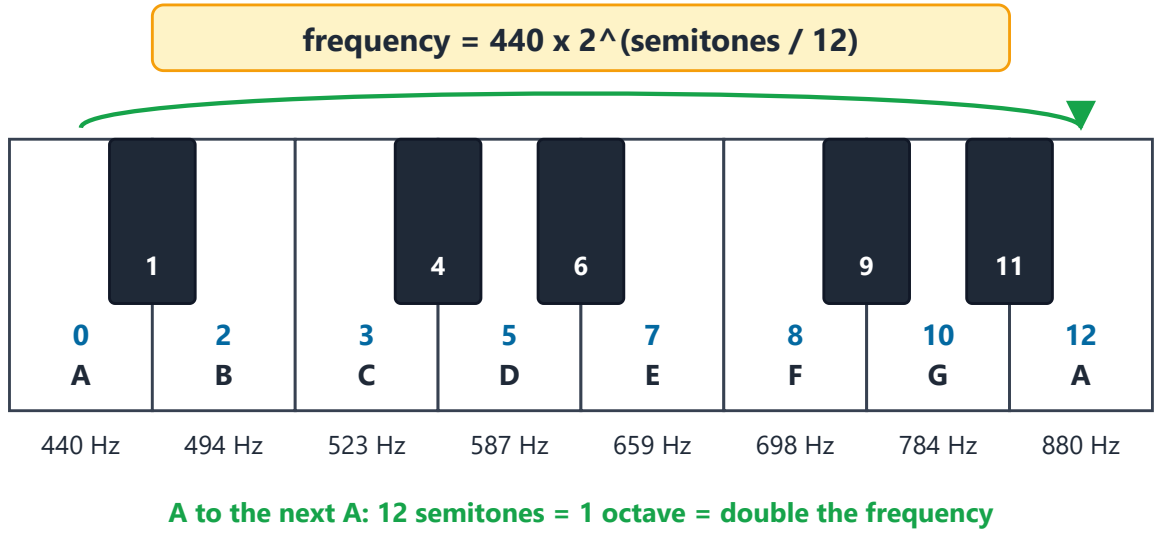


Figure 11.6: Moving up one semitone multiplies the frequency by about 1.0595. Twelve semitones make an octave, which doubles it.

To play music we need the frequency of each note. The rule is simple: counted in *semitones* (piano keys, black and white) up or down from A at 440 Hz, the frequency is `440 * 2^(semitones / 12)`. So 12 semitones up gives exactly twice 440. In code:

```
static func note_freq(semitones: int) -> float:
    return 440.0 * pow(2.0, semitones / 12.0)
```

`pow(a, b)` means "a to the power b". Our 8-second tune is a list of semitone numbers for the melody, and another list for the bass. The helper `_add_note` renders each note and **adds** its samples into the shared buffer, which is exactly how mixing works: two sounds at the same moment are simply the sum of their samples.

> **Why it works:** Mixing really is addition. If the melody sample is 0.1 and the bass sample is 0.3, the speaker receives 0.4. That is why we keep each voice at low volume (0.12 and 0.3): if the sum goes beyond 1.0, `clampf` cuts it off and you hear distortion.

Looping without a click

For a loop with no audible join, make the loop length an exact number of whole cycles (or musical beats). Our music is exactly 16 notes of 0.5 seconds, so the sound ends at the end of a beat. The hum in the project is 100 Hz and lasts exactly 1.0 second, which contains exactly 100 cycles, so the last sample connects smoothly to the first.

Saving your sounds as files

A generated stream can be written to a real WAV file, which you can then keep, edit in other software and import like any other sound:

```
var stream: AudioStreamWAV = ToneMaker.coin()
stream.save_to_wav("user://coin.wav")
```

`user://` is a folder that the game may always write to (Chapter 17 explains it). The project's button **Save sounds as .wav files** does this for all its sounds and prints where the folder is.

Live sound: AudioStreamGenerator

Everything so far prepared a *finished* sound first. `AudioStreamGenerator` is for sound computed **while it plays**, such as a synthesizer that follows the mouse. You give the player an `AudioStreamGenerator` as its stream. When it plays, the player exposes a **playback** object. Each frame you push as many new samples as the generator has room for:

```
var playback: AudioStreamGeneratorPlayback = $Theremin.get_stream_playback()
var room: int = playback.get_frames_available()
for i in range(room):
    playback.push_frame(Vector2(sample, sample))   # left and right channel
```

Each frame is a `Vector2`: `x` is the left speaker, `y` the right. If you push too few frames, the sound crackles because the speaker runs out of data. The project's "theremin" keeps a `phase` variable that advances by `frequency / mix_rate` each sample, so changing the frequency (here from the mouse's x position) never causes a jump in the wave.

> **Note:** Godot's headless mode (no window) has no sound card. The code still runs, but you hear nothing. On your machine, with speakers, you will.

11.6 Good habits for game audio

Vary repeated sounds

The same footstep sample played 20 times in a row sounds robotic. Cheap fix: change the pitch a little each time.

```
player.pitch_scale = randf_range(0.9, 1.1)
player.play()
```

`randf_range(0.9, 1.1)` picks a random decimal between 0.9 and 1.1. A 10 percent change is enough to feel different without sounding like another sound.

Let sounds overlap: a pool of players

A single `AudioStreamPlayer` can play one stream at a time. If the player collects three coins in a quarter of a second, a single player cuts each sound off. The simplest solution is a **pool**: a small number of players, with each new sound using one that is free.

```
var pool: Array[AudioStreamPlayer] = []

func _ready() -> void:
    for i in range(6):
        var p := AudioStreamPlayer.new()     # make a player in code
        p.bus = &"SFX"
        add_child(p)                         # a node only works once it is in the tree
        pool.append(p)

func _free_player() -> AudioStreamPlayer:
    for p in pool:
        if not p.playing:
            return p
    return pool[0]                           # all busy: reuse the first
```

This is a **fragment** from the project's `main.gd`. A newer alternative is the player's `max_polyphony` property, which lets one node play several copies of its stream, but a pool also lets every sound have its own pitch and volume.

Fade music instead of cutting it

A sudden start or stop of music feels rough. Use a tween (Chapter 10) on `volume_db`:

```
music_player.volume_db = -40.0
music_player.play()
create_tween().tween_property(music_player, "volume_db", 0.0, 1.0)    # fade in over 1 second
```

To fade out, tween to -40 dB and then call `stop` with `tween_callback`.

Sound that comes from somewhere

An `AudioStreamPlayer2D` is the 2D version. As the node moves left or right of the listener, the sound pans; as it moves away, it gets quieter. By default the listener is the center of the current `Camera2D` or screen. Useful properties: `max_distance` (beyond this the sound is silent), `attenuation` (how quickly it fades with distance) and `panning_strength`.

In the project, a glowing dot slides back and forth with a looping hum. Close your eyes and listen: you can hear it travel.

Always give the player controls

Offer at least Master, Music and SFX volumes. Some players want the music off and the effects on. Provide a mute option that does not make them hunt through menus.

The browser needs a click first

Web browsers refuse to play sound until the user has interacted with the page (a click or key press). When you export a game to the web (Chapter 23), the first sound will start only after the first input. A "Click to start" screen solves this naturally.

Think about silence

Silence is a design tool. Constant music and effects numb the ear. Leave gaps, lower the music during important events (a technique called ducking), and let sounds breathe.

Project 11: A sound board with generated sounds

You will build a control panel that plays three generated sound effects, a looping music track with fade, volume sliders for each bus, a muffle effect, a sliding 2D hum and a mouse-controlled live synthesizer. It takes 45 to 75 minutes.

Project folder: `projects/ch11-sound/`

Files:

File	Purpose
`project.godot`, `icon.svg`	Project files
`default_bus_layout.tres`	The Music and SFX buses (made in the Audio panel)
`tone_maker.gd`	Generates every sound with code
`emitter.gd`	The sliding dot with a 2D hum
`main.tscn`, `main.gd`	The control panel

Step 1: Create the project and the buses

1. Create a project named `ch11-sound` (Compatibility renderer).
2. Set the window size to 960 by 540 (Project Settings > Display > Window).
3. Open the Audio panel and create the layout from section 11.4: a `Music` bus (volume -6 dB, with a **Low-Pass Filter** effect at 900 Hz *disabled*) and an `SFX` bus (with a **Reverb**, wet 0.15). Save the layout (default name `default_bus_layout.tres`).

Step 2: The sound generator

Create `tone_maker.gd` (**Create New > Script**, Inherits `RefCounted`) and paste:

`tone_maker.gd`

```gdscript
class_name ToneMaker
extends RefCounted
## Makes sound effects and a music loop out of pure maths. No audio files needed.
## Every function returns an AudioStreamWAV (16-bit, mono) that an AudioStreamPlayer can play.

const MIX_RATE := 22050  # samples per second


## One point of a wave. `phase` counts whole cycles; the result is between -1 and 1.
static func wave(kind: String, phase: float) -> float:
    var p := fposmod(phase, 1.0)
    match kind:
        "square":
            return 1.0 if p < 0.5 else -1.0
        "saw":
            return 2.0 * p - 1.0
        "triangle":
            return 4.0 * absf(p - 0.5) - 1.0
        _:
            return sin(TAU * p)


## Turns a list of numbers (-1.0 to 1.0) into a playable stream.
static func to_stream(samples: PackedFloat32Array, loop: bool = false) -> AudioStreamWAV:
    var data := PackedByteArray()
    data.resize(samples.size() * 2)  # 2 bytes per 16-bit sample
    for i in range(samples.size()):
        var v := int(clampf(samples[i], -1.0, 1.0) * 32767.0)
        data.encode_s16(i * 2, v)
    var stream := AudioStreamWAV.new()
    stream.format = AudioStreamWAV.FORMAT_16_BITS
    stream.mix_rate = MIX_RATE
    stream.stereo = false
    stream.data = data
    if loop:
        stream.loop_mode = AudioStreamWAV.LOOP_FORWARD
        stream.loop_begin = 0
        stream.loop_end = samples.size()
    return stream
```

```
## A steady note with a quick fade in and a fade out so it does not click.
static func tone(freq: float, seconds: float, kind: String = "sine", volume: float = 0.5, loop: bool =
false) -> AudioStreamWAV:
    var count := int(seconds * MIX_RATE)
    var samples := PackedFloat32Array()
    samples.resize(count)
    for i in range(count):
        var t := float(i) / MIX_RATE
        var env := 1.0
        if not loop:
            env = minf(t / 0.01, 1.0) * minf((seconds - t) / 0.05, 1.0)
        samples[i] = wave(kind, freq * t) * volume * env
    return to_stream(samples, loop)


## A note whose pitch slides from `f_start` to `f_end` (jump, laser, power-up sounds).
static func sweep(f_start: float, f_end: float, seconds: float, kind: String = "square", volume: float =
0.4) -> AudioStreamWAV:
    var count := int(seconds * MIX_RATE)
    var samples := PackedFloat32Array()
    samples.resize(count)
    var phase := 0.0
    for i in range(count):
        var t := float(i) / count  # 0.0 to 1.0 through the sound
        var freq := lerpf(f_start, f_end, t)
        phase += freq / MIX_RATE
        samples[i] = wave(kind, phase) * volume * (1.0 - t)
    return to_stream(samples)


## A burst of random noise that dies away (hits, explosions, footsteps).
static func noise_burst(seconds: float, volume: float = 0.5) -> AudioStreamWAV:
    var count := int(seconds * MIX_RATE)
    var samples := PackedFloat32Array()
    samples.resize(count)
    for i in range(count):
        var t := float(i) / count
        samples[i] = randf_range(-1.0, 1.0) * volume * pow(1.0 - t, 2.0)
    return to_stream(samples)


## The classic two-note coin "ding".
static func coin() -> AudioStreamWAV:
    var samples := PackedFloat32Array()
    _add_note(samples, 0.0, 988.0, 0.08, "square", 0.25, 8.0)    # B5
    _add_note(samples, 0.08, 1319.0, 0.30, "square", 0.25, 8.0)  # E6
    return to_stream(samples)


static func jump() -> AudioStreamWAV:
    return sweep(250.0, 700.0, 0.18, "square", 0.3)


static func hit() -> AudioStreamWAV:
    return noise_burst(0.25, 0.6)


## Frequency in Hz of a note counted in semitones from the A above middle C (440 Hz).
static func note_freq(semitones: int) -> float:
    return 440.0 * pow(2.0, semitones / 12.0)


## An 8-second looping tune: a lead melody over a slow bass.
static func music_loop() -> AudioStreamWAV:
    var beat := 0.5  # seconds per note
    var lead: Array[int] = [3, 7, 5, 3, 0, 3, -2, -5, 3, 7, 5, 3, 5, 3, 0, -2]
    var bass: Array[int] = [-21, -24, -21, -26]
    var samples := PackedFloat32Array()
    for i in range(lead.size()):
        _add_note(samples, i * beat, note_freq(lead[i]), beat, "square", 0.12, 3.0)
```

```
    for i in range(bass.size()):
        _add_note(samples, i * beat * 4.0, note_freq(bass[i]), beat * 4.0, "triangle", 0.3, 1.0)
    return to_stream(samples, true)


## Mixes one note into `samples` (making the array longer if needed).
## `decay` makes the note fade away faster when bigger.
static func _add_note(samples: PackedFloat32Array, start: float, freq: float, length: float, kind: String,
volume: float, decay: float) -> void:
    var first := int(start * MIX_RATE)
    var count := int(length * MIX_RATE)
    if samples.size() < first + count:
        samples.resize(first + count)  # new slots are filled with 0.0 (silence)
    for i in range(count):
        var t := float(i) / MIX_RATE
        var env := minf(t / 0.01, 1.0) * exp(-decay * t) * minf((length - t) / 0.02, 1.0)
        samples[first + i] += wave(kind, freq * t) * volume * env
```

Study these functions: `wave` (one sample of a shape), `to_stream` (numbers to a playable stream), `tone`, `sweep`, `noise_burst`, `coin`, `jump`, `hit`, `note_freq`, `music_loop`, and the helper `_add_note`. Everything is `static`, so the rest of the project calls them as `ToneMaker.coin()` with no node needed.

Step 3: The moving 2D sound

Create a scene with a `Node2D` root named `Emitter` and one child `AudioStreamPlayer2D` named `Hum`. Set Hum's **Bus** to `SFX`. Attach `emitter.gd` to the root. Save it as `emitter.tscn` if you wish to keep it separate; in the finished project it is simply part of the main scene.

`emitter.gd`

```
extends Node2D
## A glowing dot that slides left and right forever, with a looping hum attached.
## It shows how AudioStreamPlayer2D pans the sound as the dot moves.

@onready var hum: AudioStreamPlayer2D = $Hum

var humming: bool = false


func _ready() -> void:
    hum.stream = ToneMaker.tone(100.0, 1.0, "saw", 0.2, true)  # 100 whole cycles in 1 second: loops
cleanly
    position.x = 120.0
    var tween := create_tween().set_loops()  # no number = loop forever
    tween.tween_property(self, "position:x", 840.0,
3.0).set_trans(Tween.TRANS_SINE).set_ease(Tween.EASE_IN_OUT)
    tween.tween_property(self, "position:x", 120.0,
3.0).set_trans(Tween.TRANS_SINE).set_ease(Tween.EASE_IN_OUT)


func set_humming(on: bool) -> void:
    humming = on
    if on:
        hum.play()
    else:
        hum.stop()
    queue_redraw()


func _draw() -> void:
    var color := Color(1.0, 0.85, 0.3) if humming else Color(0.5, 0.5, 0.5)
    draw_circle(Vector2.ZERO, 14.0, color)
```

`set_loops()` with no number makes the tween repeat forever. The two steps move the dot to the right and back, each with `TRANS_SINE` and `EASE_IN_OUT` so it slows at the edges like a pendulum.

Step 4: The scene

Create the main scene with a `Control` root named `Main` (use the **Full Rect** layout preset so that it fills the window). Build this tree. Every node marked with `%` must have **Access as Unique Name** turned on (right-click the node in the Scene dock). That lets the script find it as `%Name` wherever it sits in the tree.

```
Main (Control)                              <- script: main.gd
├── Background (ColorRect)                  Full Rect, dark color, Mouse Filter: Ignore
├── Margin (MarginContainer)                Full Rect, margins 24/16/24/16
│   └── VBox (VBoxContainer)                separation 10
│       ├── Title (Label)
│       ├── SfxRow (HBoxContainer)
│       │   ├── %CoinButton (Button)        text "Coin"
│       │   ├── %JumpButton (Button)        text "Jump"
│       │   ├── %HitButton (Button)         text "Hit"
│       │   └── %ExportButton (Button)      text "Save sounds as .wav files"
│       ├── OptionRow (HBoxContainer)
│       │   ├── %PitchCheck (CheckButton)   text "Random pitch", Button Pressed on
│       │   ├── %MuffleCheck (CheckButton)  text "Muffle music"
│       │   ├── %BeeCheck (CheckButton)     text "Moving hum (2D sound)"
│       │   └── %ThereminCheck (CheckButton) text "Theremin (move the mouse)"
│       ├── %MusicButton (Button)           Toggle Mode on, text "Music: off"
│       ├── Sliders (GridContainer)         Columns 3
│       │   ├── MasterLabel, %MasterSlider, %MasterValue
│       │   ├── MusicLabel,  %MusicSlider,  %MusicValue
│       │   └── SfxLabel,    %SfxSlider,    %SfxValue
│       └── %StatusLabel (Label)            text "Click a button to hear a sound."
├── %Emitter (Node2D)                       position (120, 490), script emitter.gd
│   └── Hum (AudioStreamPlayer2D)           Bus: SFX
├── %MusicPlayer (AudioStreamPlayer)        Bus: Music
└── %Theremin (AudioStreamPlayer)           Bus: SFX, Stream: New AudioStreamGenerator
```

Details to set:

- Each `HSlider`: Min Value 0, Max Value 1, Step 0.01, Value 1, Custom Minimum Size x = 400, Size Flags Vertical = Shrink Center.
- The `AudioStreamGenerator` on `%Theremin`: **Mix Rate** 22050, **Buffer Length** 0.1.
- Labels "Master", "Music", "SFX" in the left column; value labels start with the text "100%".

> **Note:** The finished project folder contains this scene already, so if you get stuck you can open `main.tscn` and compare.

Step 5: The script

Attach `main.gd` to `Main`:

`main.gd`

```gdscript
extends Control
## Chapter 11 sound board: sound effects pool, music with fades, three buses, 2D sound, live generator.

const POOL_SIZE := 6
const SILENT_DB := -40.0

@onready var coin_button: Button = %CoinButton
@onready var jump_button: Button = %JumpButton
@onready var hit_button: Button = %HitButton
@onready var export_button: Button = %ExportButton
@onready var pitch_check: CheckButton = %PitchCheck
@onready var muffle_check: CheckButton = %MuffleCheck
@onready var bee_check: CheckButton = %BeeCheck
```

```
@onready var theremin_check: CheckButton = %ThereminCheck
@onready var music_button: Button = %MusicButton
@onready var status: Label = %StatusLabel
@onready var music_player: AudioStreamPlayer = %MusicPlayer
@onready var theremin: AudioStreamPlayer = %Theremin
@onready var emitter: Node2D = %Emitter

var sounds: Dictionary = {}                # name (String) -> AudioStreamWAV
var pool: Array[AudioStreamPlayer] = []    # reusable players for sound effects
var theremin_playback: AudioStreamGeneratorPlayback
var theremin_phase: float = 0.0
var music_tween: Tween


func _ready() -> void:
    sounds["coin"] = ToneMaker.coin()
    sounds["jump"] = ToneMaker.jump()
    sounds["hit"] = ToneMaker.hit()
    music_player.stream = ToneMaker.music_loop()

    for i in range(POOL_SIZE):
        var p := AudioStreamPlayer.new()
        p.bus = &"SFX"
        add_child(p)
        pool.append(p)

    coin_button.pressed.connect(play_sfx.bind("coin"))
    jump_button.pressed.connect(play_sfx.bind("jump"))
    hit_button.pressed.connect(play_sfx.bind("hit"))
    export_button.pressed.connect(_export_wavs)
    muffle_check.toggled.connect(_on_muffle_toggled)
    bee_check.toggled.connect(emitter.set_humming)
    theremin_check.toggled.connect(_on_theremin_toggled)
    music_button.toggled.connect(_on_music_toggled)

    _hook_slider(%MasterSlider, %MasterValue, "Master")
    _hook_slider(%MusicSlider, %MusicValue, "Music")
    _hook_slider(%SfxSlider, %SfxValue, "SFX")


## Connect one slider to one bus and show the starting position of the bus.
func _hook_slider(slider: HSlider, value_label: Label, bus_name: String) -> void:
    var bus := AudioServer.get_bus_index(bus_name)
    var linear := db_to_linear(AudioServer.get_bus_volume_db(bus))
    slider.set_value_no_signal(linear)
    value_label.text = "%d%%" % roundi(linear * 100.0)
    slider.value_changed.connect(_on_slider_changed.bind(value_label, bus_name))


func _on_slider_changed(value: float, value_label: Label, bus_name: String) -> void:
    var bus := AudioServer.get_bus_index(bus_name)
    AudioServer.set_bus_volume_db(bus, linear_to_db(value))  # sliders are linear, buses use dB
    AudioServer.set_bus_mute(bus, value < 0.01)
    value_label.text = "%d%%" % roundi(value * 100.0)


func play_sfx(sound_name: String) -> void:
    var player := _free_player()
    player.stream = sounds[sound_name]
    player.pitch_scale = randf_range(0.9, 1.1) if pitch_check.button_pressed else 1.0
    player.play()
    status.text = "Played '%s' (pitch %.2f)" % [sound_name, player.pitch_scale]


## Returns a player that is not busy; if all are busy, reuses the first one.
func _free_player() -> AudioStreamPlayer:
    for p in pool:
        if not p.playing:
            return p
    return pool[0]
```

```
func _on_music_toggled(on: bool) -> void:
    if music_tween:
        music_tween.kill()
    music_tween = create_tween()
    if on:
        music_button.text = "Music: on"
        music_player.volume_db = SILENT_DB
        music_player.play()
        music_tween.tween_property(music_player, "volume_db", 0.0, 1.0)   # fade in
    else:
        music_button.text = "Music: off"
        music_tween.tween_property(music_player, "volume_db", SILENT_DB, 1.0)  # fade out
        music_tween.tween_callback(music_player.stop)


func _on_muffle_toggled(on: bool) -> void:
    var bus := AudioServer.get_bus_index("Music")
    AudioServer.set_bus_effect_enabled(bus, 0, on)  # effect 0 on the Music bus is the low-pass filter


func _on_theremin_toggled(on: bool) -> void:
    if on:
        theremin.play()
        theremin_playback = theremin.get_stream_playback() as AudioStreamGeneratorPlayback
    else:
        theremin.stop()
        theremin_playback = null


func _process(_delta: float) -> void:
    if theremin_playback == null:
        return
    var generator := theremin.stream as AudioStreamGenerator
    var freq := remap(get_global_mouse_position().x, 0.0, size.x, 200.0, 800.0)
    var step := freq / generator.mix_rate
    # Fill whatever room is free in the generator's little buffer.
    for _i in range(theremin_playback.get_frames_available()):
        var sample := sin(theremin_phase * TAU) * 0.25
        theremin_playback.push_frame(Vector2(sample, sample))  # left, right
        theremin_phase = fposmod(theremin_phase + step, 1.0)


func _export_wavs() -> void:
    DirAccess.make_dir_recursive_absolute("user://sounds")
    for sound_name in sounds:
        var stream: AudioStreamWAV = sounds[sound_name]
        stream.save_to_wav("user://sounds/%s.wav" % sound_name)
    var music: AudioStreamWAV = music_player.stream
    music.save_to_wav("user://sounds/music.wav")
    status.text = "Saved to " + ProjectSettings.globalize_path("user://sounds")
```

Walk through it:

- `_ready` builds the sounds into a **dictionary** (Chapter 4) called `sounds`, assigns the music stream, creates the pool of six players, and connects every button and check box. `play_sfx.bind("coin")` connects a button to the function *with the sound's name already filled in*.
- `_hook_slider` links a slider and its value label to one bus, and starts the slider at the bus's current volume. `set_value_no_signal` sets the slider's position without triggering `value_changed`, so the bus volume is not set twice.
- `_on_slider_changed` converts the slider's 0 to 1 value with `linear_to_db`, sets the bus volume, and mutes the bus when the slider is nearly at zero.
- `play_sfx` picks a free player from the pool, sets a random pitch if the check box is on, and plays.
- `_on_music_toggled` fades the music in or out with a tween. `music_tween.kill()` prevents two fades from fighting (Chapter 10).
- `_on_muffle_toggled` turns the low-pass filter (effect 0 on the Music bus) on and off.

- `_process` fills the generator buffer for the theremin while it is turned on.
- `_export_wavs` writes each sound to `user://sounds/` with `save_to_wav`.

Step 6: Test

Press **F5** and check:

1. Click **Coin**, **Jump**, **Hit**: each sound plays and a line in the status label says which. With **Random pitch** on, pressing a button repeatedly gives slightly different sounds. Turn it off and they are identical.
2. Click **Music: off**. The tune fades in over a second and loops with no click. Click again to fade it out.
3. Drag the **Music** slider down: only the music gets quieter. Drag **SFX**: only effects change (reverb included). Drag **Master**: everything.
4. While music plays, tick **Muffle music**: it sounds like it is behind a wall. Untick to restore.
5. Tick **Moving hum**: a buzz moves from left to right in your headphones as the dot slides, and the dot turns yellow while the sound is on.
6. Tick **Theremin** and move the mouse left and right: pitch follows the mouse.
7. Click **Save sounds as .wav files**: the status label shows the folder. Open it in your file explorer and play the WAV files in any player.

Expected result

A dark window with a row of buttons, three volume sliders, several check boxes and a glowing dot at the bottom edge. Every sound heard comes from code; the game folder contains no audio files.

Chapter summary

- Digital sound is a list of samples taken many times a second. Frequency sets the pitch, amplitude sets the loudness.
- Use WAV for short effects and Ogg Vorbis for music. Looping is a property of the stream and is set in the Import dock.
- `AudioStreamPlayer` plays a stream through a bus. `AudioStreamPlayer2D` pans and fades with position.
- Volume is measured in decibels: 0 is unchanged, about -6 is half, and sliders should convert with `linear_to_db`.
- Buses mix sounds, apply effects (reverb, low-pass, compressor), and send to Master. Create them in the Audio panel; control them at run time with `AudioServer`.
- A sound can be computed: fill a `PackedByteArray` with 16-bit samples and wrap it in an `AudioStreamWAV`. Use envelopes to avoid clicks. Mixing is adding.
- `AudioStreamGenerator` plays sound computed live, one frame at a time.
- Good audio habits: vary pitch, use a pool for overlapping sounds, fade music, offer volume controls, and remember that browsers need a click before sound starts.

Review questions

1. **Short answer.** What is a sample, and what does the sample rate tell you?

2. **Multiple choice.** Which format is the best fit for a 0.2 second jump sound that must play instantly? a) Ogg Vorbis b) WAV c) A `.txt` file d) MP3

3. **True or false.** A volume of -6 dB is about half the amplitude of 0 dB, and 0 dB means the sound is unchanged.

4. **Short answer.** A designer wants separate music and effects volume sliders in an options menu. What Godot feature makes this easy, and why?

5. **Multiple choice.** Which line converts a slider value `v` between 0 and 1 into decibels? a) `v * 80` b) `linear_to_db(v)` c) `db_to_linear(v)` d) `sqrt(v)`

6. **Short answer.** Why does a generated note start with a short fade-in and end with a short fade-out?

7. **Multiple choice.** A tone needs to be exactly one octave above 440 Hz. Which frequency is it? a) 441 Hz b) 550 Hz c) 880 Hz d) 660 Hz

8. **Short answer.** The player picks up three coins very quickly, but you hear only one coin sound cut off each time. What is the problem, and what is a simple fix?

9. **True or false.** `AudioStreamGenerator` needs the whole sound to be computed before `play()` is called.
10. **Short answer.** Name two things that give an `AudioStreamPlayer2D` a different sound from a plain `AudioStreamPlayer`.

Exercises

1. **Practice (easy).** In `ToneMaker`, add a function `static func beep() -> AudioStreamWAV` that returns `tone(880.0, 0.1, "sine", 0.4)`. Add a fourth button "Beep" to the scene and connect it. Hear how a pure sine differs from the square-wave coin.
2. **Practice (easy).** Change the melody array in `music_loop` to a different list of semitone numbers (keep 16 entries). Which numbers sound good together? Try only numbers from the list `[-5, -2, 0, 3, 5, 7, 10]`, a *pentatonic* set that is hard to get wrong.
3. **Apply (medium).** Add a **Power-up** sound: a sweep from 300 Hz up to 1200 Hz over 0.4 seconds with a triangle wave. Add a button for it. Then try the reverse for a "power-down".
4. **Apply (medium).** Add a **Mute** check box that mutes the Master bus with `AudioServer.set_bus_mute`. Make the Master slider and the check box not fight each other.
5. **Challenge (hard).** Write `ToneMaker.explosion()` that combines a noise burst with a low sine sweep from 150 Hz down to 40 Hz (both lasting about 0.6 seconds, mixed by adding samples). Then add *ducking*: while the explosion plays, lower the Music bus by 8 dB with a tween and bring it back after 0.8 seconds.

Common mistakes

- **Symptom:** No sound at all. **Cause:** The player has no stream, `play()` was never called, the bus is muted, or the volume is -80 dB. **Fix:** Check the stream, `autoplay`, the Audio panel's mute buttons and `volume_db`.
- **Symptom:** The sound is silent only for some players. **Cause:** The player's `bus` property names a bus that does not exist, or the bus is muted by a slider. **Fix:** Spell the bus name exactly (it is case sensitive), and check it in the Audio panel.
- **Symptom:** A click or pop when a sound starts, ends, or loops. **Cause:** The wave begins or ends at a nonzero value. **Fix:** Add a short fade-in and fade-out (an envelope). For loops, make the loop an exact number of cycles or beats.
- **Symptom:** The sound is distorted and harsh. **Cause:** Samples went beyond -1.0 to 1.0 (mixing too many voices) or the bus volume is above 0 dB. **Fix:** Lower each voice's volume, or add a Limiter or Compressor effect on the bus.

- **Symptom:** The loop setting has no effect on a sound. **Cause:** The loop mode was set on the player, not the stream, or the file was not reimported. **Fix:** Set the loop in the Import dock and click Reimport.
- **Symptom:** A generated sound is much higher, lower or faster than expected. **Cause:** `mix_rate` of the stream does not match the rate used to compute the samples. **Fix:** Use one `MIX_RATE` constant for both the maths and the stream.

Further reading

- Audio buses: https://docs.godotengine.org/en/stable/tutorials/audio/audio_buses.html
- Audio streams: https://docs.godotengine.org/en/stable/tutorials/audio/audio_streams.html
- Recording and procedural audio (AudioStreamGenerator): https://docs.godotengine.org/en/stable/tutorials/audio/sync_with_audio.html
- AudioStreamPlayer class reference: https://docs.godotengine.org/en/stable/classes/class_audiostreamplayer.html
- AudioServer class reference: https://docs.godotengine.org/en/stable/classes/class_audioserver.html
- AudioStreamWAV class reference: https://docs.godotengine.org/en/stable/classes/class_audiostreamwav.html

Chapter 12: Tilemaps and Level Building

Learning objectives

By the end of this chapter you will be able to:

- Explain what a tile map is and convert between pixel positions and grid cells.
- Create a `TileSet` from a tile atlas image and paint a level with `TileMapLayer` nodes.
- Stack several layers (ground, walls) and give tiles collision shapes so characters cannot walk through them.
- Attach your own information to tiles with custom data, and read it from a script.
- Set up a terrain so that Godot picks the correct edge and corner tile automatically while you paint.
- Build and change a level from code with `set_cell`, `local_to_map` and `set_cells_terrain_connect`.

Key terms

- **Tile**: a small picture, usually a square, that is repeated across a level.
- **Tile atlas**: one image holding many tiles in a grid (a sprite sheet for tiles).
- **TileSet**: a resource that describes the tiles you can paint with: pictures, collision shapes, custom data and terrain rules.
- **TileMapLayer**: a node that stores which tile sits in which grid cell and draws them. Each layer holds one grid.
- **Cell**: one square of the grid, addressed by whole-number coordinates such as `(3, 5)`.
- **Atlas coordinates**: the position of a tile inside the atlas image, counted in tiles (not pixels).
- **Physics layer (on a TileSet)**: a slot that lets tiles carry collision shapes.
- **Custom data layer**: a named value, of a type you choose, that every tile can carry (for example `speed_factor`).
- **Terrain**: a rule set that lets Godot choose the right tile from a family (edges, corners) automatically.
- **Peering bits**: the marks on a tile that say which of its neighbors belong to the same terrain.
- **Autotiling**: choosing a tile based on what is next to it, so that paths and walls join up.

12.1 Why tile maps?

Think of a level in a classic 2D game: a dungeon, a field, a platformer world. If you built it by placing hundreds of `Sprite2D` nodes by hand, your scene would be a mess, slow to edit, slow to run, and impossible to change quickly. But those levels are made of a handful of repeating pieces: a grass square, a wall square, a water square. A **tile map** exploits that. You make a small set of tile pictures once, then fill a grid with them, the way you would fill graph paper with colored squares.

Advantages:

- **Fast to build**: you paint with a brush instead of placing nodes.
- **Fast to run**: Godot draws thousands of tiles as a few batches, not thousands of nodes.
- **Small to store**: a level is just a list of "cell → tile."
- **Easy to change**: swap the tile art, and every use changes at once.
- **Collision built in**: a tile can carry its own collision shape.

The grid and its two coordinate systems

A tile map lives on a grid of **cells**. Every cell has integer coordinates `(x, y)`. As in the rest of Godot 2D, `x` grows to the right and `y` grows **down**. The top-left cell is `(0, 0)`.

The same point can be described in two ways:

System	Unit	Example
Cell (map) coordinates	Whole cells	`(3, 2)`
Pixel (local) coordinates	Pixels, relative to the layer node	`(112.0, 80.0)`

With 32-pixel tiles, the cell `(3, 2)` covers pixels 96 to 128 across and 64 to 96 down. Its center is at `(112, 80)`.

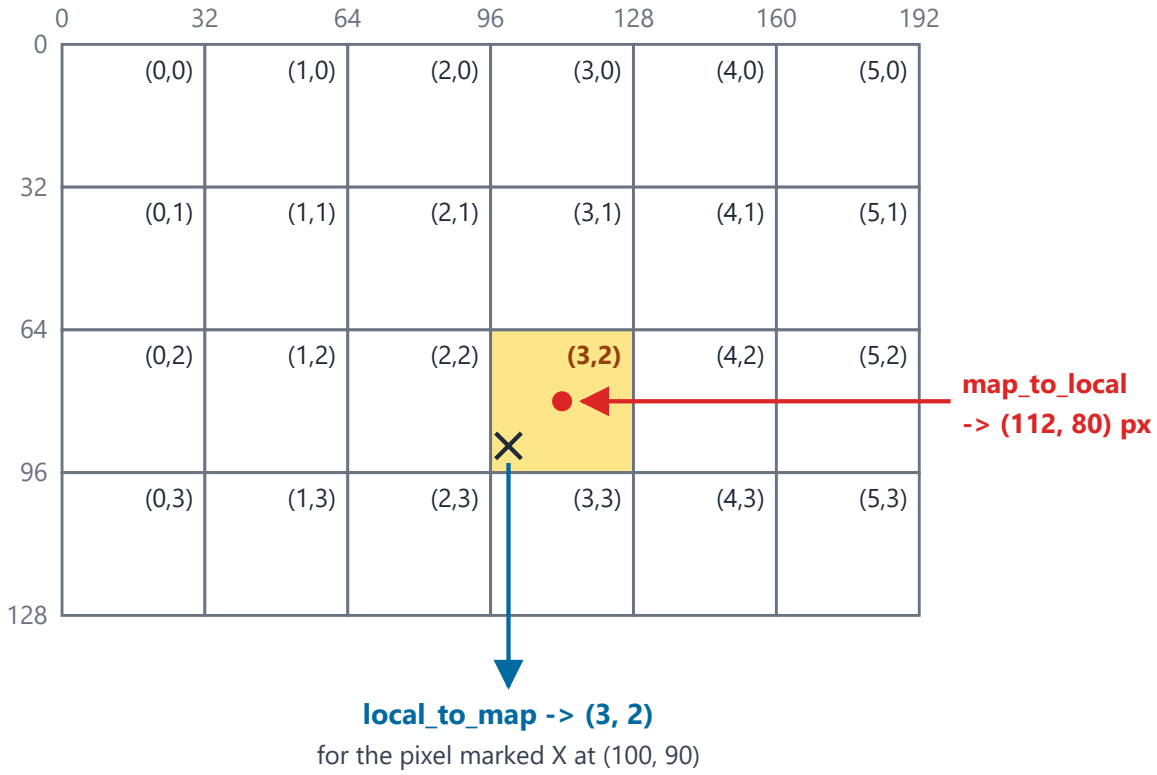


Figure 12.1: A pixel position and the cell that contains it. `local_to_map` converts pixels to a cell, and `map_to_local` converts a cell to the pixel position of its center.

Two functions of `TileMapLayer` do the conversion, and you will use them constantly:

```
var cell: Vector2i = layer.local_to_map(Vector2(112, 80))   # pixels -> cell, gives (3, 2)
var center: Vector2 = layer.map_to_local(Vector2i(3, 2))    # cell -> pixels, gives (112, 80)
```

Note the types. A `Vector2i` is a vector of two *integers*, used for cells (you cannot have cell 3.5). A `Vector2` holds two decimal numbers, used for pixel positions. The `i` stands for integer.

> **Note:** In Godot 4.0 to 4.2 level grids were made with a node called `TileMap`. Since Godot 4.3 the recommended node is `TileMapLayer`, with one node per layer. The old `TileMap` is deprecated. Everything in this chapter uses `TileMapLayer`. If a tutorial you find uses `TileMap`, the ideas are the same but the function names have an extra layer number argument.

12.2 The TileSet: your box of tiles

A `TileMapLayer` does not hold pictures itself. It points to a `TileSet` resource, which holds all the tiles you may use. Many layers can share one `TileSet`.

The tile atlas

The standard way to give a `TileSet` its pictures is an **atlas**: one image with all tiles in a grid, all the same size. Our project uses a 256 by 128 pixel atlas with 8 by 4 tiles of 32 by 32 pixels.

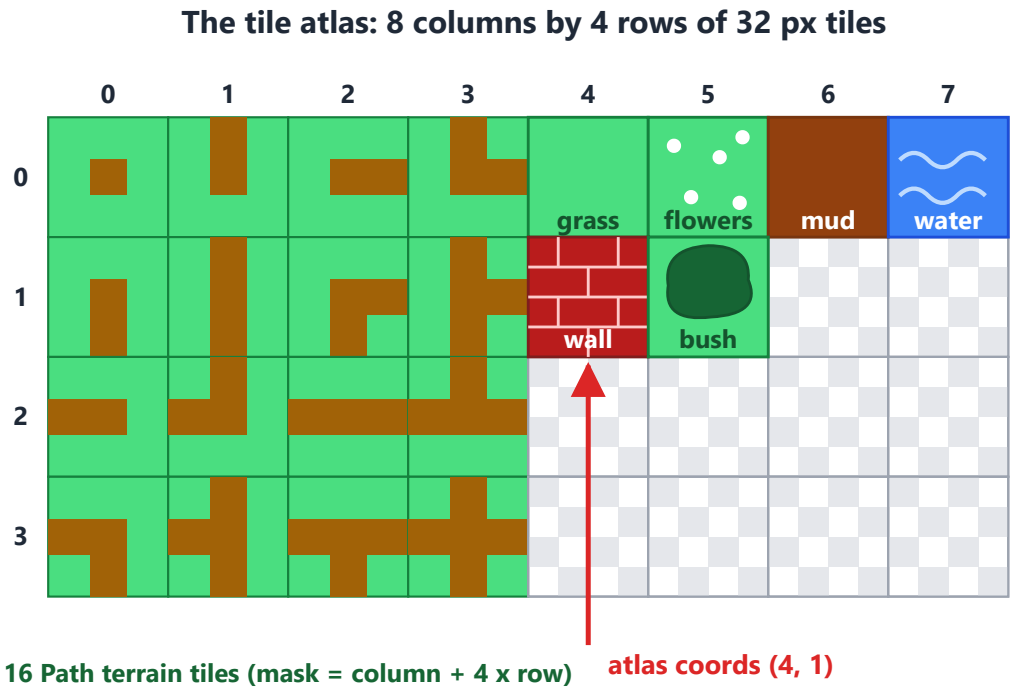


*Figure 12.2: The project's atlas. Position (4, 1) means column 4, row 1, counted in tiles: this is the **atlas coordinates** of the brick wall.*

Rule 9 says no outside art. So the project draws this atlas **with a script**. The tool `tools/make_atlas.gd` uses `Image` functions to paint colored rectangles into one image and saves it as a PNG file. You will read it in the project section. The result is a normal PNG file, just like one you would download, and from there on everything in the chapter works the same with real art.

> **Note:** If you want real tiles, Kenney.nl has many free tile packs, and itch.io has more. Check the licence of each. Appendix F lists places to look. Some packs have gaps between the tiles (a **separation**) or a border around the whole sheet (a **margin**); the TileSet's atlas settings have fields for both.

Making a TileSet in the editor

1. Add a **TileMapLayer** node to a scene.
2. In the Inspector, click **Tile Set** and choose **New TileSet**.
3. Click the new resource to expand it. Set **Tile Size** to the size of one tile: `32, 32`. (The default is 16 by 16. If your tile size is wrong, tiles appear cut up and misaligned. This is the most common beginner mistake.)
4. Open the **TileSet** panel at the bottom of the editor.
5. Drag your atlas PNG into the panel's left area. Godot asks, "Automatically create tiles in the atlas?" Click **Yes**. Every non-empty tile becomes a tile.

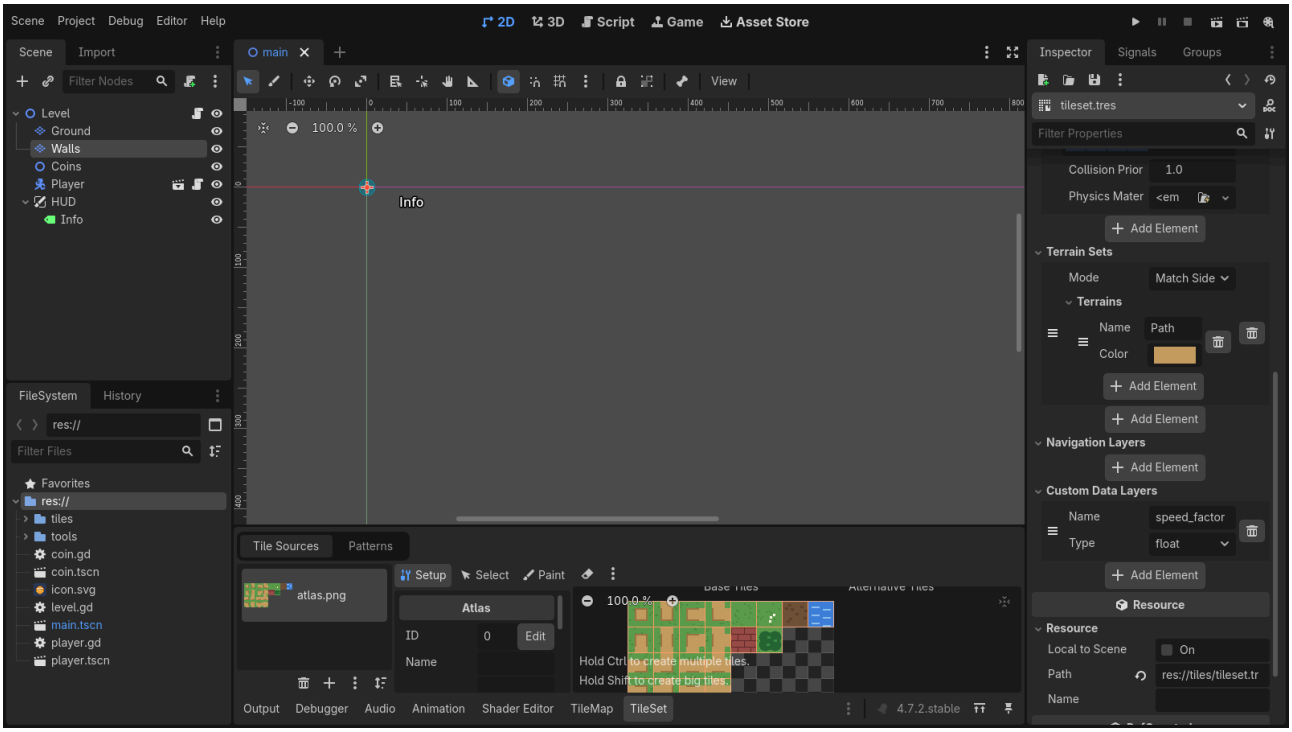


Figure 12.3: The TileSet panel. The Setup tab shows the atlas, the Paint tab is where you edit tile properties.

The TileSet resource in the Inspector has groups of settings. These are the ones we use:

Group	What it holds
Rendering	Texture settings and offsets
Physics Layers	Collision layers and masks that tiles can use
Terrain Sets	Rules for autotiling (section 12.5)
Navigation Layers	Where enemies may walk (Chapter 21)
Custom Data Layers	Your own values on tiles

Tiles, sources and alternatives

Three words that show up in the code:

- A **source** is one atlas image inside the TileSet, identified by a **source id** (a number). Our TileSet has one source, id `0`.
- A **tile** inside a source is identified by its **atlas coordinates**, for example `Vector2i(4, 1)`.
- A tile may also have **alternatives**: variations (for instance mirrored) that share the same picture. The default alternative number is `0`, and most of the time you never think about it.

So, to name any tile in code, you need three things: the source id, the atlas coordinates, and (almost always zero) the alternative. You will see this in `set_cell`.

12.3 Painting levels

The TileMap panel

Select a `TileMapLayer` and click the **TileMap** tab of the bottom panel. (The same area has the TileSet tab; the two tabs switch between *editing the tile box* and *painting with it*.) You see the tile atlas and a row of tools:

Tool	Use
Select	Select cells to move, copy or delete
Paint	Click or drag to paint the chosen tile
Line	Drag to paint a straight line
Rectangle	Drag to fill (or outline) a rectangle
Bucket Fill	Fill a connected area with a tile
Picker	Click a painted cell to pick its tile
Eraser	Remove tiles

Select a tile in the atlas view and click or drag in the 2D viewport. Right-click erases. You can hold **Shift** for a straight line and **Ctrl+Shift** for filled shapes, depending on the tool. There are also **random** painting (scatter several tiles) and **patterns** (save a hand-built group of tiles for reuse); both are small buttons in the toolbar.

> **Try it:** Create a new scene with a `TileMapLayer`, make a `TileSet` from any small image (even `res://icon.svg`, with tile size matching), and paint a smiley by hand. This takes five minutes and teaches the tools.

Layers

One layer can hold only one tile per cell. A tree on top of grass needs two tiles at the same place, so we use **two layers**: a `Ground` layer with the grass and a `Walls` layer on top with the tree. Layers are separate nodes, drawn in the order they appear in the Scene dock: lower in the list means drawn on top.

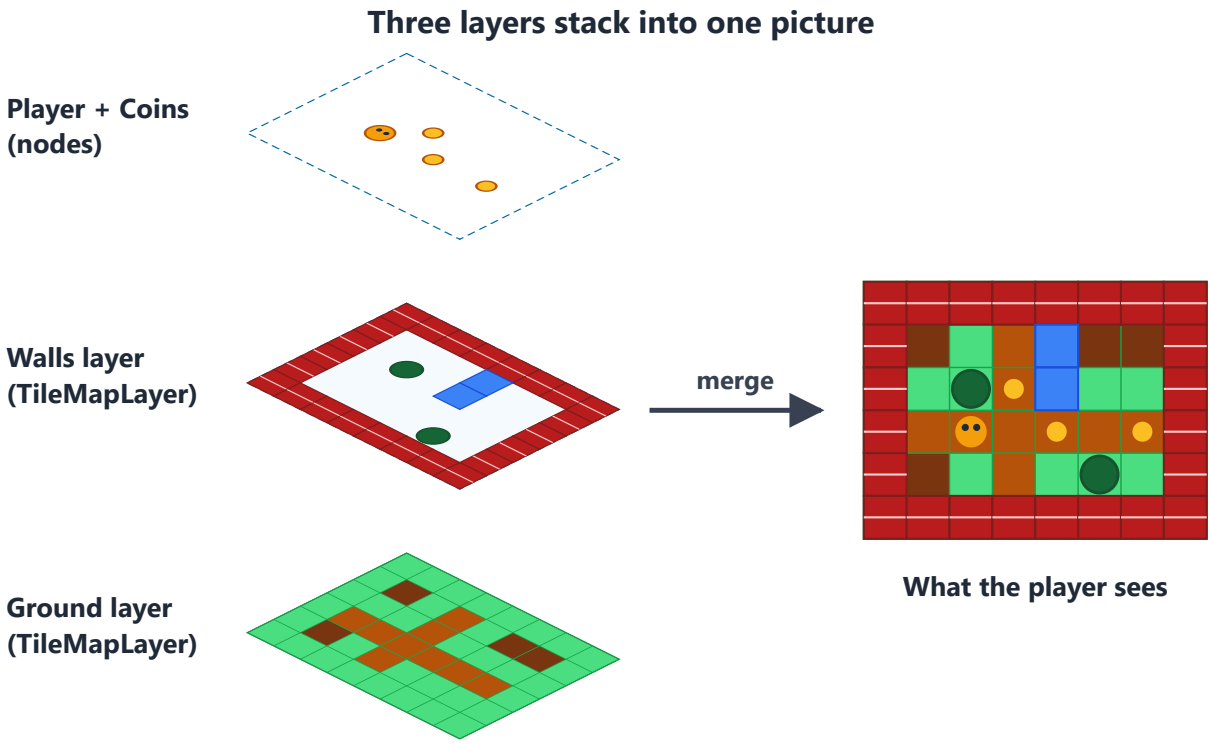


Figure 12.4: Layers are transparent sheets. Ground at the bottom, walls above, then the player and the coins.

Typical layers for a top-down game: `Ground` (floor and paths), `Walls` (blocking things), `Decoration` (flowers, cracks) drawn under the characters, and sometimes `Above` (tree tops) drawn over the characters. Our project uses `Ground` and `Walls`.

Why separate collision from ground? Because you can give `Walls` a physics layer and leave `Ground` without any collision, which keeps floors cheap and clean. Also, in code, "what is on the ground here?" and "is there something blocking here?" become two simple questions.

12.4 Tiles with collision and with custom data

Collision shapes on tiles

For a character not to walk through walls, each wall tile needs a collision shape. The TileSet has to know *which* physics layer to put the shape on. Steps in the editor:

1. Select the TileSet resource. Open **Physics Layers** and click **Add Element**. A **Physics Layer 0** appears with a **Collision Layer** and **Collision Mask**. Leave layer 1 on. (Chapter 8 explained layers and masks: a body collides with things on layers that are in its mask.)
2. In the TileSet panel choose the **Paint** tab. Under **Paint Properties** pick **Physics Layer 0**.
3. Select the wall tile, and draw the shape on it in the editor: for a full square, click its four corners (or use the default full-tile shape).
4. Repeat for every solid tile (water, bush).

Now any `CharacterBody2D` whose collision mask includes layer 1 stops at those tiles. On the layer node itself, the property **Collision Enabled** must be on (it is, by default).

The code version, from `tools/make_tileset.gd`:

```
data.add_collision_polygon(0)              # add a polygon to physics layer 0
data.set_collision_polygon_points(0, 0, PackedVector2Array([
    Vector2(-16, -16), Vector2(16, -16), Vector2(16, 16), Vector2(-16, 16)]))
```

Points are measured from the *middle* of the tile, so a 32-pixel square runs from -16 to 16.

> **Warning:** The player must be on a collision layer/mask combination that matches the TileSet's physics layer. If Pip walks through walls, check that his collision mask includes layer 1 and that the TileSet's Physics Layer 0 has Collision Layer 1 on.

Custom data: your own information on tiles

A tile can carry a value of your choosing. Suppose mud should slow the player and a path should speed him up. Instead of listing special cases in the code ("if the tile is at (6, 0)..."), store the answer *on the tile*:

1. In the TileSet, under **Custom Data Layers**, click **Add Element**. Name it `speed_factor` and set its **Type** to `float`.
2. In the TileSet panel's **Paint** tab choose **Custom Data > speed_factor** and set the value for each tile: 0.5 for mud, 1.25 for path, 1.0 for everything else.

In code, you ask the layer for the tile data of a cell and read the value by name:

```
var cell: Vector2i = ground.local_to_map(ground.to_local(global_position))
var data: TileData = ground.get_cell_tile_data(cell)    # null if the cell is empty
if data:
    var factor: float = data.get_custom_data("speed_factor")
```

`to_local` converts a global position into the layer's own coordinates, which `local_to_map` expects. `get_cell_tile_data` returns a `TileData` object (or `null`, meaning "nothing there"), and the `if data:` check protects you from reading a missing tile.

This idea scales far: custom data for "damage per second" on lava tiles, "footstep sound" names for different floors (Chapter 11!), "is_door", "material", or any other level knowledge. The art and the rules live together in the TileSet, and the code stays short.

12.5 Terrains: tiles that connect themselves

The problem

A path made of single tiles looks fine as a straight line. But a path that turns, branches or ends needs different pictures: a corner piece, a T-junction, a dead end. Choosing the right tile for each cell by hand is slow, and changing a path means re-choosing all its neighbors. Godot's **terrain** system does this for you. You say "this cell is path" and Godot picks the tile that connects to the surrounding path cells.

Peering bits

Each tile in a terrain declares which of its *neighbors* are the same terrain. These marks are **peering bits**. How many bits a tile has depends on the terrain **mode**:

Mode	Neighbors considered	Number of tiles for a full set
Match Sides	4 sides (up, right, down, left)	16
Match Corners	4 corners	16
Match Corners and Sides	4 sides and 4 corners	47 (the "blob" set)

Our project uses **Match Sides**: only the four sides matter. Four yes/no sides give 2 × 2 × 2 × 2 = 16 combinations, so a complete set has 16 tiles. A handy way to count: give each side a number (top 1, right 2, bottom 4, left 8) and add up the sides that connect. The result, from 0 to 15, is the tile's **mask**. We even arrange the atlas so that the tile with mask *m* sits at column *m mod 4*, row *m div 4*.

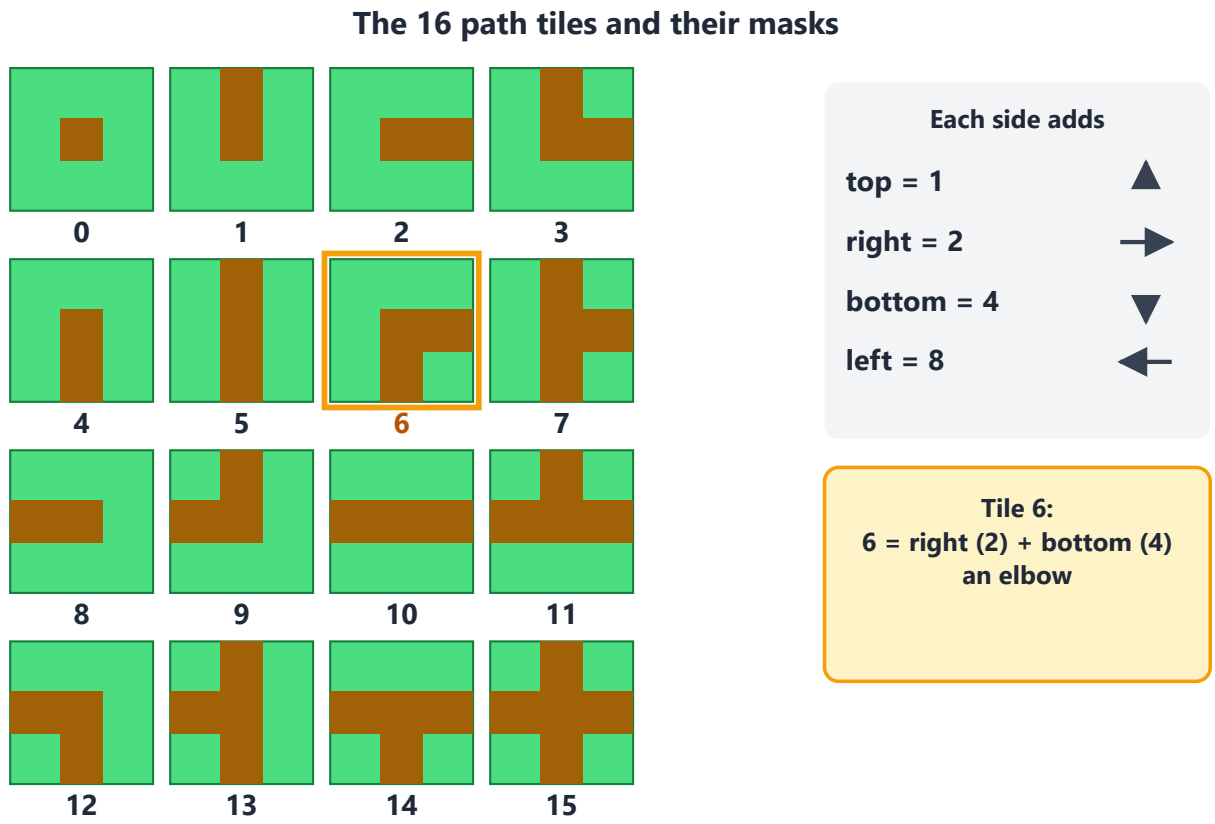


Figure 12.5: The 16 tiles of a Match Sides terrain. The number is the sum of the connected sides. Tile 6 connects right (2) and bottom (4): an elbow.

Setting it up in the editor

1. In the TileSet resource open **Terrain Sets**, **Add Element**, and set **Mode** to *Match Sides*. Under it add a terrain, and name it `Path` (give it a color).
2. In the TileSet panel's **Paint** tab choose **Terrain Set 0**, then **Terrain** > Path.
3. For each of the 16 tiles, set its terrain to `Path` and click the sides that connect to other path tiles.
4. In the TileMap panel, switch to the **Terrains** tab. Pick `Path` and paint. Godot chooses the right tile for every cell, and fixes the neighbors as you draw.

The code version

Setup, from `tools/make_tileset.gd`:

```
tileset.add_terrain_set()
tileset.set_terrain_set_mode(0, TileSet.TERRAIN_MODE_MATCH_SIDES)
tileset.add_terrain(0)
tileset.set_terrain_name(0, 0, "Path")

# for each path tile:
data.terrain_set = 0
data.terrain = 0
if mask & 1:
    data.set_terrain_peering_bit(TileSet.CELL_NEIGHBOR_TOP_SIDE, 0)
if mask & 2:
    data.set_terrain_peering_bit(TileSet.CELL_NEIGHBOR_RIGHT_SIDE, 0)
if mask & 4:
    data.set_terrain_peering_bit(TileSet.CELL_NEIGHBOR_BOTTOM_SIDE, 0)
if mask & 8:
    data.set_terrain_peering_bit(TileSet.CELL_NEIGHBOR_LEFT_SIDE, 0)
```

This is a **fragment** from the tool script (shown in full in the project). `mask & 1` is a **bitwise AND**: it asks "is the 1 bit set in `mask`?" and so yields a nonzero value (true) when the top side is in the mask. You do not need to master bit operations; read it as "if this tile's mask includes the top side."

Painting with terrains in code is one call:

```
var cells: Array[Vector2i] = [Vector2i(3, 1), Vector2i(4, 1), Vector2i(5, 1)]
ground.set_cells_terrain_connect(cells, 0, 0)     # cells, terrain set, terrain
```

Godot places a tile in each of those cells, choosing the right one by looking at which of them (and which existing cells) are path. The tiles it picks for the first and last cell will have an arm only in one direction.

> **Why it works:** Terrain painting is a puzzle solver. For each cell, it looks at every neighbor's terrain and finds a tile whose peering bits match all of them. That is why one painted cell can change its neighbors: they must re-fit.

12.6 Building and changing levels in code

Everything you can do with the brush, you can do in code. This lets you generate random dungeons, load levels from text files, and let the player build and destroy the world.

The core functions

Call	Meaning
`layer.set_cell(coords, source_id, atlas_coords)`	Put a tile in a cell. The alternative defaults to 0.
`layer.erase_cell(coords)`	Remove the tile from a cell.
`layer.clear()`	Remove every tile.
`layer.get_cell_source_id(coords)`	The source id of the cell, or **-1** if the cell is empty.
`layer.get_cell_atlas_coords(coords)`	Which tile (in the atlas) fills the cell.
`layer.get_cell_tile_data(coords)`	The `TileData` of the cell, or `null`.
`layer.get_used_cells()`	An array with every filled cell.
`layer.get_used_rect()`	A `Rect2i` that tightly surrounds the filled cells.
`layer.local_to_map(pixel)` / `map_to_local(cell)`	Convert between pixels and cells.

Call	Meaning
`layer.set_cells_terrain_connect(cells, set, terrain)`	Paint a terrain over several cells.

`set_cell` example:

```
const SOURCE_ID := 0
const GRASS := Vector2i(4, 0)

func plant_grass(layer: TileMapLayer, cell: Vector2i) -> void:
    layer.set_cell(cell, SOURCE_ID, GRASS)
```

A level in a text map

Here is the method our project uses. Draw the level as text, one character per cell, then loop over it.

```
const MAP: Array[String] = [
    "#####",
    "#@.c#",
    "#####",
]
```

A **fragment** to show the pattern: each string is a row (so its index is `y`), each character a cell (its index is `x`). Characters get meanings: `#` is wall, `.` is grass, `c` a coin, `@` the start.

```
for y in range(MAP.size()):
    var line: String = MAP[y]
    for x in range(line.length()):
        var cell := Vector2i(x, y)
        var symbol: String = line[x]
        match symbol:
            "#":
                walls.set_cell(cell, SOURCE_ID, WALL)
            "c":
                add_coin(ground.map_to_local(cell))
```

A **nested loop**, a loop inside a loop (Chapter 4): the outer one visits each row and the inner one each character in the row. Level designers love this: you can see the whole level in your text editor, copy it, change it, and even generate it with other code.

Letting the player edit the world

`get_local_mouse_position()` gives the mouse in the layer's coordinates, and `local_to_map` turns it into a cell:

```
func _unhandled_input(event: InputEvent) -> void:
    if event is InputEventMouseButton and event.pressed:
        var cell: Vector2i = ground.local_to_map(ground.get_local_mouse_position())
        ground.set_cells_terrain_connect([cell], 0, 0)
```

That is a complete "click to lay path" feature in four lines. Removing a path piece needs a small extra step, because neighbors that pointed at the removed cell must be re-fitted; the project's `erase_path` does this and explains it in the code.

Other TileSet features, briefly

- **Navigation layers** mark which tiles enemies can walk on, so that pathfinding follows your map (Chapter 21).
- **Occlusion layers** let tiles cast shadows for 2D lights.
- **Scene collections** let you paint whole scenes (a chest, a torch with an animation) as tiles. Useful when a "tile" needs to be a real node with a script.
- **Y-sorting** on a layer draws tiles lower on the screen in front of higher ones, which gives trees and pillars a sense of depth in top-down games. Enable *Y Sort Enabled* on the layer.
- **Rendering quadrants**: for speed Godot groups nearby tiles into chunks when drawing. You do not have to do anything, but if you see a note about *Rendering Quadrant Size* it is this.

Performance

Tile maps are fast. A layer of 100,000 cells is fine. The cost to watch is *collision shapes*: every solid tile adds a shape to the physics world. Godot merges neighboring tiles' shapes when it can, but huge levels full of tiny shapes can slow physics. For giant open worlds, keep solid tiles only where they matter.

Project 12: A walkable level

You will make a small top-down level: grass, mud that slows you down, water and walls that block you, a path network you can extend with the mouse, and coins to collect. It takes 60 to 90 minutes.

Project folder: `projects/ch12-tilemaps/`

Files:

File	Purpose
`project.godot`, `icon.svg`	Project files
`tools/make_atlas.gd`	Draws the atlas PNG with code
`tools/make_tileset.gd`	Builds the TileSet resource from the atlas
`tiles/atlas.png`, `tiles/tileset.tres`	The two generated files (already in the folder)
`player.tscn`, `player.gd`	Pip, seen from above
`coin.tscn`, `coin.gd`	A collectible `Area2D`
`main.tscn`, `level.gd`	The level, built from a text map

Step 1: Create the project

Create a project `ch12-tilemaps` with the Compatibility renderer. In Project Settings set the window to 960 by 540 and **Rendering > Textures > Default Texture Filter** to *Nearest*. Create the folders `tools` and `tiles`.

Step 2: Draw the atlas with a script

The first tool draws the image. Create `tools/make_atlas.gd` with this content. Because it is a **tool script** that is run from the command line, it extends `SceneTree` instead of a node: Godot runs it, runs `_init`, and then it calls `quit()` to stop.

`tools/make_atlas.gd`

```
extends SceneTree
## Tool script: draws the tile atlas (res://tiles/atlas.png) with code.
## Run it from the project folder:
##   godot --headless --path . --script res://tools/make_atlas.gd
##
## Layout (each tile is 32 x 32 pixels, the sheet is 8 tiles wide and 4 tall):
##   x 0-3, y 0-3 : the 16 "Path" terrain tiles (index = top 1 + right 2 + bottom 4 + left 8)
##   (4,0) grass   (5,0) grass with flowers   (6,0) mud   (7,0) water
##   (4,1) brick wall   (5,1) bush

const TILE := 32
const GRASS := Color(0.36, 0.62, 0.30)
const GRASS_DARK := Color(0.30, 0.54, 0.25)
const DIRT := Color(0.78, 0.62, 0.38)
const DIRT_EDGE := Color(0.62, 0.47, 0.27)
const MUD := Color(0.45, 0.33, 0.22)
const WATER := Color(0.25, 0.50, 0.85)
const WATER_LIGHT := Color(0.55, 0.75, 0.95)
const BRICK := Color(0.60, 0.30, 0.28)
const BRICK_LINE := Color(0.38, 0.17, 0.17)
const BUSH := Color(0.15, 0.40, 0.20)
const BUSH_LIGHT := Color(0.25, 0.55, 0.28)


func _init() -> void:
    var img := Image.create(TILE * 8, TILE * 4, false, Image.FORMAT_RGBA8)
    img.fill(Color(0, 0, 0, 0))
    var rng := RandomNumberGenerator.new()
    rng.seed = 12345  # the same seed always gives the same picture

    # 16 path tiles. Bit 1 = top, 2 = right, 4 = bottom, 8 = left.
    for mask in range(16):
        var cell := Vector2i(mask % 4, floori(mask / 4.0))
        _grass(img, cell, rng, false)
        _path(img, cell, mask)

    _grass(img, Vector2i(4, 0), rng, false)
    _grass(img, Vector2i(5, 0), rng, true)
    _mud(img, Vector2i(6, 0), rng)
    _water(img, Vector2i(7, 0))
    _wall(img, Vector2i(4, 1))
    _bush(img, Vector2i(5, 1), rng)

    DirAccess.make_dir_recursive_absolute("res://tiles")
    var err := img.save_png("res://tiles/atlas.png")
    print("Saved atlas.png, result code: ", err, " (0 means OK)")
    quit()


func _rect(img: Image, cell: Vector2i, x: int, y: int, w: int, h: int, color: Color) -> void:
    img.fill_rect(Rect2i(cell.x * TILE + x, cell.y * TILE + y, w, h), color)


func _grass(img: Image, cell: Vector2i, rng: RandomNumberGenerator, flowers: bool) -> void:
    _rect(img, cell, 0, 0, TILE, TILE, GRASS)
    for i in range(14):  # darker speckles
        _rect(img, cell, rng.randi_range(0, 29), rng.randi_range(0, 29), 2, 2, GRASS_DARK)
    if flowers:
        for i in range(3):
            var x := rng.randi_range(4, 26)
            var y := rng.randi_range(4, 26)
            _rect(img, cell, x, y, 3, 3, Color(1, 1, 1))
            _rect(img, cell, x + 1, y + 1, 1, 1, Color(1, 0.8, 0.2))


func _path(img: Image, cell: Vector2i, mask: int) -> void:
    # The dirt is always a 16 x 16 square in the middle ...
    _rect(img, cell, 8, 8, 16, 16, DIRT)
    # ... plus an arm for every side that has a neighbour.
    if mask & 1:
        _rect(img, cell, 8, 0, 16, 8, DIRT)
```

```
    if mask & 2:
        _rect(img, cell, 24, 8, 8, 16, DIRT)
    if mask & 4:
        _rect(img, cell, 8, 24, 16, 8, DIRT)
    if mask & 8:
        _rect(img, cell, 0, 8, 8, 16, DIRT)
    # A thin darker outline on the sides where the dirt stops.
    if not (mask & 1):
        _rect(img, cell, 8, 8, 16, 2, DIRT_EDGE)
    if not (mask & 2):
        _rect(img, cell, 22, 8, 2, 16, DIRT_EDGE)
    if not (mask & 4):
        _rect(img, cell, 8, 22, 16, 2, DIRT_EDGE)
    if not (mask & 8):
        _rect(img, cell, 8, 8, 2, 16, DIRT_EDGE)


func _mud(img: Image, cell: Vector2i, rng: RandomNumberGenerator) -> void:
    _rect(img, cell, 0, 0, TILE, TILE, MUD)
    for i in range(10):
        _rect(img, cell, rng.randi_range(0, 27), rng.randi_range(0, 27), 4, 3, Color(0.36, 0.26, 0.17))


func _water(img: Image, cell: Vector2i) -> void:
    _rect(img, cell, 0, 0, TILE, TILE, WATER)
    for y in [6, 16, 26]:
        _rect(img, cell, 4, y, 10, 2, WATER_LIGHT)
        _rect(img, cell, 18, y + 3, 10, 2, WATER_LIGHT)


func _wall(img: Image, cell: Vector2i) -> void:
    _rect(img, cell, 0, 0, TILE, TILE, BRICK)
    for row in range(4):
        var y := row * 8
        _rect(img, cell, 0, y, TILE, 1, BRICK_LINE)
        var offset := 0 if row % 2 == 0 else 8
        for x in range(offset, TILE, 16):
            _rect(img, cell, x, y, 1, 8, BRICK_LINE)


func _bush(img: Image, cell: Vector2i, rng: RandomNumberGenerator) -> void:
    _grass(img, cell, rng, false)
    for pos in [Vector2i(16, 16), Vector2i(10, 12), Vector2i(22, 12), Vector2i(10, 21), Vector2i(22, 21)]:
        for y in range(-8, 9):
            for x in range(-8, 9):
                if x * x + y * y <= 64:
                    img.set_pixel(cell.x * TILE + pos.x + x, cell.y * TILE + pos.y + y, BUSH)
    for pos in [Vector2i(13, 11), Vector2i(21, 15), Vector2i(12, 20)]:
        _rect(img, cell, pos.x, pos.y, 4, 3, BUSH_LIGHT)
```

Points to notice:

- `Image.create(256, 128, false, Image.FORMAT_RGBA8)` makes an empty 256 by 128 picture; `fill_rect` paints rectangles into it. Our `_rect` helper adds the tile's position, so we can draw with coordinates *inside* the tile (0 to 31).
- A `RandomNumberGenerator` with a fixed `seed` makes the "random" speckles the same every time, so everyone gets the same atlas.
- `_path` draws the dirt square in the middle and an arm to each side present in the `mask`. That is Figure 12.5 as code.
- `img.save_png("res://tiles/atlas.png")` writes the file. Godot prints the result code: `0` means OK.

To run it, open a terminal in the project folder. `godot` here means your Godot executable (on Windows, the full path to the console version of the `.exe`, the one with `_console` in its name):

```
godot --headless --path . --script res://tools/make_atlas.gd
godot --headless --path . --import
```

The first command makes the PNG. The second makes Godot import it, so that it can be loaded as a texture. If you open the PNG, you should see the 8 by 4 atlas of Figure 12.2.

Step 3: Build the TileSet with a script

The second tool does what Section 12.2, 12.4 and 12.5 described, in code. Create `tools/make_tileset.gd`:

`tools/make_tileset.gd`

```
extends SceneTree
## Tool script: builds the TileSet resource (res://tiles/tileset.tres) from the atlas picture.
## Run make_atlas.gd first, then let Godot import the PNG, then run this:
##   godot --headless --path . --script res://tools/make_atlas.gd
##   godot --headless --path . --import
##   godot --headless --path . --script res://tools/make_tileset.gd
##
## Everything done here with code is what you do by clicking in the TileSet editor.

const TILE := 32
const SOURCE_ID := 0


func _init() -> void:
    var tileset := TileSet.new()
    tileset.tile_size = Vector2i(TILE, TILE)

    # Physics layer 0: tiles can carry collision shapes.
    tileset.add_physics_layer()
    tileset.set_physics_layer_collision_layer(0, 1)

    # Custom data layer 0: our own number per tile.
    tileset.add_custom_data_layer()
    tileset.set_custom_data_layer_name(0, "speed_factor")
    tileset.set_custom_data_layer_type(0, TYPE_FLOAT)

    # Terrain set 0 with one terrain called "Path". MATCH_SIDES looks at the 4 sides.
    tileset.add_terrain_set()
    tileset.set_terrain_set_mode(0, TileSet.TERRAIN_MODE_MATCH_SIDES)
    tileset.add_terrain(0)
    tileset.set_terrain_name(0, 0, "Path")
    tileset.set_terrain_color(0, 0, Color(0.78, 0.62, 0.38))

    var source := TileSetAtlasSource.new()
    source.texture = load("res://tiles/atlas.png")
    source.texture_region_size = Vector2i(TILE, TILE)
    tileset.add_source(source, SOURCE_ID)  # add it BEFORE editing tile data

    # The 16 path tiles.
    for mask in range(16):
        var coords := Vector2i(mask % 4, floori(mask / 4.0))
        source.create_tile(coords)
        var data := source.get_tile_data(coords, 0)
        data.terrain_set = 0
        data.terrain = 0
        if mask & 1:
            data.set_terrain_peering_bit(TileSet.CELL_NEIGHBOR_TOP_SIDE, 0)
        if mask & 2:
            data.set_terrain_peering_bit(TileSet.CELL_NEIGHBOR_RIGHT_SIDE, 0)
        if mask & 4:
            data.set_terrain_peering_bit(TileSet.CELL_NEIGHBOR_BOTTOM_SIDE, 0)
        if mask & 8:
            data.set_terrain_peering_bit(TileSet.CELL_NEIGHBOR_LEFT_SIDE, 0)
        data.set_custom_data("speed_factor", 1.25)
```

```
    # Plain tiles. Arguments: atlas coordinates, speed factor, solid or not.
    _plain(source, Vector2i(4, 0), 1.0, false)   # grass
    _plain(source, Vector2i(5, 0), 1.0, false)   # grass with flowers
    _plain(source, Vector2i(6, 0), 0.5, false)   # mud: slow
    _plain(source, Vector2i(7, 0), 1.0, true)    # water: solid
    _plain(source, Vector2i(4, 1), 1.0, true)    # brick wall: solid
    _plain(source, Vector2i(5, 1), 1.0, true)    # bush: solid

    var err := ResourceSaver.save(tileset, "res://tiles/tileset.tres")
    print("Saved tileset.tres, result code: ", err, " (0 means OK)")
    quit()


func _plain(source: TileSetAtlasSource, coords: Vector2i, speed: float, solid: bool) -> void:
    source.create_tile(coords)
    var data := source.get_tile_data(coords, 0)
    data.set_custom_data("speed_factor", speed)
    if solid:
        data.add_collision_polygon(0)
        # Points are measured from the middle of the tile.
        var h := TILE / 2.0
        data.set_collision_polygon_points(0, 0, PackedVector2Array([
            Vector2(-h, -h), Vector2(h, -h), Vector2(h, h), Vector2(-h, h)]))
```

Read the order of operations. It matters:

1. Create the `TileSet` and set the tile size.
2. Add the physics layer, the custom data layer and the terrain set. (A tile can only use layers that exist.)
3. Create the atlas source, give it the texture, and **add it to the tile set before editing tiles**. Tile data is sized according to the layers of the tile set it belongs to, so the source must already be inside it.
4. Create the 16 path tiles with their terrain and peering bits, then the plain tiles with their data and collision.
5. Save the resource with `ResourceSaver.save`.

Run it:

```
godot --headless --path . --script res://tools/make_tileset.gd
godot --headless --path . --import
```

You now have `tiles/tileset.tres`. Double-click it in the FileSystem dock and open the TileSet panel: you will see exactly the tiles, collision shapes, terrain and custom data you would have made by hand. (If you prefer the clicking approach, make the TileSet in the editor following Section 12.2 to 12.5 instead; the rest of the project works the same.)

> **Note:** The finished project folder already contains `atlas.png` and `tileset.tres`, so you can skip Steps 2 and 3 and go to Step 4. The scripts are there so you can see how they came to be, and change them.

Step 4: Pip

Create `player.tscn`:

```
Player (CharacterBody2D)        <- script: player.gd, Motion Mode: Floating
└── CollisionShape2D            shape: CircleShape2D, radius 10
```

Motion Mode (in the Inspector under CharacterBody2D) is **Floating** for top-down games: there is no "floor" and no gravity. Attach `player.gd`:

`player.gd`

```gdscript
extends CharacterBody2D
## Pip, seen from above. Walks in 4 directions and slows down in mud.

const BASE_SPEED := 160.0

## The Ground layer, set by the level. Pip asks it what he is standing on.
var ground: TileMapLayer
var current_speed_factor: float = 1.0


func _physics_process(_delta: float) -> void:
    _read_ground()
    var direction := Input.get_vector("ui_left", "ui_right", "ui_up", "ui_down")
    velocity = direction * BASE_SPEED * current_speed_factor
    move_and_slide()


func _read_ground() -> void:
    current_speed_factor = 1.0
    if ground == null:
        return
    var cell := ground.local_to_map(ground.to_local(global_position))
    var data := ground.get_cell_tile_data(cell)  # null if the cell is empty
    if data:
        current_speed_factor = data.get_custom_data("speed_factor")


func _draw() -> void:
    draw_circle(Vector2.ZERO, 11.0, Color(0.62, 0.36, 0.1))
    draw_circle(Vector2.ZERO, 9.0, Color(0.95, 0.65, 0.2))
    draw_circle(Vector2(-4, -3), 3.0, Color.WHITE)
    draw_circle(Vector2(4, -3), 3.0, Color.WHITE)
    draw_circle(Vector2(-3.5, -2.5), 1.4, Color(0.1, 0.1, 0.15))
    draw_circle(Vector2(4.5, -2.5), 1.4, Color(0.1, 0.1, 0.15))
```

`Input.get_vector` combines four actions into one direction vector with length at most 1, so diagonal movement is not faster (Chapter 7). `_read_ground` asks the layer which tile Pip stands on and reads its `speed_factor`. Pip draws himself with `_draw`.

Step 5: The coin

Create `coin.tscn` with an `Area2D` named `Coin`, one child `CollisionShape2D` with a `CircleShape2D` of radius 9, and attach `coin.gd`:

`coin.gd`

```gdscript
extends Area2D
## A coin lying on the floor. Tells the level when Pip picks it up.

signal collected


func _ready() -> void:
    body_entered.connect(_on_body_entered)


func _on_body_entered(_body: Node2D) -> void:
    collected.emit()
    queue_free()


func _draw() -> void:
    draw_circle(Vector2.ZERO, 9.0, Color(1.0, 0.85, 0.2))
    draw_arc(Vector2.ZERO, 9.0, 0.0, TAU, 24, Color(0.7, 0.5, 0.0), 2.0)
```

Because the `Player` is on collision layer 1 and the coin's mask includes layer 1 by default, `body_entered` fires when Pip touches it. The coin sends its own signal, `collected`, so that the level can count it without the coin knowing anything about the level (Chapter 5's lesson on signals).

Step 6: The level scene

Create `main.tscn`:

```
Level (Node2D)                  <- script: level.gd
├── Ground (TileMapLayer)       Tile Set: tiles/tileset.tres
├── Walls (TileMapLayer)        Tile Set: tiles/tileset.tres
├── Coins (Node2D)
├── Player (instance of player.tscn)
└── HUD (CanvasLayer)
    └── Info (Label)            top of the screen, with a black outline for readability
```

The order of children is the draw order: `Walls` over `Ground`, then the coins, then Pip, so he walks in front of everything. Both layers use the same `tileset.tres`. Attach `level.gd` to the root:

`level.gd`

```gdscript
extends Node2D
## Builds a level from a text map. Left click paints path, right click removes it.

const SOURCE_ID := 0                # the atlas source inside the TileSet
const GRASS := Vector2i(4, 0)       # atlas coordinates of each plain tile
const FLOWERS := Vector2i(5, 0)
const MUD := Vector2i(6, 0)
const WATER := Vector2i(7, 0)
const WALL := Vector2i(4, 1)
const BUSH := Vector2i(5, 1)
const PATH_TERRAIN_SET := 0
const PATH_TERRAIN := 0

# # wall   T bush   ~ water   , mud   P path   c coin   @ Pip's start   . grass
const MAP: Array[String] = [
    "################################",
    "#@.PPPPPPPPP.....T....~~~..c#",
    "#..P.........,,.......~~~...#",
    "#..P....T....,,,.............#",
    "#.cP.........,,,....T.......#",
    "#..PPPPPPP.....,,.............#",
    "#........P.....T.............#",
    "#...TT...P..........PPPPPP..#",
    "#...TT...PPPPPPPPPPPP....P..#",
    "#...................T....P..#",
    "#.~~~~.........,,,,.......Pc.#",
    "#.~~~~.........,,,,......P..#",
    "#.~~~~....c.....,,.....T.P..#",
    "#.............T.........P..#",
    "#.c.............c.......P.c#",
    "################################",
]

const COIN_SCENE: PackedScene = preload("res://coin.tscn")

@onready var ground: TileMapLayer = $Ground
@onready var walls: TileMapLayer = $Walls
@onready var player: CharacterBody2D = $Player
@onready var coins: Node2D = $Coins
@onready var info: Label = $HUD/Info

var coins_total: int = 0
var coins_found: int = 0
```

```
func _ready() -> void:
    _build_level()
    player.ground = ground
    _update_info()


func _process(_delta: float) -> void:
    _update_info()


func _build_level() -> void:
    var rng := RandomNumberGenerator.new()
    rng.seed = 7
    var path_cells: Array[Vector2i] = []
    for y in range(MAP.size()):
        var line := MAP[y]
        for x in range(line.length()):
            var cell := Vector2i(x, y)
            var symbol := line[x]
            # Every cell has some ground. Pick it first.
            if symbol == ",":
                ground.set_cell(cell, SOURCE_ID, MUD)
            else:
                var flower_chance := rng.randf() < 0.15
                ground.set_cell(cell, SOURCE_ID, FLOWERS if flower_chance else GRASS)
            # Then the things that stand on the ground.
            match symbol:
                "#":
                    walls.set_cell(cell, SOURCE_ID, WALL)
                "T":
                    walls.set_cell(cell, SOURCE_ID, BUSH)
                "~":
                    walls.set_cell(cell, SOURCE_ID, WATER)
                "P":
                    path_cells.append(cell)
                "c":
                    _add_coin(cell)
                "@":
                    player.position = ground.map_to_local(cell)
    # One call places the right tile from the 16 for every path cell.
    ground.set_cells_terrain_connect(path_cells, PATH_TERRAIN_SET, PATH_TERRAIN)


func _add_coin(cell: Vector2i) -> void:
    var coin := COIN_SCENE.instantiate()
    coin.position = ground.map_to_local(cell)
    coin.collected.connect(_on_coin_collected)
    coins.add_child(coin)
    coins_total += 1


func _on_coin_collected() -> void:
    coins_found += 1


func _unhandled_input(event: InputEvent) -> void:
    if not (event is InputEventMouseButton and event.pressed):
        return
    var cell := ground.local_to_map(ground.get_local_mouse_position())
    if walls.get_cell_source_id(cell) != -1 or ground.get_cell_source_id(cell) == -1:
        return  # solid tile or outside the map: leave it alone
    if event.button_index == MOUSE_BUTTON_LEFT:
        paint_path(cell)
    elif event.button_index == MOUSE_BUTTON_RIGHT:
        erase_path(cell)


func paint_path(cell: Vector2i) -> void:
    ground.set_cells_terrain_connect([cell], PATH_TERRAIN_SET, PATH_TERRAIN)


func erase_path(cell: Vector2i) -> void:
```

```
    ground.set_cell(cell, SOURCE_ID, GRASS)
    # Path neighbours must be re-fitted so they stop pointing at the empty spot.
    var neighbours: Array[Vector2i] = []
    for offset in [Vector2i.UP, Vector2i.RIGHT, Vector2i.DOWN, Vector2i.LEFT]:
        var n: Vector2i = cell + offset
        var data := ground.get_cell_tile_data(n)
        if data and data.terrain_set == PATH_TERRAIN_SET and data.terrain == PATH_TERRAIN:
            neighbours.append(n)
    ground.set_cells_terrain_connect(neighbours, PATH_TERRAIN_SET, PATH_TERRAIN, false)


func _update_info() -> void:
    info.text = "Coins %d / %d    Speed x%.2f    Arrows: walk   Left click: paint path   Right click:
erase path" \
        % [coins_found, coins_total, player.current_speed_factor]
```

Read these parts:

- `MAP` is the level as text (30 columns by 16 rows, 960 by 512 pixels). The comment above it lists the symbols.
- `_build_level` visits every character. Every cell first gets ground (grass, with about 15 percent flowers chosen by a seeded random generator, or mud). Then a `match` places the things standing on it: walls, bushes and water go on the `Walls` layer; `c` makes a coin; `@` moves Pip to the start cell.
- `P` cells are collected in `path_cells` and painted **in one go** at the end with `set_cells_terrain_connect`, so that Godot can see the whole network at once and join it correctly.
- `_unhandled_input` handles mouse clicks: left paints a path, right erases it. It ignores clicks on walls or outside the map.
- `erase_path` replaces the cell with grass, finds the neighbors that are still path, and refits them with `set_cells_terrain_connect(..., false)`. The last argument, `ignore_empty_terrains`, set to `false` makes the empty neighbors count, so the arms that pointed at the removed cell disappear.
- `_update_info` shows the coin count and the speed factor of the tile under Pip.

Set `main.tscn` as the main scene.

Step 7: Run and test

Press **F5** and check:

1. The level appears: brick walls around the edge, bushes, a lake in the bottom left and a pond at the top right, mud patches, and paths with proper corners and T-junctions.
2. Pip appears in the top-left corner. Arrow keys move him in all four directions.
3. He cannot walk through walls, bushes or water, but walks freely on grass and paths.
4. On mud, the label shows **Speed x0.50** and he crawls. On a path it shows **x1.25** and he is faster.
5. Walk over each coin: the count at the top goes up and the coin disappears. There are 7 coins.
6. Left-click on any grass cell: a path tile appears, and it joins any path next to it with the right piece. Click several cells next to each other to draw a road.
7. Right-click on a path cell: it turns to grass and the neighbors lose their connection to it.
8. Click on a wall or water: nothing happens.

Expected result

A 30 by 16 tile top-down garden with a brick border. Paths wind across it, you can draw more with the mouse, Pip walks around collecting 7 coins, slowing down in mud and speeding up on paths. Everything in the picture comes from tiles that were drawn by a script.



Figure 12.6: The finished project running.

Making it your own (optional)

- Edit the `MAP` text: add rooms, change the coin positions. You can see the whole level at a glance.
- Add a tile of your own in `make_atlas.gd` (for example, ice) and a `speed_factor` of 2.0 in `make_tileset.gd`.
- Change `MAP` rows to add a second door-like gap in the wall.

Chapter summary

- A tile map fills a grid with repeated tile pictures. It is faster, smaller and easier to edit than placing many sprites.
- Cells use integer `Vector2i` coordinates, `y` pointing down. `local_to_map` and `map_to_local` convert between pixels and cells.
- A `TileSet` holds the tiles (usually from an atlas image), and one or more `TileMapLayer` nodes use it. Set the TileSet's tile size to match the atlas, or tiles look cut up.
- Use several layers to separate ground from walls and decoration.
- Add a physics layer to the TileSet and draw collision shapes on the tiles that should block movement.
- Custom data layers store your own values on tiles. Read them with `get_cell_tile_data(cell).get_custom_data("name")`.
- Terrains let Godot pick the correct edge and corner tile automatically. In Match Sides mode, 16 tiles cover every case.
- `set_cell`, `erase_cell` and `set_cells_terrain_connect` build and edit levels from code. A text map is a convenient way to describe a level.

Review questions

1. **Short answer.** Give two advantages of building a level with a `TileMapLayer` instead of hundreds of separate `Sprite2D` nodes.

2. **Multiple choice.** With 32 by 32 pixel tiles, which cell contains the pixel position `(100, 40)`? a) `(2, 1)` b) `(3, 1)` c) `(100, 40)` d) `(1, 3)`
3. **True or false.** The cell `(3.5, 2)` is a valid cell coordinate.
4. **Short answer.** What is the difference between a tile's *atlas coordinates* and a cell's *map coordinates*?
5. **Multiple choice.** Your tiles look cut into pieces and misaligned. What is the most likely cause? a) The atlas is a PNG b) The TileSet's Tile Size does not match the atlas tiles c) There is only one layer d) The cell coordinates are negative
6. **Short answer.** Why does the project put walls on a separate `Walls` layer from the ground?
7. **Multiple choice.** What does `layer.get_cell_source_id(cell)` return for a cell that has no tile? a) 0 b) `null` c) -1 d) An error
8. **Short answer.** In a Match Sides terrain, why does a full set need 16 tiles?
9. **True or false.** Calling `get_cell_tile_data(cell)` on an empty cell returns `null`, so you should check the result before reading data from it.
10. **Short answer.** A character walks straight through a wall tile that has a collision shape in the TileSet. Name two things you would check.

Exercises

1. **Practice (easy).** Edit the `MAP` in `level.gd`: add a second pond of water (`~`) and three more coins. Run the project and check that the HUD shows the new total.
2. **Practice (easy).** In `make_tileset.gd`, change the speed factor of path tiles from 1.25 to 2.0. Re-run the tool, re-import, and observe the speed on paths.
3. **Apply (medium).** Add an `ice` tile to the atlas (draw it in `make_atlas.gd` at atlas position `(6, 1)`, in light blue) with `speed_factor` 1.6 and put it in the map with the symbol `i`. Extend `_build_level` so that `i` sets the ice tile on the ground layer.
4. **Apply (medium).** Add a key press (`Space`) that fills the `Walls` layer cell *in front of Pip* with a bush if it is empty, and removes it if one is there. Hint: use `player.global_position`, `walls.local_to_map`, and `get_cell_source_id`. (Use the cell Pip stands on plus an offset of `Vector2i.RIGHT`.)
5. **Challenge (hard).** Write a function that **generates** a random level: start from a `Walls` border, fill the inside with grass, then place 15 bushes and 3 water cells at random free cells and 8 coins at random free cells. Make sure the start cell is free, and make a key `R` rebuild the level. (Remember to call `clear()` on both layers and to free old coins.)

Common mistakes

- **Symptom:** Tiles look cut up, shifted, or show half of two tiles. **Cause:** The TileSet's Tile Size (default 16 by 16) does not match the atlas. **Fix:** Set Tile Size to the true tile size (here 32 by 32) before creating tiles.
- **Symptom:** The character walks through walls. **Cause:** The TileSet has no physics layer, the wall tile has no collision polygon, or the character's collision mask does not include the layer. **Fix:** Add Physics Layer 0, draw shapes on the solid tiles, and check layers and masks (Chapter 8).
- **Symptom:** `get_custom_data` prints an error or returns `null`. **Cause:** The cell is empty (`TileData` is `null`), or the custom data name is misspelled. **Fix:** Check `if data:` first and copy the name exactly from the Custom Data Layers list.

- **Symptom:** Terrain painting puts wrong or no tiles. **Cause:** Peering bits were not set on every tile of the family, or the tiles use a different terrain set. **Fix:** Check each tile's terrain and peering bits in the TileSet panel; with Match Sides there must be exactly one tile for each of the 16 combinations.
- **Symptom:** The tile you set in code is blank or wrong. **Cause:** Wrong `source_id` or atlas coordinates (for example using pixels, or swapping x and y). **Fix:** Atlas coordinates count tiles, not pixels, and are `(column, row)`. Source ids start at 0.
- **Symptom:** `set_cell` appears to do nothing at runtime. **Cause:** The layer has no TileSet assigned, or you changed a different layer than the one you are looking at. **Fix:** Check the **Tile Set** property of the layer and the node name you are editing.

Further reading

- Using tilemaps: https://docs.godotengine.org/en/stable/tutorials/2d/using_tilemaps.html
- Using TileSets: https://docs.godotengine.org/en/stable/tutorials/2d/using_tilesets.html
- TileMapLayer class reference: https://docs.godotengine.org/en/stable/classes/class_tilemaplayer.html
- TileSet class reference: https://docs.godotengine.org/en/stable/classes/class_tileset.html
- TileData class reference: https://docs.godotengine.org/en/stable/classes/class_tiledata.html

Chapter 13: The Platformer Project

Learning objectives

By the end of this chapter you will be able to:

- Explain how gravity, velocity and `move_and_slide()` work together to make a character fall, land and jump.
- Calculate how high and how far a jump goes from three numbers (jump speed, gravity, run speed), and tune those numbers on purpose.
- Add "game feel" to a jump with variable jump height, coyote time and a jump buffer.
- Build a level from a text map by generating a `TileSet` in code and placing tiles with `TileMapLayer.set_cell()`.
- Write an enemy that patrols, turns at walls and ledges, and can be defeated by a stomp from above.
- Connect pickups, hazards and a goal to a game controller with signals, and run a two-level game with lives and a HUD.

Key terms

- **Platformer**: a game seen from the side in which the player runs, jumps and avoids hazards while gravity pulls everything down.
- **Gravity**: a constant downward acceleration, written in pixels per second per second, that we add to the vertical velocity each physics frame.
- **Jump velocity**: the upward speed (a negative number in Godot's 2D space) given to the character at the moment of a jump.
- **Variable jump height**: letting the player choose a short hop or a high jump by how long they hold the jump button.
- **Coyote time**: a short grace period after walking off a ledge during which the character may still jump.
- **Jump buffer**: remembering a jump press that happened slightly too early, and using it the moment the character lands.
- **Patrol**: an enemy behavior of walking back and forth along a path, turning around at obstacles.
- **Stomp**: defeating an enemy by landing on its head while moving downward.
- **Hazard**: anything that harms the player on touch, such as spikes or a bottomless pit.
- **Level map**: a grid of text characters in which each letter stands for a tile or an object, used to build a level.
- **Game controller**: a script that sits above the levels and tracks lives, score and which level is loaded.
- **Kill plane**: an invisible line below the level; anything that falls past it counts as having fallen out of the world.

13.1 What we are building

In this chapter you build a complete small platformer. Pip, our round hero, runs and jumps across two levels. Slimes patrol the ground; you can squash them by jumping on their heads. Coins hang in the air, spikes and pits are dangerous, and a flag marks the end of each level. You have three lives. When you finish the second level you win.

You already own every skill you need. Chapter 7 gave you input and movement. Chapter 8 gave you `CharacterBody2D`, `Area2D` and collision layers. Chapter 9 taught you to spawn things and keep score. Chapter 10 brought tweens. Chapter 12 brought `TileMapLayer`. This chapter is about combining them. Beginners often find combining harder than learning, so we go slowly and test each piece.

> **Note:** Everything in this project is drawn with code or built-in nodes. There are no image or sound files to download. The tile pictures, Pip, the slimes, the coins and the flag are all produced with `_draw()` and an `Image` created while the game runs. If you later want prettier art, free packs exist at Kenney.nl, itch.io and OpenGameArt, and you can swap them in. That is optional.

The pieces

Figure 13.1 shows the whole design before we write any code. It is worth studying, because a platformer that is planned is much easier to build than one that grows by accident.

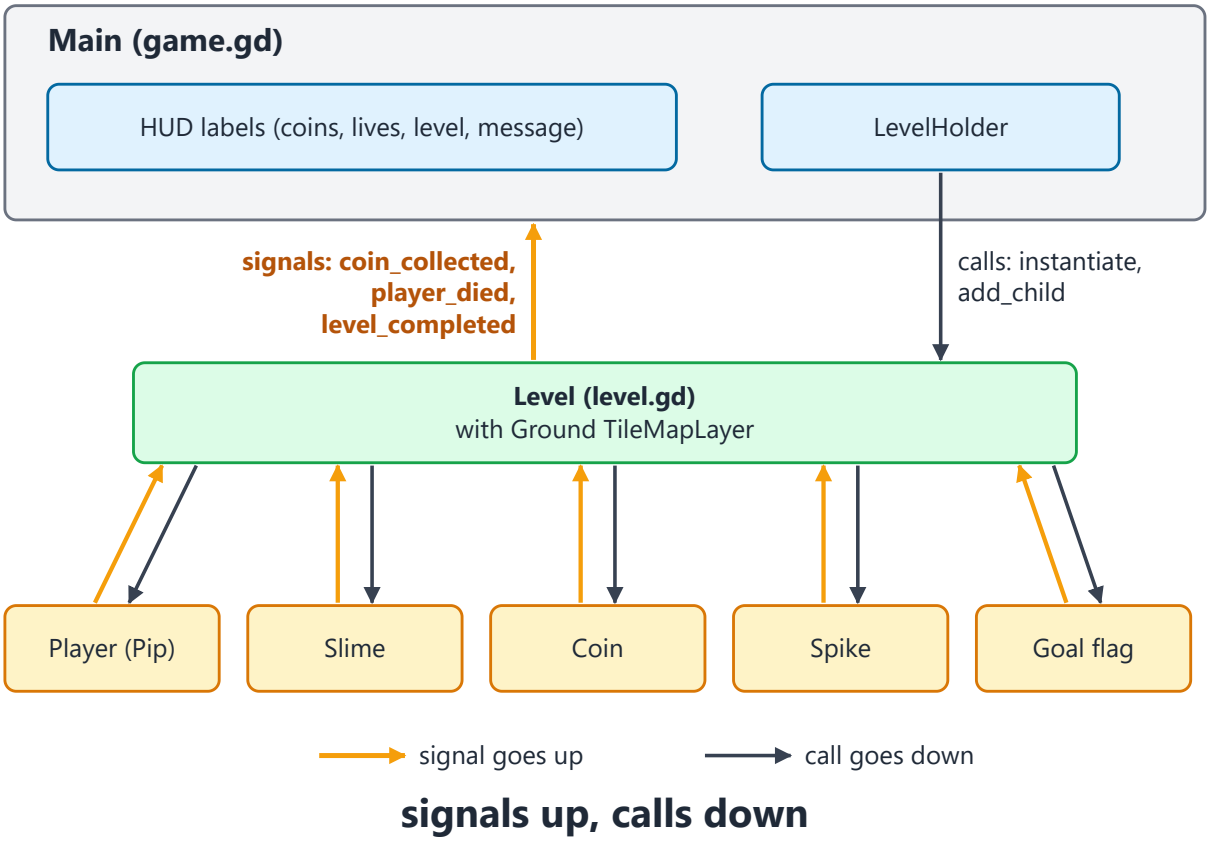


Figure 13.1: Main holds the HUD and the current Level. The Level holds the ground, Pip, slimes, coins, spikes and the flag. Signals travel upward; calls travel downward.

Read the diagram from top to bottom. `Main` (script `game.gd`) is the game controller. It owns the HUD labels and one `Level` at a time. A `Level` (script `level.gd`) builds itself from a text map. It creates Pip, slimes, coins, spikes and a flag. Those small objects never reach up to change the score or the lives. They emit **signals**, the level passes the important ones upward, and `Main` reacts. This rule, "signals go up, calls go down", keeps the project tidy. You will meet it again in Chapter 15.

The project folder

Create a new project named `ch13-platformer`. When you finish, the folder looks like this:

```
ch13-platformer/
  project.godot
  icon.svg
  scenes/
    main.tscn
    level_1.tscn
    level_2.tscn
    player.tscn
    slime.tscn
    coin.tscn
    spike.tscn
    goal.tscn
  scripts/
    game.gd
    level.gd
    tile_factory.gd
    player.gd
    slime.gd
    coin.gd
    spike.gd
    goal.gd
  tests/
    test_platformer.gd
```

The `tests` folder holds a script that plays the game automatically and checks the results. You do not need it to play, but Section 13.8 shows how it works because it is a good way to learn how a game "thinks".

Project settings

Open **Project > Project Settings**.

1. Under **Display > Window**, set **Viewport Width** to `960` and **Viewport Height** to `540`.
2. Under **Display > Window > Stretch**, set **Mode** to `canvas_items`.
3. Under **Layer Names > 2D Physics**, name the first four layers: `world`, `player`, `enemies`, `pickups`.

Names make the collision checkboxes readable. Our rules are:

Layer	Who sits on it	Who looks at it (mask)
1 world	ground tiles	Pip, slimes (they stand on the world)
2 player	Pip	slime hitboxes, coins, spikes, flag
3 enemies	slime bodies	nothing needs to
4 pickups	coins, spikes, flag	nothing needs to

Notice that Pip's mask is only `world`. Slimes and Pip do not push each other physically. Slimes detect Pip with an `Area2D` instead, so we can decide in code what a touch means.

Next open **Project > Project Settings > Input Map** and add three actions:

Action	Keys
`move_left`	A, Left arrow
`move_right`	D, Right arrow
`jump`	Space, W, Up arrow

> **Why it works:** Actions let the script ask "is the player pressing jump?" without caring which key it is. The same script works for keyboard, gamepad or remapped keys.

13.2 Gravity and falling

Gravity in a game is not a mysterious force. It is one line of arithmetic repeated sixty times a second:

```
velocity.y = velocity.y + gravity * delta
```

`velocity.y` is the vertical speed in pixels per second. In Godot's 2D space the y axis points **down**, so a positive `velocity.y` means falling and a negative one means rising. Each physics frame lasts about `delta = 1/60` of a second. If gravity is 1500 pixels per second per second, then every frame adds `1500 / 60 = 25` pixels per second to the fall speed. After one second of falling Pip moves at 1500 pixels per second. That is fast, so we also limit it with a **terminal velocity** (a maximum fall speed). Without a limit, a long fall would make Pip so fast that he could skip past a thin floor between two frames.

Here is the pattern in the smallest possible form. This is a **fragment**; the complete script comes in a moment.

```gdscript
# Fragment from player.gd
if not is_on_floor():
    velocity.y = minf(velocity.y + gravity * delta, max_fall_speed)
```

`is_on_floor()` is a `CharacterBody2D` method. It answers "did the last `move_and_slide()` end with my feet on something solid?" So the check uses information from the previous frame. That is fine, because things only change a little in 1/60 of a second.

`move_and_slide()` does the rest. It moves the body by `velocity * delta`. If it hits the floor, it stops the vertical motion and slides along the surface. When Pip is standing on the floor, `velocity.y` is set back to zero by the collision, so gravity starts fresh every frame without piling up.

The player scene

Make a new scene with a `CharacterBody2D` as root and name it `Player`. Add these children:

```
Player (CharacterBody2D)        layer: 2 (player)   mask: 1 (world)
  CollisionShape2D              shape: CapsuleShape2D, radius 12, height 30
  Camera2D                      Position Smoothing: On
```

Save it as `scenes/player.tscn`. Then attach a new script, `scripts/player.gd`. We will write it in three steps: movement and gravity first, then jumping, then the polish.

Here is the first step. It already runs: Pip falls onto the floor and runs left and right.

```gdscript
# scripts/player.gd  (step 1, will be extended)
class_name Player
extends CharacterBody2D

@export var run_speed: float = 220.0
@export var ground_accel: float = 1800.0
@export var air_accel: float = 1200.0
@export var gravity: float = 1500.0
@export var max_fall_speed: float = 900.0
```

```
func _physics_process(delta: float) -> void:
    if not is_on_floor():
        velocity.y = minf(velocity.y + gravity * delta, max_fall_speed)

    var direction: float = Input.get_axis("move_left", "move_right")
    var accel: float = ground_accel if is_on_floor() else air_accel
    velocity.x = move_toward(velocity.x, direction * run_speed, accel * delta)

    move_and_slide()
```

Three new things appear here.

- `class_name Player` gives the script a global name. Other scripts can write `body is Player` to ask "is this thing Pip?". Rule of thumb: never use a name that Godot already uses for one of its own classes. `Player` is safe.
- `Input.get_axis("move_left", "move_right")` returns `-1.0` when only left is held, `1.0` when only right is held and `0.0` when neither or both are held. Gamepad sticks give values in between.
- `move_toward(from, to, step)` moves a number toward a target by at most `step`. Here it moves Pip's horizontal speed toward the wanted speed. Using `accel * delta` as the step gives a small **acceleration**: Pip speeds up and slows down smoothly instead of instantly. We use a larger acceleration on the ground (more grip) and a smaller one in the air (more floaty).

Drawing Pip

Pip needs a face. Add a `_draw()` function at the bottom of the script. We also want the eyes to look in the direction Pip walks, so we add a `facing` variable.

```
# Add near the top of player.gd with the other variables
var facing: int = 1
```

```
# Add at the bottom of player.gd
func _draw() -> void:
    draw_circle(Vector2.ZERO, 15.0, Color("ffd166"))
    draw_arc(Vector2.ZERO, 15.0, 0.0, TAU, 32, Color("b5832a"), 2.0)
    var eye_x: float = 5.0 * facing
    draw_circle(Vector2(eye_x - 4.0, -4.0), 3.5, Color.WHITE)
    draw_circle(Vector2(eye_x + 6.0, -4.0), 3.5, Color.WHITE)
    draw_circle(Vector2(eye_x - 3.0 + facing, -4.0), 1.8, Color("222222"))
    draw_circle(Vector2(eye_x + 7.0 + facing, -4.0), 1.8, Color("222222"))
```

`_draw()` runs once, then the picture is remembered. When `facing` changes we must ask for a fresh drawing with `queue_redraw()`. You will see that call in the final script.

> **Try it:** To test gravity before you have a level, make a scene `test.tscn` with a `Node2D`, an instance of `player.tscn` at (200, 100), and a `StaticBody2D` with a wide `CollisionShape2D` (a `RectangleShape2D` of size 800 x 40) at (200, 300). Run it. Pip should fall and land, and the arrow keys should move him. Change `gravity` to 400 in the Inspector and notice the moon-like fall. Put it back to 1500.

13.3 The math of a jump

A jump is gravity run backwards for a moment. At the instant of the jump we give Pip a strong upward speed. Gravity then slows that speed down, reaches zero at the top of the arc, and pulls Pip back down. Because the same gravity acts on the way up and the way down, the path is a smooth curve called a parabola.

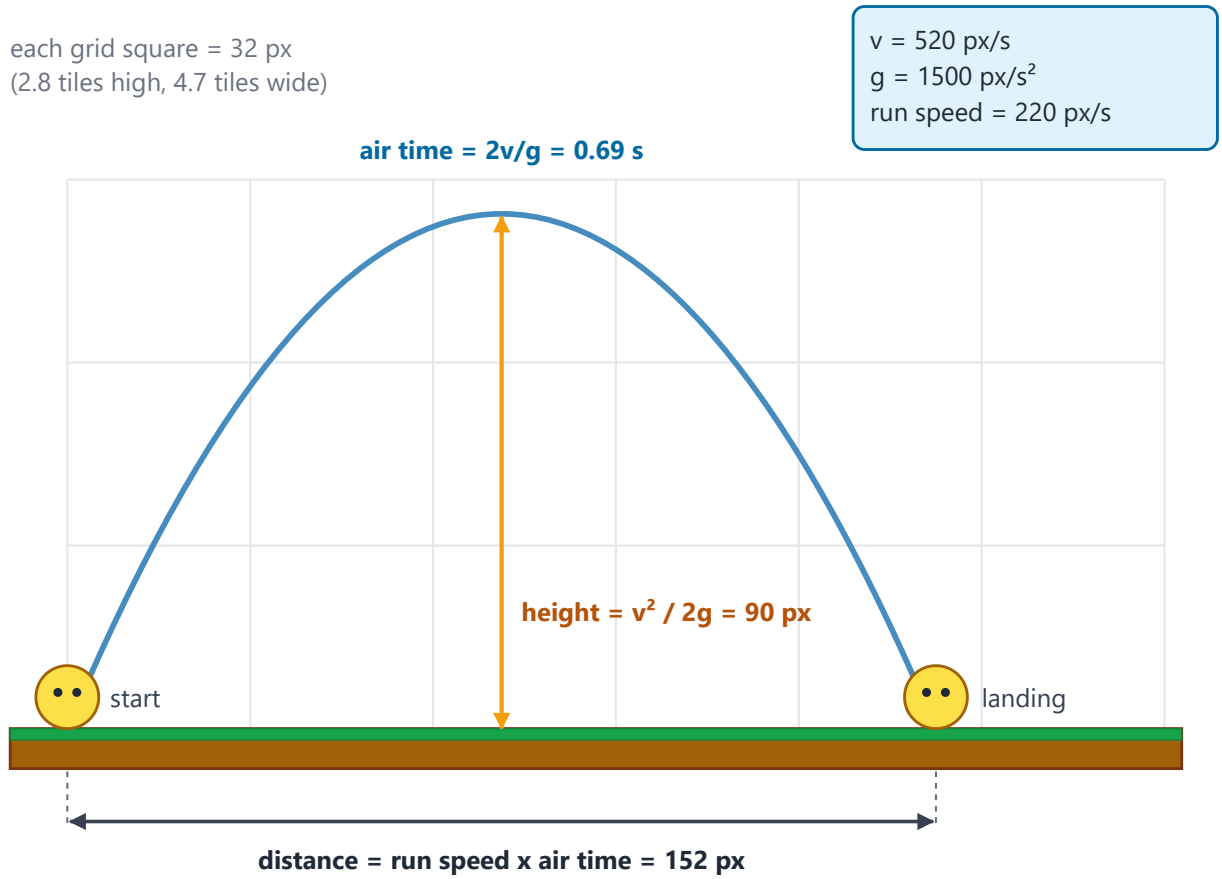


Figure 13.2: Jumping at 520 px/s with gravity 1500 px/s² reaches about 90 pixels high and takes about 0.69 seconds. At 220 px/s Pip travels about 152 pixels sideways in that time.

Three small formulas let you design jumps with a calculator instead of by guessing. Let `v` be the jump speed (use its positive size, 520), `g` the gravity (1500) and `s` the run speed (220).

- **Jump height** = `v² / (2 × g)`. With our numbers: `520 × 520 / 3000 = 90.1` pixels.
- **Air time** (up and back to the same height) = `2 × v / g`. That is `1040 / 1500 = 0.69` seconds.
- **Jump distance** = `s × air time`. That is `220 × 0.69 = 152` pixels.

Our tiles are 32 pixels wide and tall. So Pip can jump up about 2.8 tiles and across about 4.7 tiles. This tells us how to design levels: a wall of height 2 can be jumped, a wall of height 3 cannot; a gap of 4 tiles can be crossed, a gap of 5 cannot. Level designers do exactly this kind of arithmetic. When you build level 2 you will keep every gap at 4 tiles or fewer, and every step at 2 tiles or fewer.

Now the jump itself. Add one variable and one line:

```
# Add with the other @export variables
@export var jump_velocity: float = -520.0   # negative = up

# Add inside _physics_process, before move_and_slide()
if Input.is_action_just_pressed("jump") and is_on_floor():
    velocity.y = jump_velocity
```

`is_action_just_pressed` is true only on the single frame when the key goes down. Holding the key does not repeat it. Run the test scene: Pip jumps. Try the numbers in the table below, one at a time, and feel the difference.

Gravity	Jump velocity	Height (px)	Feel
900	-400	89	slow, floaty, moon-like
1500	-520	90	balanced (our choice)
2400	-650	88	snappy, heavy, "arcade"
1500	-700	163	super jump

Notice that the first three rows reach almost the same height but feel very different. Higher gravity with a higher jump speed makes the same height happen faster. Most modern platformers use fairly high gravity because it feels responsive. Feel is the whole point; the formulas only get you near.

> **Note:** Stage by stage, make one change, play, and decide. Tuning many numbers at once teaches you nothing. Write down numbers you like.

13.4 Making the jump feel good

A jump that obeys physics can still feel bad. Players are human: they press the button a little early or a little late, and they want to control how high they go. Three well-known tricks fix this. Together they take about twenty lines.

Variable jump height

We want a short tap to give a short hop and a long hold to give a full jump. The trick is to cut the upward speed in half if the player lets go of the button while Pip is still rising.

```
# Add to @export variables
@export var jump_cut: float = 0.5

# Add inside _physics_process
if Input.is_action_just_released("jump") and velocity.y < 0.0:
    velocity.y *= jump_cut
```

Holding the button gives the full 90 pixels. A quick tap gives roughly 40. In our automatic test the numbers were 95 and 43 pixels. (The test jumps slightly higher than the formula because the physics engine integrates a fraction of a frame differently. A few pixels do not matter.)

Coyote time

In the cartoons, Wile E. Coyote runs off a cliff and does not fall until he looks down. Platformers borrow the idea. When the player walks off a ledge, we keep allowing a jump for about a tenth of a second. Without this, players feel the game "ate" their jump when they pressed it a hair too late, and they blame the game.

Jump buffer

The opposite problem: the player presses jump just before landing. A buffer remembers the press for about 0.12 seconds and performs the jump the moment Pip touches the floor. Without it the early press is thrown away and the jump seems unresponsive.

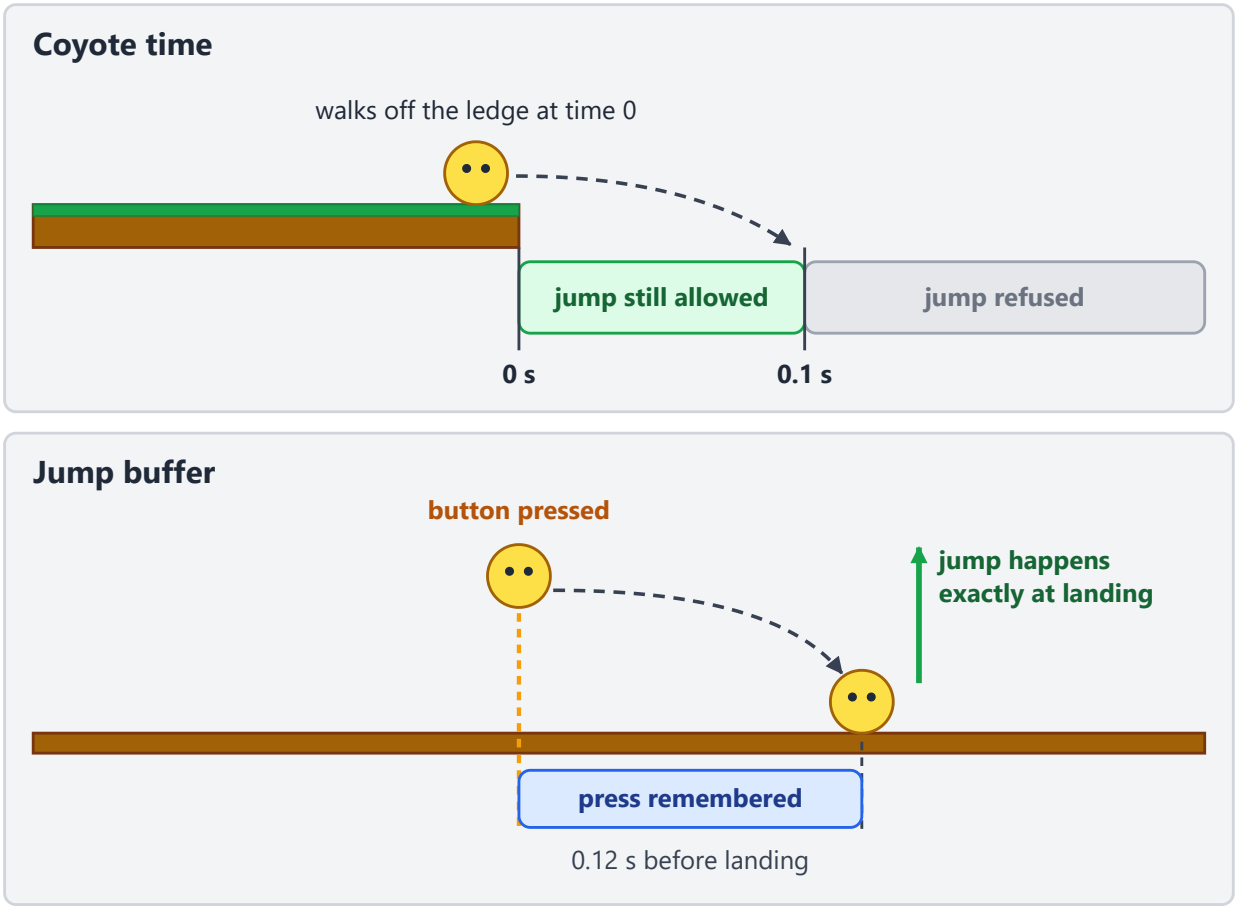


Figure 13.3: Two small windows of forgiveness. Coyote time extends the jump after leaving the ground. The buffer extends the jump request before reaching the ground.

Both tricks use the same pattern: a **countdown timer variable**. We set it to a starting value when something happens, subtract `delta` every frame, and test whether it is still above zero.

Here is the complete `player.gd`. It includes everything from this chapter so far, plus death and bouncing, which the slimes will need.

```gdscript
# scripts/player.gd
class_name Player
extends CharacterBody2D
## Pip, the hero. Runs, jumps, stomps slimes.

signal died

@export var run_speed: float = 220.0
@export var ground_accel: float = 1800.0
@export var air_accel: float = 1200.0
@export var jump_velocity: float = -520.0
@export var gravity: float = 1500.0
@export var max_fall_speed: float = 900.0
@export var jump_cut: float = 0.5
@export var coyote_time: float = 0.1
@export var jump_buffer_time: float = 0.12
@export var stomp_bounce: float = -350.0

var kill_y: float = 100000.0   # the level sets this: fall below it and Pip dies
var is_dead: bool = false
var facing: int = 1

var _coyote_left: float = 0.0
var _buffer_left: float = 0.0
```

```
func _physics_process(delta: float) -> void:
    if is_dead:
        return

    # 1. Gravity
    if not is_on_floor():
        velocity.y = minf(velocity.y + gravity * delta, max_fall_speed)

    # 2. Timers for forgiving jumps
    if is_on_floor():
        _coyote_left = coyote_time
    else:
        _coyote_left -= delta
    if Input.is_action_just_pressed("jump"):
        _buffer_left = jump_buffer_time
    else:
        _buffer_left -= delta

    # 3. Jump
    if _buffer_left > 0.0 and _coyote_left > 0.0:
        velocity.y = jump_velocity
        _buffer_left = 0.0
        _coyote_left = 0.0
    if Input.is_action_just_released("jump") and velocity.y < 0.0:
        velocity.y *= jump_cut   # let go early: shorter jump

    # 4. Horizontal movement
    var direction: float = Input.get_axis("move_left", "move_right")
    var accel: float = ground_accel if is_on_floor() else air_accel
    velocity.x = move_toward(velocity.x, direction * run_speed, accel * delta)
    if direction != 0.0 and signi(int(direction)) != facing:
        facing = signi(int(direction))
        queue_redraw()

    move_and_slide()

    # 5. Fell out of the world?
    if global_position.y > kill_y:
        die()


func bounce() -> void:
    velocity.y = stomp_bounce


func die() -> void:
    if is_dead:
        return
    is_dead = true
    velocity = Vector2.ZERO
    died.emit()
    var tween: Tween = create_tween()
    tween.tween_property(self, "scale", Vector2.ZERO, 0.4)


func _draw() -> void:
    draw_circle(Vector2.ZERO, 15.0, Color("ffd166"))
    draw_arc(Vector2.ZERO, 15.0, 0.0, TAU, 32, Color("b5832a"), 2.0)
    var eye_x: float = 5.0 * facing
    draw_circle(Vector2(eye_x - 4.0, -4.0), 3.5, Color.WHITE)
    draw_circle(Vector2(eye_x + 6.0, -4.0), 3.5, Color.WHITE)
    draw_circle(Vector2(eye_x - 3.0 + facing, -4.0), 1.8, Color("222222"))
    draw_circle(Vector2(eye_x + 7.0 + facing, -4.0), 1.8, Color("222222"))
```

Walk through the numbered blocks:

1. **Gravity** is the same as before.

2. **Timers.** While Pip stands on the floor, the coyote timer is refilled to 0.1 seconds. In the air it counts down. A jump press always refills the buffer timer to 0.12; otherwise the buffer counts down.
3. **Jump.** If both timers are still positive, a jump was requested recently *and* Pip was on the ground recently. Do the jump, and set both timers to zero so it cannot fire twice. Then the "cut" rule trims a jump that was released early.
4. **Horizontal movement** works as before, plus remembering which way Pip faces.
5. **Kill plane.** If Pip's y position is greater than `kill_y`, he fell below the level. The level sets `kill_y` when it builds itself.

`die()` marks Pip dead (so the physics stops), emits the `died` signal, and shrinks him away with a tween. `bounce()` is what a slime will call when Pip lands on its head. The `signal died` line at the top declares a signal; we will use it in 13.7.

> **Warning:** `signi(int(direction))` looks odd. `signi` returns -1, 0 or 1 for an integer, so it turns the decimal axis value into a facing direction. If you write `sign(direction)` it returns a decimal and comparing it to the integer `facing` is clumsy. Keep the integer version.
>
> **Try it:** Put `coyote_time` to 0 and `jump_buffer_time` to 0 in the Inspector and play the test scene. Walk off an edge and try to jump late; try to jump just before landing. Then put the values back. You have just felt exactly what these two numbers buy.

13.5 A world made from text

We could paint levels in the TileMap editor from Chapter 12. For this project we use a different approach that is easier to explain in a book and easier to share: **a level is a block of text**. Each character is one tile, 32 by 32 pixels.

```
..........................................................
.............................####.......................
..P.........s.ccc.............s................^^...s....F..
###################...####################...################
```

(This is only a taste; the complete maps are shown below.) The legend:

Character	Meaning
`.`	empty air
`#`	solid ground
`P`	where Pip starts
`c`	a coin
`s`	a slime
`^`	spikes
`F`	the finish flag

The text is typed into a multi-line property on the level scene. A script reads it line by line, letter by letter, and creates the tiles and objects. Text maps are also great for tests, for sharing levels on forums, and for generating levels with code later.

Making a TileSet with code

A `TileMapLayer` needs a `TileSet`: a collection of tile pictures plus collision shapes. In Chapter 12 you made one in the editor from an image file. Here we have no image file, so we **paint one in memory** with an `Image`, turn it into a texture, and describe the tiles to the `TileSet` through code. The editor does exactly the same things; it just hides them behind buttons.

Create `scripts/tile_factory.gd`:

```gdscript
# scripts/tile_factory.gd
class_name TileFactory
extends RefCounted
## Builds a TileSet entirely in code: no image files needed.

const TILE_SIZE: int = 32


static func make_tile_set() -> TileSet:
    # Draw a two-tile picture: dirt (left) and grass-topped dirt (right).
    var image: Image = Image.create(TILE_SIZE * 2, TILE_SIZE, false, Image.FORMAT_RGBA8)
    image.fill(Color("7a4b2a"))
    for x in TILE_SIZE * 2:
        for y in TILE_SIZE:
            if (x * 7 + y * 13) % 11 == 0:
                image.set_pixel(x, y, Color("6a3f22"))   # dark speckles
    for x in range(TILE_SIZE, TILE_SIZE * 2):
        for y in 8:
            image.set_pixel(x, y, Color("58b368"))        # grass cap
        for y in range(8, 11):
            image.set_pixel(x, y, Color("3f8a4d"))
    var texture: ImageTexture = ImageTexture.create_from_image(image)

    var tile_set: TileSet = TileSet.new()
    tile_set.tile_size = Vector2i(TILE_SIZE, TILE_SIZE)
    tile_set.add_physics_layer()          # physics layer 0; defaults to collision layer 1 ("world")

    var source: TileSetAtlasSource = TileSetAtlasSource.new()
    source.texture = texture
    source.texture_region_size = Vector2i(TILE_SIZE, TILE_SIZE)
    source.create_tile(Vector2i(0, 0))
    source.create_tile(Vector2i(1, 0))
    tile_set.add_source(source, 0)

    var h: float = TILE_SIZE / 2.0
    var square: PackedVector2Array = PackedVector2Array([
        Vector2(-h, -h), Vector2(h, -h), Vector2(h, h), Vector2(-h, h)])
    for x in 2:
        var data: TileData = source.get_tile_data(Vector2i(x, 0), 0)
        data.add_collision_polygon(0)
        data.set_collision_polygon_points(0, 0, square)
    return tile_set
```

Let us unpack it, because this one function teaches how tilemaps really work.

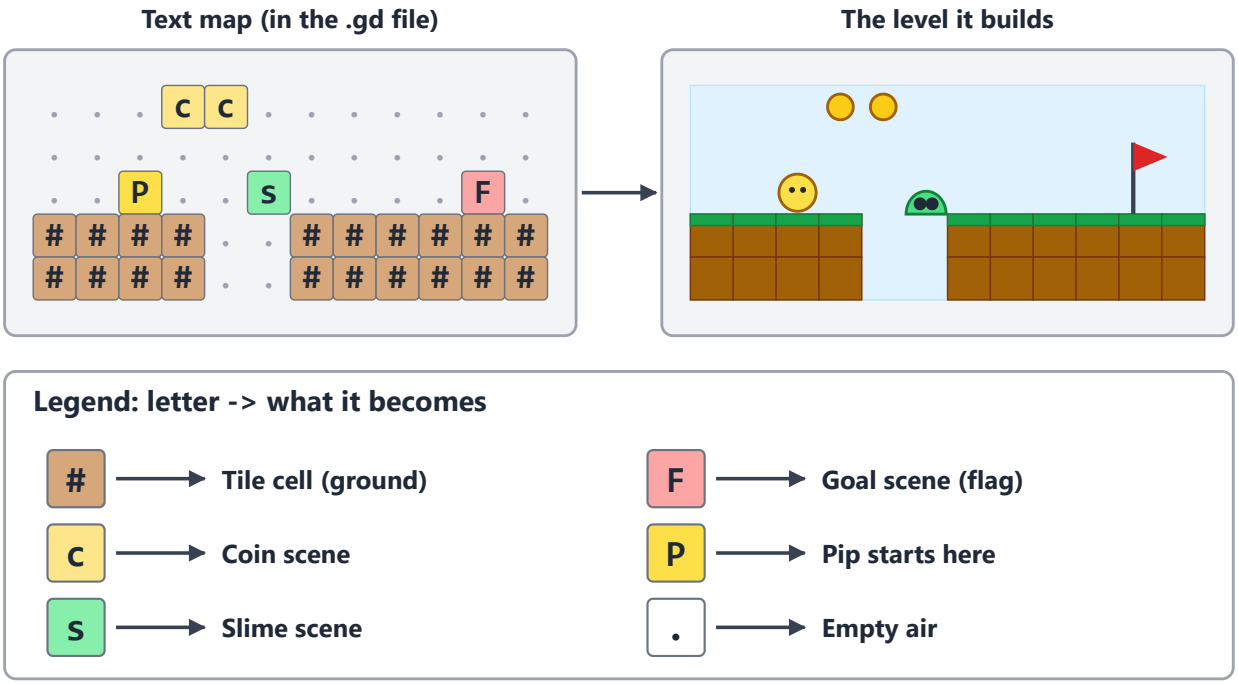


Figure 13.4: The level script reads the text. Each `#` becomes a tile cell in the TileMapLayer; each other letter becomes an instance of a scene at the same grid position.

- `Image.create(width, height, false, format)` makes a blank picture in memory. The second `false` means "no mipmaps" (smaller copies used for distant 3D objects, irrelevant here).
- We `fill` it brown, then use two nested loops to sprinkle darker pixels. The expression `(x * 7 + y * 13) % 11 == 0` is true for about one pixel in eleven, scattered in a regular but natural-looking pattern. `%` is the remainder operator you met in Chapter 4.
- The second tile, on the right half of the picture, gets a green cap on its top eight rows and a darker green band under it.
- `ImageTexture.create_from_image()` turns the picture into a texture the GPU can draw.
- A `TileSet` needs a **physics layer** for collisions. `add_physics_layer()` creates layer number 0 inside the tile set. Its collision layer defaults to 1, which our project named `world`. We add the physics layer **before** adding the atlas source so the tiles are created with room for collision data.
- A `TileSetAtlasSource` ("atlas" means one big picture holding many tiles) is told its texture and the size of one tile. `create_tile(Vector2i(x, y))` declares that the cell at column x, row y of the picture is a tile.
- Finally each tile gets a square collision polygon. Tile collision points are measured from the **center** of the tile, hence `-h` to `h`.

The function is `static`, so you call it without creating a `TileFactory` object: `TileFactory.make_tile_set()`.

> **Why it works:** Everything the editor does when you drag an image into the TileSet panel is available as a function. Reading the "Tiles" and "Physics" sections of the TileSet docs in the Further reading list shows the same calls.

The level script

Create `scripts/level.gd`. It is attached to a plain `Node2D`. First the scenes that will use it.

Make `scenes/coin.tscn`, `scenes/spike.tscn` and `scenes/goal.tscn` as follows (details in 13.6 and 13.7). Make `scenes/slime.tscn` (13.6). Then the level script:

```
# scripts/level.gd  (attached to the root Node2D of level_1.tscn and level_2.tscn)
extends Node2D
## Builds a level from a text map. Letters: # ground, P player start, c coin,
## s slime, ^ spike, F finish flag, . empty.

signal coin_collected
signal player_died
signal level_completed

const TILE: int = 32
const PLAYER_SCENE: PackedScene = preload("res://scenes/player.tscn")
const SLIME_SCENE: PackedScene = preload("res://scenes/slime.tscn")
const COIN_SCENE: PackedScene = preload("res://scenes/coin.tscn")
const SPIKE_SCENE: PackedScene = preload("res://scenes/spike.tscn")
const GOAL_SCENE: PackedScene = preload("res://scenes/goal.tscn")

@export var level_title: String = "Level"
@export_multiline var layout: String = ""

var player: Player
var coins_in_level: int = 0

@onready var ground: TileMapLayer = $Ground


func _ready() -> void:
    ground.tile_set = TileFactory.make_tile_set()
    var rows: PackedStringArray = layout.split("\n")
    var width: int = 0
    for row in rows:
        width = maxi(width, row.length())
    var start: Vector2 = Vector2(TILE, TILE)

    # Pass 1: solid ground tiles.
    for y in rows.size():
        for x in rows[y].length():
            if rows[y][x] == "#":
                var above_is_solid: bool = y > 0 and x < rows[y - 1].length() and rows[y - 1][x] == "#"
                var atlas_x: int = 0 if above_is_solid else 1
                ground.set_cell(Vector2i(x, y), 0, Vector2i(atlas_x, 0))

    # Pass 2: things that live on the ground.
    for y in rows.size():
        for x in rows[y].length():
            var center: Vector2 = Vector2(x * TILE + TILE / 2.0, y * TILE + TILE / 2.0)
            match rows[y][x]:
                "P":
                    start = center
                "c":
                    var coin: Node2D = COIN_SCENE.instantiate()
                    coin.position = center
                    coin.collected.connect(_on_coin_collected)
                    add_child(coin)
                    coins_in_level += 1
                "s":
                    var slime: Node2D = SLIME_SCENE.instantiate()
                    slime.position = center + Vector2(0, 6)
                    add_child(slime)
                "^":
                    var spike: Node2D = SPIKE_SCENE.instantiate()
                    spike.position = center
                    add_child(spike)
                "F":
                    var goal: Node2D = GOAL_SCENE.instantiate()
                    goal.position = center
                    goal.reached.connect(_on_goal_reached)
                    add_child(goal)
```

```
    # The player goes in last so it draws on top.
    player = PLAYER_SCENE.instantiate()
    player.position = start
    player.kill_y = rows.size() * TILE + 120.0
    player.died.connect(_on_player_died)
    add_child(player)

    var camera: Camera2D = player.get_node("Camera2D")
    camera.limit_left = 0
    camera.limit_top = 0
    camera.limit_right = width * TILE
    camera.limit_bottom = rows.size() * TILE


func _on_coin_collected() -> void:
    coin_collected.emit()


func _on_player_died() -> void:
    player_died.emit()


func _on_goal_reached() -> void:
    if not player.is_dead:
        level_completed.emit()
```

Read the script in four pieces.

The setup. `layout.split("\n")` cuts the text at each line break and gives an array of lines. We find the widest line so that we know how wide the level is.

Pass 1: tiles. Two nested loops visit every character. `rows[y][x]` means "line number y, character number x". For each `#` we call `ground.set_cell(coords, source_id, atlas_coords)`. The first argument is the grid position. The second is `0`, the source we added. The third picks which tile of the picture: `Vector2i(0, 0)` is plain dirt, `Vector2i(1, 0)` is grass-topped. The line with `above_is_solid` looks at the character directly above; if it is also `#`, we are underground and use dirt, otherwise we are on the surface and use grass. This is a tiny taste of "autotiling", which Chapter 12 did with terrain sets.

Pass 2: objects. We need a second loop because objects should be added after all tiles. `match` is GDScript's tidy replacement for a long `if ... elif` chain: it compares the character with each pattern. The position `center` is the middle of the grid cell in pixels. Everything here follows the same recipe: `instantiate()` the scene, set its `position`, connect a signal, `add_child()`. The slime is nudged 6 pixels down so its feet sit on the floor.

Pip, camera and limits. Pip is added last, so he is drawn on top. `kill_y` is set to 120 pixels below the bottom edge. The `Camera2D` that lives inside `player.tscn` has `limit_*` properties. They stop the camera at the edges of the map, so you never see empty space past the level.

`@export_multiline var layout` creates a text box in the Inspector where the map can be edited. Create `scenes/level_1.tscn`: a `Node2D` named `Level1` with the script attached and a child `TileMapLayer` named `Ground` (leave its TileSet empty; the script builds it). In the Inspector set **Level Title** to `Green Meadow` and paste the map below into **Layout**.

```
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
```

```
.............................cc.........................
............................####........................
.......................cc.............................
.........cc............####.........c........c..c.........
........####.......c................###...c...............
.....................................................
..P.........s.ccc............s.................^^...s....F..
###################...####################...################
###################...####################...################
```

Each line is exactly 60 characters and there are 17 lines. 17 lines of 32 pixels is 544 pixels, which fills the 540-pixel-high window. The game scrolls sideways across 60 × 32 = 1,920 pixels.

> **Warning:** Keep all lines the same length. If one line is shorter, its right end is treated as empty air, which is a silent bug. The test script checks this for you.

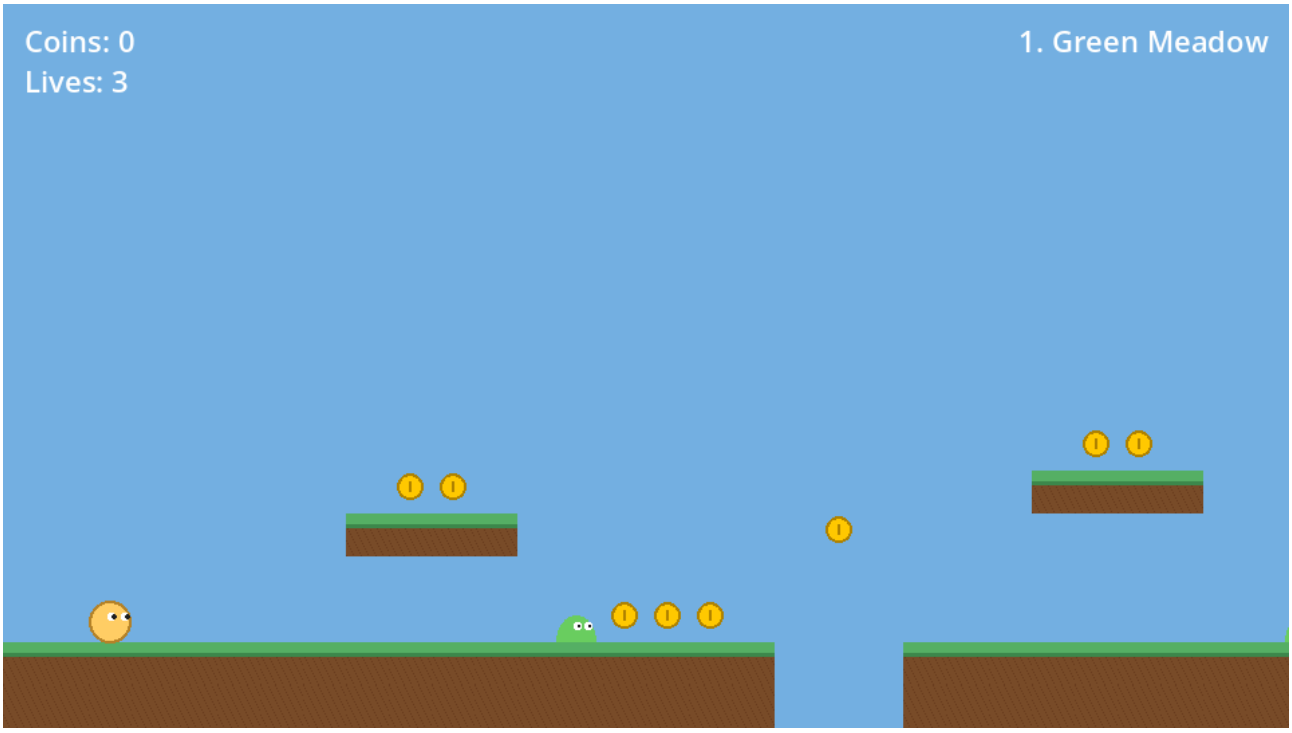


Figure 13.5: Level 1: Pip at the left, a coin arc over the first slime, platforms, a gap and the flag far to the right.

Level 2

Level 2 uses the same script. Make `scenes/level_2.tscn` with the same structure, **Level Title** `Slime Caves` and this map:

```
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.............................c.......................
............................###.....c.....
.........................c.............###.....
.......................###.....................c.....
...............sc.....................s.....###....
...........cc...###.....c.............c...####....c.....
.....................................................
```

```
..P...s.............s.........s...^^...................F..
##########....########.....###########...#########...#######
##########....########.....###########...#########...#######
```

Level 2 has four gaps (all 4 tiles or fewer), floating stepping stones, a slime living on a platform, and spikes in the last stretch. We did the arithmetic of 13.3 for it: the widest gap is 128 pixels and Pip's jump covers about 152.

Now the small objects.

13.6 Enemies, coins and hazards

The slime

A slime does three things: it falls under gravity, walks in one direction, and turns around when it hits a wall or when the floor ahead ends. And it is dangerous unless Pip lands on it.

Scene `scenes/slime.tscn`:

```
Slime (CharacterBody2D)         layer: 3 (enemies)   mask: 1 (world)
  CollisionShape2D              RectangleShape2D 28 x 20
  LedgeRay (RayCast2D)          Target Position (0, 24), Collision Mask: 1 (world)
  Hitbox (Area2D)               layer: none    mask: 2 (player)
    CollisionShape2D            RectangleShape2D 30 x 22
```

A **RayCast2D** is an invisible line. It checks whether the line crosses something solid. The `LedgeRay` starts at the slime and points 24 pixels down. We will place it a little ahead of the slime. If the ray hits nothing, there is no floor ahead: a ledge. The slime then turns.

```gdscript
# scripts/slime.gd  (attached to Slime)
extends CharacterBody2D
## A slime that patrols back and forth. Jump on its head to squash it.

@export var speed: float = 60.0
@export var gravity: float = 1500.0

var direction: int = -1
var squashed: bool = false

@onready var ledge_ray: RayCast2D = $LedgeRay
@onready var hitbox: Area2D = $Hitbox


func _ready() -> void:
    hitbox.body_entered.connect(_on_hitbox_body_entered)
    _update_ray()


func _physics_process(delta: float) -> void:
    if squashed:
        return
    if not is_on_floor():
        velocity.y += gravity * delta
    velocity.x = direction * speed
    move_and_slide()
    # Turn around at a wall, or when there is no floor ahead.
    if is_on_wall() or (is_on_floor() and not ledge_ray.is_colliding()):
        direction = -direction
        _update_ray()
        queue_redraw()


func _update_ray() -> void:
```

```
    ledge_ray.position.x = direction * 16.0


func _on_hitbox_body_entered(body: Node2D) -> void:
    if squashed or not body is Player:
        return
    var player: Player = body
    var from_above: bool = player.velocity.y > 0.0 and player.global_position.y < global_position.y - 12.0
    if from_above:
        player.bounce()
        squash()
    else:
        player.die()


func squash() -> void:
    squashed = true
    $CollisionShape2D.set_deferred("disabled", true)
    hitbox.set_deferred("monitoring", false)
    var tween: Tween = create_tween()
    tween.tween_property(self, "scale:y", 0.1, 0.12)
    tween.tween_callback(queue_free)


func _draw() -> void:
    var outline: PackedVector2Array = PackedVector2Array()
    for i in 17:
        var angle: float = PI + PI * i / 16.0
        outline.append(Vector2(cos(angle) * 15.0, 10.0 + sin(angle) * 20.0))
    draw_colored_polygon(outline, Color("6bcf63"))
    var eye_x: float = 5.0 * direction
    draw_circle(Vector2(eye_x - 4.0, -2.0), 3.0, Color.WHITE)
    draw_circle(Vector2(eye_x + 4.0, -2.0), 3.0, Color.WHITE)
    draw_circle(Vector2(eye_x - 4.0 + direction, -2.0), 1.5, Color("222222"))
    draw_circle(Vector2(eye_x + 4.0 + direction, -2.0), 1.5, Color("222222"))
```

The important part is the stomp test in `_on_hitbox_body_entered`. When Pip's body overlaps the hitbox, we must decide: stomp or ouch? Figure 13.6 shows the rule.

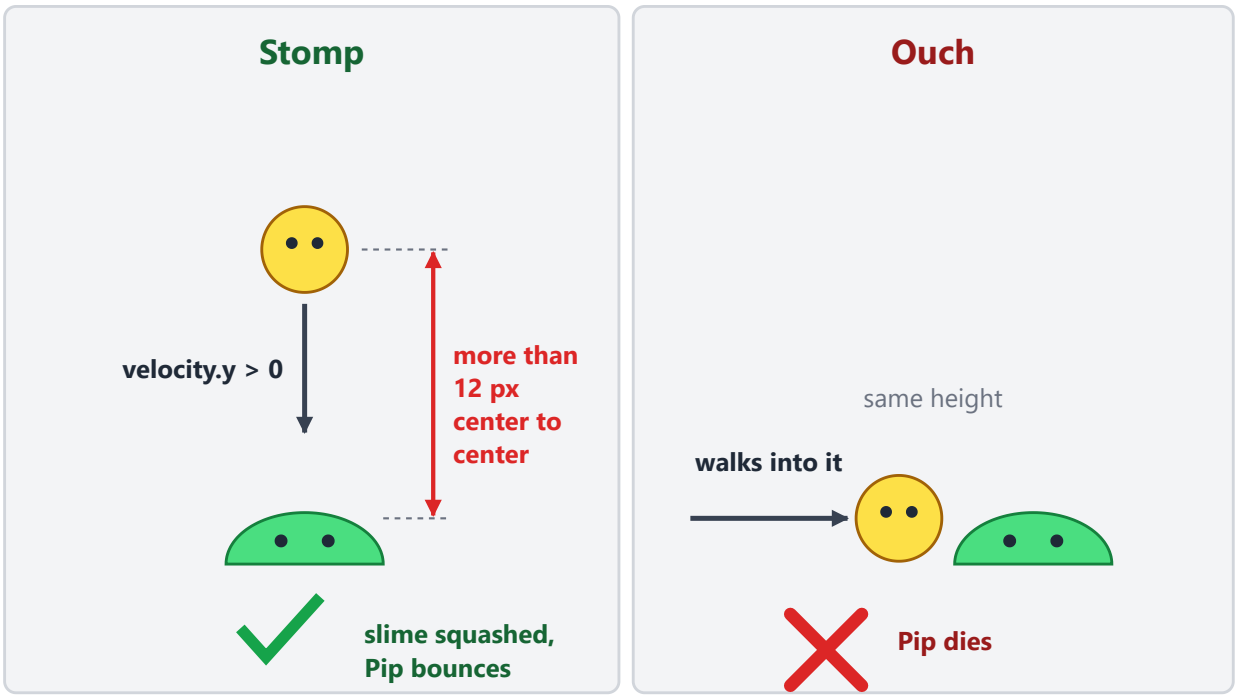


Figure 13.6: Two touches. Left: Pip is moving down and his center is more than 12 pixels above the slime's center, so it is a stomp. Right: Pip arrives from the side, so the slime wins.

We use two conditions joined by `and`:

1. `player.velocity.y > 0.0`: Pip is moving **down** (positive y is down).
2. `player.global_position.y < global_position.y - 12.0`: Pip's center is clearly **above** the slime's center. `global_position` is the position in the world, not relative to a parent.

If both are true, we call `player.bounce()` (a small hop upward) and `squash()`. Otherwise Pip is hurt. Both conditions matter. Only checking height would let Pip kill a slime while rising through it. Only checking velocity would let a falling Pip who brushes the side of a slime win.

`squash()` has two lines you may not recognise. `set_deferred("disabled", true)` changes the property at the end of the current physics frame. Godot does not allow changing collision shapes in the middle of a collision callback, and trying produces an error. `set_deferred` is the safe way. Remember it: **when you change physics objects from inside a physics signal, defer it.** The tween squishes the slime flat and `tween_callback(queue_free)` removes it when the squish is done.

> **Why it works:** The patrol logic is just two `if`s: `is_on_wall()` is true if the last move bumped into something to the side, and the ray tells us about the floor ahead. No complicated AI is needed for a satisfying enemy. Chapter 21 will build smarter ones with state machines.

The coin

`scenes/coin.tscn`:

```
Coin (Area2D)                  layer: 4 (pickups)   mask: 2 (player)
  CollisionShape2D             RectangleShape2D 18 x 18
```

```gdscript
# scripts/coin.gd  (attached to Coin)
extends Area2D
## A coin. Tells the world it was collected, then disappears.

signal collected

var _taken: bool = false


func _ready() -> void:
    body_entered.connect(_on_body_entered)
    # A gentle bob up and down, forever.
    var tween: Tween = create_tween().set_loops()
    tween.tween_property(self, "position:y", position.y - 4.0, 0.6).set_trans(Tween.TRANS_SINE)
    tween.tween_property(self, "position:y", position.y + 4.0, 0.6).set_trans(Tween.TRANS_SINE)


func _on_body_entered(body: Node2D) -> void:
    if _taken or not body is Player:
        return
    _taken = true
    collected.emit()
    queue_free()


func _draw() -> void:
    draw_circle(Vector2.ZERO, 9.0, Color("ffcc00"))
    draw_arc(Vector2.ZERO, 9.0, 0.0, TAU, 24, Color("b38600"), 2.0)
    draw_line(Vector2(0, -4), Vector2(0, 4), Color("b38600"), 2.0)
```

The coin does not know about the score. It only shouts "collected!" Whoever listens decides what that means. That is the point of signals, and it is why you could reuse this coin in a game with no score at all.

The `_taken` flag guards against a tricky case: if two bodies overlap on the same frame, `body_entered` could fire twice and count a coin twice.

The tween moves the coin 4 pixels up, then 4 pixels down, and `set_loops()` repeats this forever. Because `set_loops()` was called with no number, it loops infinitely. The easing `TRANS_SINE` makes the motion slow down at the ends like a real bob.

Spikes

`scenes/spike.tscn`: an `Area2D` named `Spike` with layer 4 and mask 2, and one `CollisionShape2D` (`RectangleShape2D` 24 x 14) positioned at (0, 8).

```
# scripts/spike.gd  (attached to Spike)
extends Area2D
## A spike. Touching it is deadly.


func _ready() -> void:
    body_entered.connect(_on_body_entered)


func _on_body_entered(body: Node2D) -> void:
    if body is Player:
        (body as Player).die()


func _draw() -> void:
    var points: PackedVector2Array = PackedVector2Array([Vector2(-14, 14), Vector2(0, -2), Vector2(14,
14)])
    draw_colored_polygon(points, Color("c9d1d9"))
    draw_polyline(PackedVector2Array([Vector2(-14, 14), Vector2(0, -2), Vector2(14, 14), Vector2(-14,
14)]), Color("6e7681"), 2.0)
```

The expression `(body as Player)` is a **cast**: it tells GDScript "treat this `Node2D` as a `Player`", which unlocks the `die()` method for the type checker. The `is` check just before makes it safe.

The flag

`scenes/goal.tscn`: an `Area2D` named `Goal`, layer 4, mask 2, with a `RectangleShape2D` 20 x 64 at (0, -16).

```
# scripts/goal.gd  (attached to Goal)
extends Area2D
## The finish flag.

signal reached


func _ready() -> void:
    body_entered.connect(_on_body_entered)


func _on_body_entered(body: Node2D) -> void:
    if body is Player:
        reached.emit()


func _draw() -> void:
    draw_line(Vector2(0, 16), Vector2(0, -48), Color("8d99ae"), 4.0)
    draw_colored_polygon(PackedVector2Array([Vector2(2, -48), Vector2(28, -38), Vector2(2, -28)]),
Color("ef476f"))
```

You now have four little objects that follow the same shape: an `Area2D`, a `body_entered` connection, a check `body is Player`, and either a signal or a direct call. Once you see the shape you can invent new ones in minutes: a spring that bounces Pip, a heart that gives a life, a key and a locked door.

> **Try it:** Make a `spring.tscn`: an `Area2D` on layer 4/mask 2. When Pip enters, call `body.velocity.y = -900.0`. Add an `S` letter to the legend in `level.gd` and place a few in level 1. Where do you need to change `level.gd`? (Add a `SPRING_SCENE` constant, and one more `match` branch.)

13.7 The game controller: lives, coins and two levels

The levels know how to build themselves and report events. Now we need something above them to decide what the events mean. Create the main scene `scenes/main.tscn`:

```
Main (Node)                          script: game.gd
  Background (CanvasLayer)           Layer: -1
    Sky (ColorRect)                  Layout: Full Rect, Color: a sky blue
  LevelHolder (Node2D)
  HUD (CanvasLayer)
    CoinLabel (Label)                top-left, font size 22
    LivesLabel (Label)               below the coin label
    LevelLabel (Label)               top-right, right-aligned
    MessageLabel (Label)             centered, font size 40, centered text
```

Set **Project Settings > Application > Run > Main Scene** to `scenes/main.tscn`.

Two `CanvasLayer` nodes appear. A `CanvasLayer` draws its children on a separate "sheet" that does not move with the camera. The HUD sheet stays glued to the screen while the world scrolls. The Background sheet has `layer = -1`, so it is drawn behind the world.

```gdscript
# scripts/game.gd  (attached to Main)
extends Node
## The game controller: loads levels, tracks lives and coins, shows the HUD.

const LEVELS: Array[PackedScene] = [
    preload("res://scenes/level_1.tscn"),
    preload("res://scenes/level_2.tscn"),
]
const START_LIVES: int = 3

var level_index: int = 0
var lives: int = START_LIVES
var coins: int = 0
var coins_at_level_start: int = 0
var current_level: Node2D
var waiting_for_restart: bool = false

@onready var level_holder: Node = $LevelHolder
@onready var coin_label: Label = $HUD/CoinLabel
@onready var lives_label: Label = $HUD/LivesLabel
@onready var level_label: Label = $HUD/LevelLabel
@onready var message_label: Label = $HUD/MessageLabel


func _ready() -> void:
    _start_game()


func _unhandled_input(event: InputEvent) -> void:
    if waiting_for_restart and event.is_action_pressed("jump"):
        _start_game()
```

```
func _start_game() -> void:
    waiting_for_restart = false
    lives = START_LIVES
    coins = 0
    level_index = 0
    _load_level(level_index)


func _load_level(index: int) -> void:
    if current_level:
        current_level.queue_free()
    coins_at_level_start = coins
    current_level = LEVELS[index].instantiate()
    current_level.coin_collected.connect(_on_coin_collected)
    current_level.player_died.connect(_on_player_died)
    current_level.level_completed.connect(_on_level_completed)
    level_holder.add_child(current_level)
    message_label.text = ""
    _update_hud()


func _update_hud() -> void:
    coin_label.text = "Coins: %d" % coins
    lives_label.text = "Lives: %d" % lives
    level_label.text = "%d. %s" % [level_index + 1, current_level.level_title]


func _on_coin_collected() -> void:
    coins += 1
    _update_hud()


func _on_player_died() -> void:
    lives -= 1
    _update_hud()
    if lives <= 0:
        message_label.text = "Game over\nPress Jump to try again"
        waiting_for_restart = true
        return
    await get_tree().create_timer(1.0).timeout
    coins = coins_at_level_start     # coins from the failed try do not count
    _load_level(level_index)


func _on_level_completed() -> void:
    if level_index + 1 < LEVELS.size():
        level_index += 1
        message_label.text = "Level complete!"
        await get_tree().create_timer(1.2).timeout
        _load_level(level_index)
    else:
        message_label.text = "You win!  Coins: %d\nPress Jump to play again" % coins
        waiting_for_restart = true
```

Here is how the pieces cooperate.

- `LEVELS` is a list of `PackedScene` resources, our two levels. Adding a third level means one more `preload` line and one new scene.
- `_load_level` removes the old level (`queue_free()`), creates the new one with `instantiate()`, and connects **three signals**. The syntax `signal_name.connect(function)` is the Godot 4 way; the function name is passed without parentheses because we are handing over the function itself, not calling it.
- `_on_player_died` subtracts a life. If there are lives left, the line `await get_tree().create_timer(1.0).timeout` pauses this function for a second without freezing the game, then reloads the same level. `await` is covered in depth in Chapter 15. If lives hit zero, we show "Game over" and wait for a jump press.

- `_on_level_completed` moves to the next level after a short pause, or shows the winning message if the last level is done.
- `coins_at_level_start` is a small fairness rule: if you die, the coins from that attempt are forgotten, because the level reloads with all its coins back. Without it you could farm coins by dying and re-collecting.
- `"Coins: %d" % coins` is string formatting: `%d` is replaced by the integer on the right of `%`. For two or more values we pass an array: `"%d. %s" % [level_index + 1, title]`.

> **Warning:** In `_load_level` the old level is only *scheduled* for removal with `queue_free()`. It disappears at the end of the frame. Our new level is added in the same frame, so for a moment there are two levels. This is harmless here because the old level's Pip is already dead or the level is complete, but in other games you may want to `remove_child()` first.

Why the design uses signals

Look at what `Coin`, `Slime`, `Spike` and `Goal` do *not* contain: the words `score`, `lives`, `HUD`, `Main`. They could be dropped into any game. The `Level` collects them and passes three simple facts upward. The `Main` scene is the only script that understands the rules of *this* game. This is the pay-off of the "signals up" rule and it is the main lesson of the chapter. If you wanted to change the rules to "collect all coins to open the flag", only `game.gd` and `level.gd` would change.

13.8 Testing a game without playing it

Playing your game is the best test, but it is slow, and it is easy to forget to re-check something you changed an hour ago. Godot lets you write a script that runs a scene *without a window*, simulates input, and checks what happens. The project contains `tests/test_platformer.gd`. Run it from a terminal in the project folder (adjust the path to wherever your Godot program is):

```
godot --headless --path . --script res://tests/test_platformer.gd
```

You should see lines like:

```
PASS  both levels have equal-length rows
PASS  player lands on the floor
PASS  full jump rises 70-110 px (got 95)
PASS  short hop is lower than full jump (got 43)
PASS  stomp squashes slime
PASS  falling into a pit kills Pip
PASS  reaching the flag completes the level
FAILURES: 0
```

A test script extends `SceneTree` (the object that runs the whole game loop) instead of `Node`. It adds a level to the tree, then uses `Input.action_press("jump")` to pretend a key is held and `await physics_frame` to wait for the engine to run a physics step. After waiting it compares numbers. Here is the jump check, as a **fragment** from the test:

```gdscript
# Fragment from tests/test_platformer.gd
var floor_y: float = player.global_position.y
Input.action_press("jump")
var peak: float = 0.0
for i in 60:
    await physics_frame
```

```
    peak = maxf(peak, floor_y - player.global_position.y)
Input.action_release("jump")
check("full jump rises 70-110 px (got %.0f)" % peak, peak > 70.0 and peak < 110.0)
```

The test measures the highest point reached over one second and checks that the jump height is near the 90 pixels the formula predicted. If you changed `gravity` and forgot to adjust level 2, a check like this tells you immediately. Professional studios test games this way. You do not have to; but it is a good habit, and Chapter 22 returns to testing and debugging.

Project 13: Pip's Platformer

Goal: Build and finish the two-level platformer. Time: 60 to 90 minutes. The finished project is in `projects/ch13-platformer/` if you want to compare.

Work in this order. After each step, run the game (or the small test scene) and check the result before moving on. Small steps that work are faster than big steps that do not.

1. **Create the project** `ch13-platformer`. Set the window size, stretch mode, layer names and the three Input Map actions from 13.1.
2. **Build** `player.tscn` and write `player.gd` from 13.2. Use a throwaway test scene with a wide `StaticBody2D` floor. Check: Pip falls, lands, runs, and the eyes turn.
3. **Add jumping**: variable height, coyote time and buffer (13.3, 13.4). Check: a tap hops, a hold jumps high, you can jump just after leaving an edge.
4. **Write** `tile_factory.gd` (13.5). Check: in the test scene, add a `TileMapLayer` and, in a tiny script, set its `tile_set` to `TileFactory.make_tile_set()` and call `set_cell(Vector2i(2, 5), 0, Vector2i(1, 0))`. A grassy tile appears and Pip stands on it.
5. **Make the small objects**: `coin`, `spike`, `goal` and `slime` scenes with their scripts (13.6). Drop each in your test scene by hand and check that the touch works: coin vanishes, spike kills, slime patrols and can be stomped.
6. **Write** `level.gd` and `level_1.tscn` with the map from 13.5. Run the level scene on its own with **F6** (Run Current Scene). Check: you can walk the whole level and reach the flag.
7. **Make** `level_2.tscn` with its map. Play it with F6 as well.
8. **Build** `main.tscn` **and** `game.gd` (13.7). Set it as the main scene. Press **F5** and play both levels in order.
9. **Test the failure paths**: fall in a pit three times and see "Game over"; press Jump and see it restart. Win and see "You win!".
10. **Run the headless test** from 13.8 and confirm `FAILURES: 0`.

Expected result: The game opens at level 1 "Green Meadow". Pip runs, jumps and can squash slimes. Touching a slime from the side, hitting spikes, or falling into a pit costs a life and restarts the level after one second. Reaching the flag shows "Level complete!" and loads "Slime Caves". Reaching the second flag shows "You win!" with your coin total. Losing all three lives shows "Game over". Pressing Jump starts again.

Chapter summary

- Gravity is `velocity.y += gravity * delta` each physics frame; `move_and_slide()` moves the body and stops it on floors and walls.
- Jump height is `v² / (2g)` and air time is `2v / g`; use them to design gaps and steps before you build a level.
- Variable jump height, coyote time and a jump buffer cost about twenty lines and make a jump feel much better.

- A `TileSet` can be built with code: paint an `Image`, wrap it in a `TileSetAtlasSource`, add a physics layer and give each tile a collision polygon.
- A text map is a cheap, shareable way to define levels; a level script loops over it twice, once for tiles and once for objects.
- A patrolling enemy needs only gravity, a direction, wall detection and a `RayCast2D` that looks for floor ahead.
- A stomp is "moving down and clearly above"; use `set_deferred` when changing physics objects inside physics signals.
- Small objects emit signals, the level relays them, and one game controller turns them into rules. Signals go up, calls go down.

Review questions

1. In Godot's 2D coordinates, does a negative `velocity.y` mean the character is moving up or down?
2. True or false: `is_on_floor()` reports what happened during the most recent `move_and_slide()`.
3. A character jumps with speed 600 px/s under gravity 2000 px/s². How high does the jump go?
4. Using the same numbers and a run speed of 250 px/s, how far sideways does the character travel during one complete jump (up and back to the starting height)?
5. Name the two problems that coyote time and the jump buffer solve. Which one forgives a *late* press, and which forgives an *early* press?
6. Multiple choice: How do we make a short tap give a short hop? (a) Lower gravity while the button is down. (b) Multiply the upward speed by a number below 1 when the button is released while rising. (c) Disable the jump after 0.1 seconds. (d) Increase `max_fall_speed`.
7. In the stomp test, why do we check *both* the downward velocity and the height difference?
8. Why does `squash()` use `set_deferred("disabled", true)` instead of simply `disabled = true`?
9. Multiple choice: Why do the coin, slime and spike scenes not change the score directly? (a) GDScript forbids it. (b) They emit signals, so they stay reusable and the controller decides the rules. (c) Areas cannot call functions. (d) It would make the game slower.
10. What does a `CanvasLayer` do, and why is the HUD placed in one?

Exercises

1. **Practice (easy):** Change `run_speed`, `jump_velocity` and `gravity` to make Pip feel like (a) a heavy, snappy arcade hero and (b) a floaty moon walker. For each, use the formulas of 13.3 to predict the jump height, then measure it with the headless test. Write your findings in a comment at the top of `player.gd`.
2. **Practice (easy):** Add a third heart-giving pickup: a red circle `Area2D` called `Heart` with a `collected` signal. Add `h` to the legend in `level.gd` and let `game.gd` give back one life (up to a maximum of five) when it is collected.
3. **Apply (medium):** Make a third level, `level_3.tscn`, using only the map text. Include one section that requires a variable-height jump (a low ceiling made of `#` blocks) and one enemy gauntlet. Add it to the `LEVELS` array.
4. **Apply (medium):** Make slimes drop a coin when stomped. Hint: the slime needs a `died` signal; the level connects it to a function that creates a coin at the slime's position. (This mirrors how the other objects work.)
5. **Challenge (hard):** Add a "dash" ability: pressing a new `dash` action (Shift) moves Pip at 600 px/s horizontally for 0.15 seconds, ignores gravity during the dash, and may only be used once until Pip touches the floor again. Use countdown timers, like coyote time. Add checks for it to the headless test.

Common mistakes

- **Pip falls through the floor.** Cause: the tiles have no collision, or layer/mask do not match. Fix: confirm the `TileSet` has a physics layer *before* tiles were added, tile collision layer is 1, and Pip's mask includes layer 1.
- **Pip jumps twice as high, or jumps every frame.** Cause: `is_action_pressed` was used instead of `is_action_just_pressed`. Fix: use the "just" version for one-time events.
- **A slime kills Pip even though he landed on it.** Cause: the stomp check ran when Pip was already too deep inside, or the velocity was reset by a floor collision. Fix: test velocity and height in the hitbox signal as shown, and keep the hitbox only slightly larger than the slime body.
- **An error "Can't change this state while flushing queries".** Cause: a collision shape or `monitoring` property was changed from inside a physics signal. Fix: use `set_deferred("disabled", true)`.
- **Scrolling stops short, or empty space appears at the right.** Cause: camera limits were not set or the map lines have different lengths. Fix: keep lines equal and set `limit_right = width * TILE` as shown.
- **The score label never changes.** Cause: the signal is connected in the wrong place or the HUD node path is wrong (`$HUD/CoinLabel`). Fix: print inside the receiving function to see whether it runs, then check the node names in the scene dock.

Further reading

- 2D movement overview: https://docs.godotengine.org/en/stable/tutorials/2d/2d_movement.html
- Using CharacterBody2D: https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html
- Using TileSets: https://docs.godotengine.org/en/stable/tutorials/2d/using_tilesets.html
- Using TileMaps: https://docs.godotengine.org/en/stable/tutorials/2d/using_tilemaps.html
- Ray-casting: https://docs.godotengine.org/en/stable/tutorials/physics/ray-casting.html
- Canvas layers: https://docs.godotengine.org/en/stable/tutorials/2d/canvas_layers.html
- Tween class reference: https://docs.godotengine.org/en/stable/classes/class_tween.html

Part III: Interfaces and game structure

Chapter 14: User Interface

Learning objectives

By the end of this chapter you will be able to:

- Explain how `Control` nodes differ from `Node2D` nodes and why menus use them.
- Position and resize interface elements with anchors, offsets and size flags so they work at any window size.
- Arrange elements automatically with `VBoxContainer`, `HBoxContainer`, `GridContainer`, `MarginContainer`, `CenterContainer` and `PanelContainer`.
- Use the common controls (`Label`, `Button`, `CheckButton`, `HSlider`, `OptionButton`, `LineEdit`, `ProgressBar`) and connect their signals to scripts.
- Create a `Theme` with `StyleBoxFlat` styles and a type variation, and apply it to a whole game from one place.
- Make a menu work with keyboard and gamepad through focus, and build a pause menu that works while the game is paused.

Key terms

- **Control**: the base class of every interface node. It has a rectangle (position and size), can receive mouse and keyboard input, and understands layout.
- **Anchor**: a point on the parent rectangle (given as a fraction from 0 to 1 on each axis) that a control stays attached to when the parent changes size.
- **Offset**: the distance in pixels from a control's edge to its anchor.
- **Container**: a control that decides the position and size of its children automatically.
- **Size flags**: per-child settings (Fill, Expand, Shrink Center and so on) that tell a container how much space the child wants.
- **Minimum size**: the smallest size a control accepts; containers never make a child smaller than this.
- **Theme**: a resource that stores the fonts, colors, sizes and box styles used by all controls below the node where it is applied.
- **StyleBox**: a recipe for drawing a rectangle background, such as a rounded colored box with a border. `StyleBoxFlat` draws one without any image.
- **Type variation**: a named alternative look for one control type inside a theme, such as a big yellow "TitleLabel" based on `Label`.
- **Focus**: the one control that currently receives keyboard and gamepad input, shown with a highlight.
- **Process mode**: a node setting that says whether it runs when the game tree is paused.
- **HUD**: the heads-up display, the information (health, score, lives) drawn over the game.

14.1 Controls: the interface family

So far you have drawn things in a world: sprites, bodies, tilemaps. They live in 2D space and the camera decides what you see. User interfaces are different. A menu must stay put on the screen when the camera moves. A button must react to the mouse. A row of buttons must fit when the player resizes the window or plays on a phone. Godot has a separate family of nodes for this job. All of them inherit from **`Control`**, and their icons are green in the scene dock (2D nodes are blue).

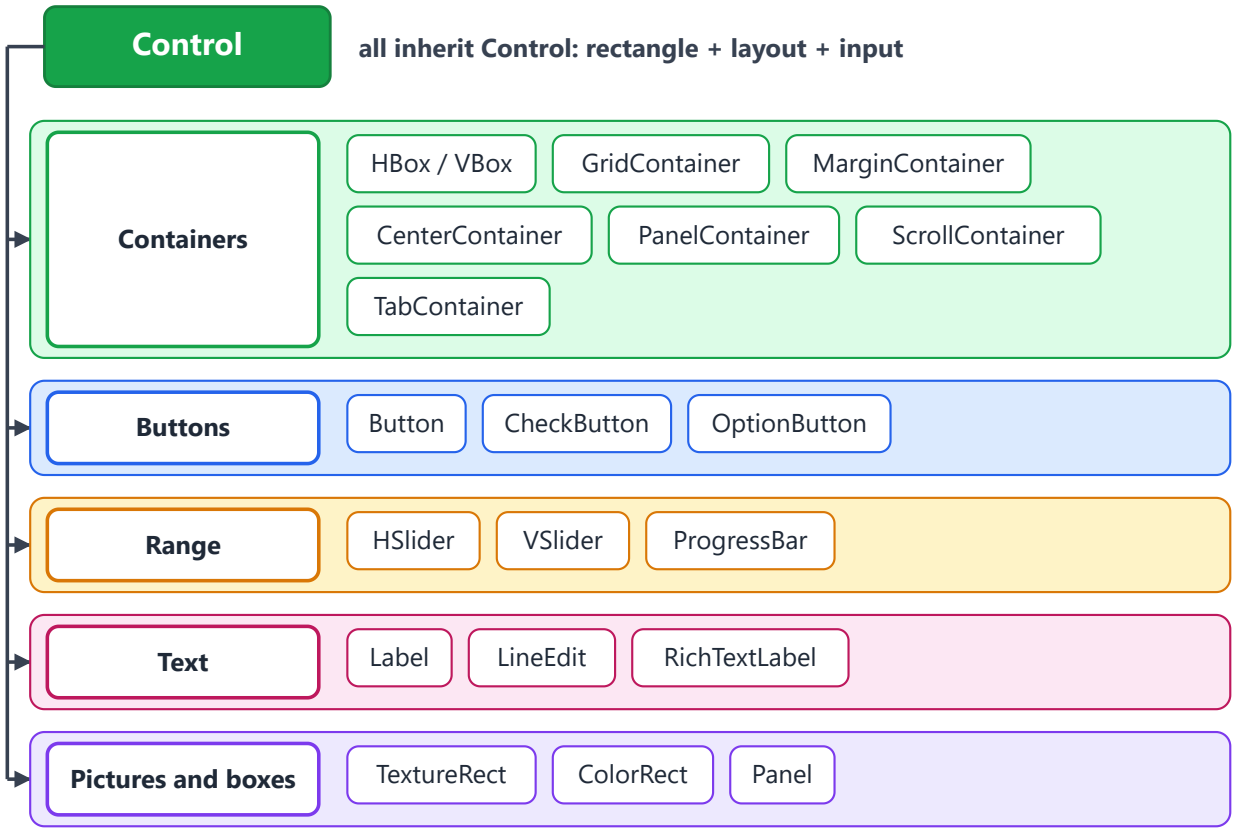


Figure 14.1: Everything green in Godot descends from Control. Containers arrange children, buttons react to clicks, and displays such as Label and ProgressBar show information.

A `Control` differs from a `Node2D` in three ways:

1. **It has a rectangle.** Instead of a single `position`, a Control has a position *and* a `size`. Its box is the clickable, drawable area.
2. **It has layout rules.** Anchors, offsets, minimum size and size flags describe how it behaves when its parent changes.
3. **It takes part in input and focus.** Clicks reach the control under the mouse, and keyboard keys reach the focused control. `mouse_filter` decides whether a control catches clicks (`Stop`), lets them pass through to things behind it (`Pass` or `Ignore`).

The simplest interface is a single `Label`. Make a new project `ch14-ui`, add a `Control` as the root node, add a `Label` as a child, and type some text into its **Text** property. Run it. You see the text in the top-left corner. Try dragging the label in the 2D editor: it moves, just like a sprite. The difference shows up as soon as the window gets resized, which is the next topic.

> **Note:** Control scenes are usually built entirely in the editor, because dragging and looking is faster than typing numbers. The chapter still shows the code that goes with each scene. When this text says "add a node", use the **Add Child Node** button, then type the node name into the Inspector's name field.

The base rule: let containers lay things out

A beginner's first instinct is to drag each control to the right spot. That works until the window changes size. The better habit, and the thread through this chapter, is:

> **Use containers for layout. Use anchors only for the big pieces. Use pixel positions almost never.**

The next two sections teach anchors (for the outer frame) and containers (for everything inside).

14.2 Anchors and offsets

Imagine a frame hung on a wall. You can describe where it sits in two ways. "Fifty centimetres from the left edge" does not change when the wall gets longer. "Always in the exact middle of the wall" does. Anchors are the second way.

Every Control has four **anchors**: `left`, `top`, `right`, `bottom`. Each is a number between 0 and 1: the fraction of the *parent's* width or height. Anchor 0 is the parent's left or top edge, 0.5 is the middle and 1 is the right or bottom edge. The control's four edges are then pushed away from those anchor points by four **offsets**, measured in pixels.

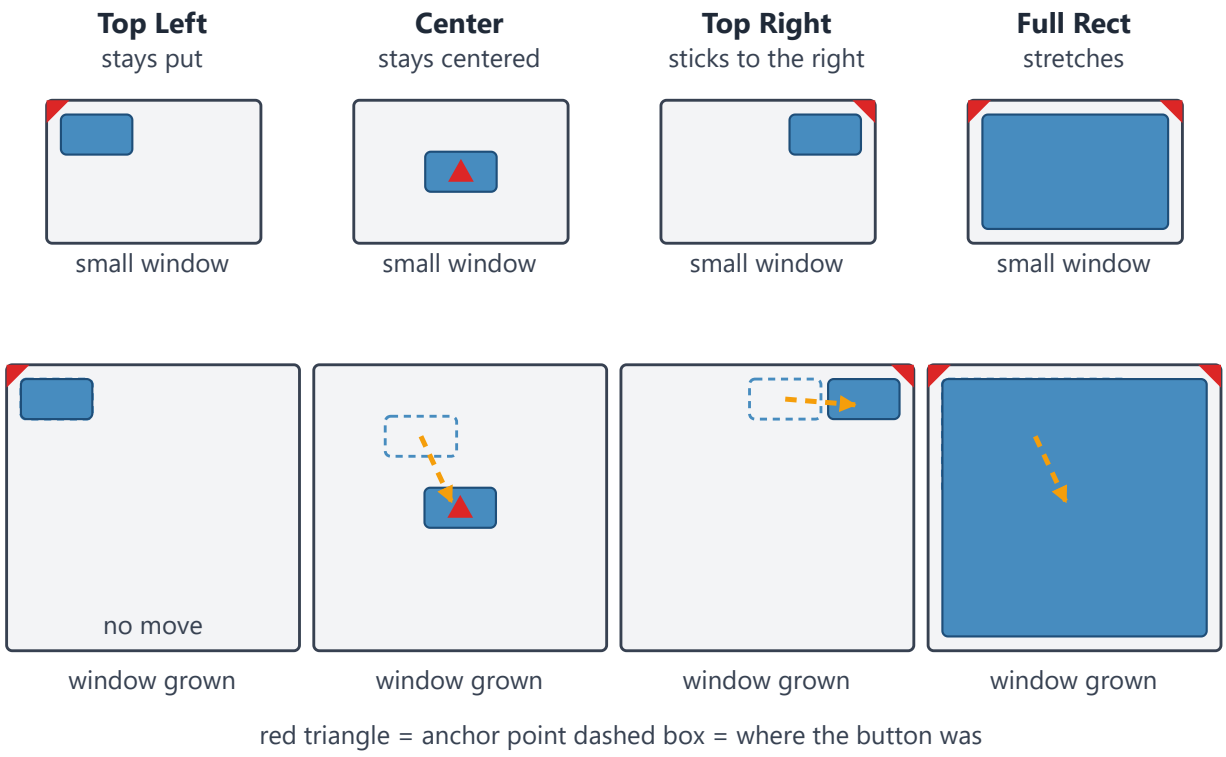


Figure 14.2: Top-left keeps its place; Center stays in the middle; Top Right sticks to the corner; Full Rect stretches with the parent.

You rarely type anchors. Select a Control and click the **Layout** menu (the little anchor icon) in the editor toolbar, or use the **Anchors Preset** in the Inspector. The common presets:

Preset	When to use it
Top Left	default; things that stay in the corner
Top Right	a score or level label
Center	a message in the middle of the screen
Full Rect	a background or a whole screen that fills its parent

Preset	When to use it
Top Wide	a bar across the top

For the screens in this chapter every root node is **Full Rect**. That means it always covers the whole window. Everything inside is arranged by containers.

Anchors are fractions of the *parent's* rectangle. The root `Control` of a scene sits inside the `Window` (the game window), so Full Rect means "same size as the window". A child of a smaller control uses that smaller control's rectangle.

> **Warning:** Anchors and offsets apply only when the parent is *not* a container. Inside a container the container is in charge, and the Inspector shows **Layout Mode: Container** with the anchor controls greyed out. This is the most common source of "why does my control not move?" questions.

Stretch settings

Anchors need help from the window. In **Project Settings > Display > Window > Stretch**, set **Mode** to `canvas_items`. This makes Godot scale the interface when the window is larger than the design size, so text and buttons stay in proportion. Set **Aspect** to `expand` so extra width or height shows more of the interface instead of black bars. In the example project the design size is 1152 by 648 pixels (the Godot default) and the settings are:

```
window/stretch/mode = canvas_items
window/stretch/aspect = expand
```

Try resizing the running game window. The menu stays centered and the sizes stay comfortable. If you set the stretch mode to `disabled`, the window grows but the UI stays small in the top-left. Both are valid; most games choose `canvas_items`.

14.3 Containers: layout that does the arithmetic

A **container** is a control whose job is to place and size its children. You give it children; it sets their position and size every time something changes. The children cannot disobey. You influence them only through their minimum size and size flags.

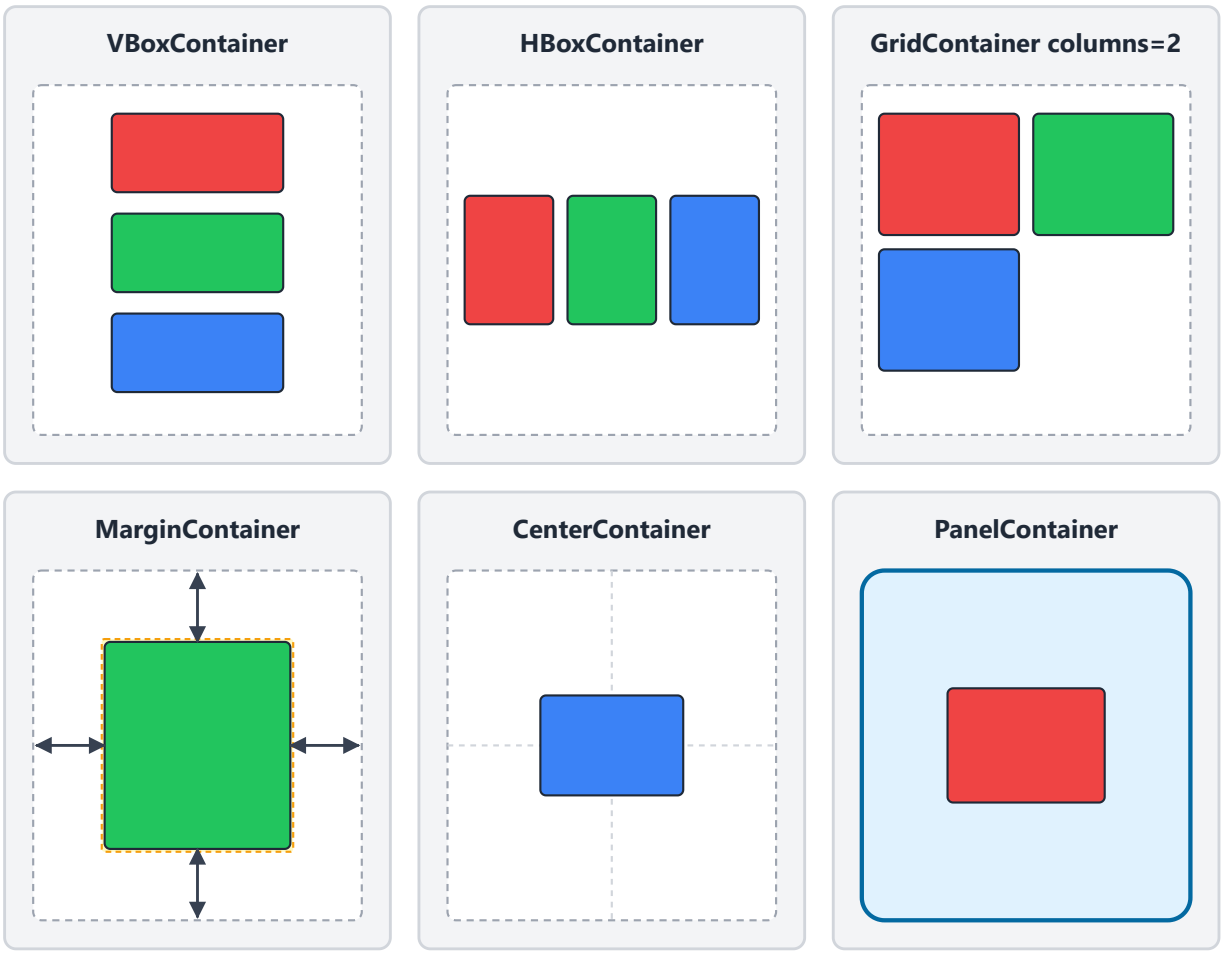


Figure 14.3: VBox stacks, HBox lines up, Grid fills rows and columns, Margin adds padding, Center centers, Panel draws a box behind.

The containers you will use every day

`VBoxContainer` stacks children vertically, top to bottom, with a gap between them (the `separation` constant). `HBoxContainer` is the same but in a row. Together these two make most layouts.

`GridContainer` places children in rows and columns. You set its `columns` property and it wraps automatically. A settings page is a grid with two columns: labels on the left, controls on the right. An inventory is a grid of buttons.

`MarginContainer` adds empty space around its child. It has four margin constants: left, top, right, bottom. Without it, text touches the edge of a panel and looks cramped.

`CenterContainer` keeps its child centered at its minimum size. It is the easiest way to center a menu.

`PanelContainer` draws a rectangular background (a `StyleBox`) and puts its child inside. It is the "card" or "window" of a UI.

`ScrollContainer` (not used in this chapter's project) gives a scrollable area when the child is bigger than the box. `TabContainer` shows one child at a time with tabs. `HSplitContainer` and `VSplitContainer` let the user drag a divider.

The classic menu recipe

Almost every centered menu uses this stack:

```
MainMenu (Control)                  Layout: Full Rect
  CenterContainer                   Layout: Full Rect
    VBoxContainer                   separation: 16
      Title (Label)
      PlayButton (Button)
      OptionsButton (Button)
      QuitButton (Button)
```

Read it from the inside out. The `VBoxContainer` stacks the labels and buttons. The `CenterContainer` keeps that stack in the middle of the screen. The root is Full Rect so the centering is relative to the whole window. If you resize the window, the menu stays centered with no code at all.

> **Try it:** Build exactly this tree. Press F6 to run the scene. Then drag the window edges. Next, select the `VBoxContainer` and, in the Inspector under **Theme Overrides > Constants**, tick **Separation** and change the number. The gaps change. Select `PlayButton` and set **Custom Minimum Size** x to `260`. Now the whole column is 260 pixels wide, because a VBox is as wide as its widest child, and the other buttons stretch to match.

Minimum size and size flags

Every control reports a **minimum size**: the smallest rectangle it needs. A `Label` needs enough room for its text. A `Button` needs its text plus padding. You can raise it with **Custom Minimum Size** (properties `custom_minimum_size`), never lower it below the content.

In a `BoxContainer` a child has **Size Flags** for each direction. The important ones are:

Flag	Meaning
Fill	grow to take the whole slot the container gives it
Expand	ask the container for extra space (leftover room is shared between expanding children)
Shrink Center / Begin / End	do not grow; sit centered, at the start, or at the end of the slot

The most useful trick is the **spacer**. To push one item to the left and another to the right of a bar, place an empty `Control` between them and give it the `Expand` horizontal flag:

```
TopBar (HBoxContainer)
  HealthBox (VBoxContainer)
  Hearts (HBoxContainer)
  Spacer (Control)                  Size Flags Horizontal: Fill + Expand
  ScoreLabel (Label)
```

The spacer swallows all leftover width, so the score is pushed to the right edge. This is exactly how the game screen of this chapter's project is arranged.

A grid works the same way. In the options screen a two-column `GridContainer` holds a label, then its control, then the next label, and so on. The children are placed row by row, so the order in the scene dock is the order on screen.

14.4 The controls you will use most

Here is the tour. All of them have a distinct job and a distinct signal. A **signal** is how a control tells you something happened (Chapter 5 introduced them, Chapter 15 goes deeper). Learning a control means learning three things: its main property, its main signal, and how to read its value.

Control	Shows or does	Main property	Main signal
`Label`	plain text	`text`	none
`Button`	a click target	`text`, `disabled`	`pressed`
`CheckButton` / `CheckBox`	on/off	`button_pressed`	`toggled(pressed)`
`HSlider` / `VSlider`	a number in a range	`value`, `min_value`, `max_value`, `step`	`value_changed(value)`
`OptionButton`	a drop-down list	`selected` (index)	`item_selected(index)`
`LineEdit`	one line of typed text	`text`, `placeholder_text`, `max_length`	`text_changed`, `text_submitted`
`ProgressBar`	a filled bar	`value`, `max_value`	none
`TextureRect`	a picture	`texture`	none
`ColorRect`	a solid color box	`color`	none
`RichTextLabel`	text with bold, colors and links	`text` (with BBCode)	`meta_clicked`

Labels and text

`Label` shows text. To make a wide message wrap onto several lines, set **Autowrap Mode** to `Word` and give the label a minimum width. To change the font size of just one label, use **Theme Overrides > Font Sizes > Font Size**. For all labels at once, use a theme (14.5).

Buttons

A `Button` emits `pressed` when clicked or when activated with Enter or the gamepad "accept" button while it has focus. Connect it in code:

```
# Fragment: inside _ready() of a script on a node that has a child named PlayButton
$PlayButton.pressed.connect(_on_play_button_pressed)

func _on_play_button_pressed() -> void:
    print("Play!")
```

A button can also have `toggle_mode` turned on, which makes it stay down like a switch, and a `tooltip_text`, which appears after the mouse rests on it for a moment. `disabled = true` greys it out and stops clicks. We use all three in the basket of the project.

Sliders, checkboxes, lists, text fields

These controls produce *values*. A slider emits `value_changed(value)` with the new number each time it moves. If you only need the current value, read `slider.value`. A good volume slider runs from 0 to 1 with a step of 0.05.

Sound volume is measured in **decibels** (dB), a logarithmic scale where 0 dB is "no change" and negative numbers are quieter. A slider that moves linearly would feel wrong, so we convert with `linear_to_db()`. Godot's audio buses (Chapter 11) have volume in dB:

```gdscript
# Fragment: connect the slider's value_changed signal to this function
func _on_volume_changed(value: float) -> void:
    AudioServer.set_bus_volume_db(0, linear_to_db(value))   # bus 0 is "Master"
    AudioServer.set_bus_mute(0, value <= 0.0)
```

`linear_to_db(0.5)` is about -6 dB, which sounds about half as loud. `linear_to_db(0)` is minus infinity, so we also mute the bus at zero.

An `OptionButton` starts empty. Fill it with `add_item("Easy")`, `add_item("Normal")` and so on, then read the chosen index from `selected`. A `CheckButton` is a switch; connect `toggled`. A `LineEdit` has a `text` property; `.strip_edges()` removes accidental spaces at both ends.

ProgressBar: bars that move smoothly

A `ProgressBar` shows `value` out of `max_value`. For a health bar, set the max to 100 and **Show Percentage** off. Setting `value` directly makes the bar jump. A tween makes it glide:

```gdscript
# Fragment: slide the health bar to a new value over a quarter of a second
var tween: Tween = create_tween()
tween.tween_property(health_bar, "value", float(health), 0.25)
```

You met `create_tween()` and `tween_property()` in Chapter 10. They work on any property of any object, including UI properties.

14.5 Themes: one look for the whole game

By default every Control uses Godot's grey editor-like style. Changing the look of every button one by one would be endless work. A **Theme** is a resource that holds the look for the whole game. You apply it once, to the root of a UI scene, and every control below it uses it.



Figure 14.4: The same scene before and after applying theme.tres. No node changed; only the theme.

A theme stores items organised by **type** (a control class such as `Button`) and by **kind**:

Kind	Examples
colors	`Button / font_color`, `Button / font_hover_color`
constants	`VBoxContainer / separation`

Kind	Examples
font sizes	`Button / font_size`, or the global `default_font_size`
styles (StyleBoxes)	`Button / normal`, `Button / hover`, `Button / pressed`, `PanelContainer / panel`

Controls ask the theme for items by name when they draw. A button asks "give me your `normal` style" while idle, `hover` while the mouse is over it, `pressed` while held, and `focus` for the outline when it has keyboard focus.

StyleBoxFlat: boxes without pictures

A **`StyleBoxFlat`** draws a colored rectangle in code. It has a background color, optional borders, rounded corners and content margins (padding inside the box). Because it needs no image, it fits our no-art rule perfectly. The properties you will touch the most:

- `bg_color`: the fill.
- `corner_radius_*`: how round each corner is.
- `border_width_*` and `border_color`: an outline.
- `content_margin_*`: padding between the box edge and the content.
- `draw_center`: turn off to get only the border (a perfect focus ring).

Building the theme in the editor

1. In the FileSystem dock, right-click the `ui` folder and choose **New Resource...**, then pick `Theme`. Save it as `ui/theme.tres`.
2. Double-click it. The **Theme editor** opens at the bottom. Click **Add Type** (or the plus) and type `Button`.
3. Under `Button`, add **Styles** called `normal`, `hover`, `pressed` and `focus`. For each, click the empty slot and choose **New StyleBoxFlat**. Click the new style to edit it: set `bg_color` and corner radii. For `focus`, switch off `draw_center`, set borders to 3 and `border_color` to a gold.
4. Add **Colors** `font_color` and `font_hover_color` for `Button`, **Font Sizes** if you like, and a `PanelContainer / panel` style.
5. In the Theme editor's top bar you can also set **Default Font Size**. It is `22` in our theme, which is comfortable at 1152 by 648.
6. Select the root node of your scene (`Main`) and, in the Inspector, drag `theme.tres` into the **Theme** property.

Instantly every button, panel and bar below the root looks different. If a child has its own **Theme** set, that wins over the parent's. And an individual **Theme Override** on one control wins over everything. The order, strongest first, is:

1. Theme overrides on the control itself.
2. A theme on the control, then on its parent, and up the tree.
3. The project's default theme.
4. Godot's built-in default.

A type variation: the title style

Sometimes you want a second look for the same class. The title of a menu is a `Label`, but 64 points tall and golden. Making an override on that one label works, but a **type variation** is cleaner and reusable. In the Theme editor, add a type named `TitleLabel`, set its **Base Type** to `Label`, and give it a `font_color` and `font_size`. Then, on any label, set **Theme Type Variation** to `TitleLabel`. It inherits everything a `Label` has and overrides just those items.

Here is the beginning and end of the finished `ui/theme.tres` for reference. It is a text file, readable like code. Normally the editor writes it for you.

```
[sub_resource type="StyleBoxFlat" id="btn_normal"]
bg_color = Color(0.23, 0.31, 0.54, 1)
corner_radius_top_left = 10
corner_radius_top_right = 10
corner_radius_bottom_right = 10
corner_radius_bottom_left = 10
content_margin_left = 20.0
content_margin_top = 10.0
content_margin_right = 20.0
content_margin_bottom = 10.0

[resource]
default_font_size = 22
Button/colors/font_color = Color(1, 1, 1, 1)
Button/styles/normal = SubResource("btn_normal")
TitleLabel/base_type = &"Label"
TitleLabel/colors/font_color = Color(1, 0.82, 0.4, 1)
TitleLabel/font_sizes/font_size = 64
```

(This listing is a trimmed extract; the full file is in the project.)

Overrides in code

Sometimes a script needs to change one control's look at run time, for example to turn a label red when health is low. Use the `add_theme_*_override` family:

```
# Fragment: inside a script that has a Label called health_label
func _flash_warning() -> void:
    health_label.add_theme_color_override("font_color", Color.RED)
    health_label.add_theme_font_size_override("font_size", 32)

func _clear_warning() -> void:
    health_label.remove_theme_color_override("font_color")
```

To *read* what the theme gives a control (handy in tests) use `get_theme_color("font_color")`, `get_theme_font_size("font_size")` or `get_theme_stylebox("normal")`.

14.6 Focus, keyboard and gamepad

Mouse users click. Keyboard and gamepad users move a highlight with the arrow keys or the stick and press accept. That highlight is the **focus**. Only one control has it at a time. Godot already knows how to move focus between neighbouring buttons using the built-in actions `ui_up`, `ui_down`, `ui_left`, `ui_right`, `ui_accept` and `ui_cancel`. You must do only two things.

1. Give something focus when the screen appears. Otherwise the first arrow-key press seems to do nothing, because no control is focused yet. The call is `grab_focus()`:

```
# Fragment: inside main_menu.gd
func _ready() -> void:
    play_button.grab_focus()
```

2. Make the focus visible. That is the `focus` style in the theme. In our theme it is a gold outline.

If a screen can be shown and hidden (our menus are, in one scene), call `grab_focus()` each time it is shown. The `visibility_changed` signal tells you:

```gdscript
# Fragment: inside any screen script
func _ready() -> void:
    visibility_changed.connect(_on_visibility_changed)

func _on_visibility_changed() -> void:
    if visible:
        play_button.grab_focus()
```

You can control the neighbours too. In the Inspector, **Focus > Neighbor Left/Right/Top/Bottom** let you point to a specific node. You rarely need them, because Godot guesses well from the layout.

`ui_cancel` is mapped to Escape (and gamepad B) by default. That is why our game uses it for "close menu" and "pause" without us defining new actions.

> **Why it works:** The UI actions are defined in the default Input Map. Open **Project > Project Settings > Input Map** and tick **Show Built-in Actions** to see them. Using them gives you free gamepad support.

14.7 Building Pip's Pantry

The project for this chapter is a small program with four screens: a main menu, an options page, a game screen with a HUD, and a pause menu. The "game" is minimal on purpose; buttons stand in for game events so that the focus is the interface. Chapter 18 will weave real gameplay into screens like these.

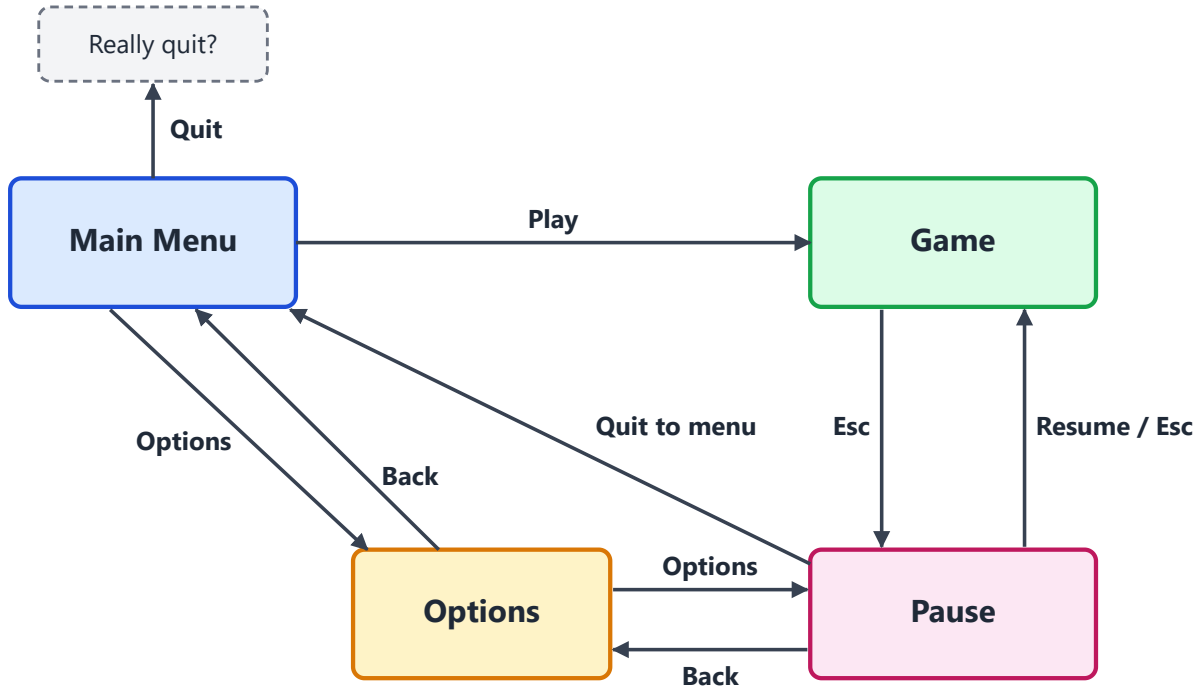


Figure 14.5: Four screens. The main menu opens Options or the game. Esc in the game opens Pause. Pause can open Options. Back always returns to where you came from.

The structure

Every screen is its own scene with a Full Rect `Control` as root. A `Main` scene instances them all and owns a small script that decides which one is visible. Keeping screens in separate scenes means that each is small, easy to edit and reusable.

```
Main (Control)                      script: main.gd    Theme: theme.tres   Process Mode: Always
  Background (ColorRect)            Full Rect, dark blue
  MainMenu (instance of main_menu.tscn)
  GameScreen (instance of game_screen.tscn)     Process Mode: Pausable, Visible: off
  OptionsMenu (instance of options_menu.tscn)   Visible: off
  PauseMenu (instance of pause_menu.tscn)       Visible: off
  QuitDialog (ConfirmationDialog)
```

For each instance node tick **Access as Unique Name** (right-click the node), so scripts can write `%MainMenu`. The percent sign is a shortcut that finds a node by its unique name anywhere in the same scene, no matter how deep it hides. It keeps working when you rearrange the tree. You saw `$Path/To/Node` before; `%Name` is the sturdier cousin.

Screen 1: the main menu

The tree is the recipe from 14.3. Give the title label **Theme Type Variation** `TitleLabel`. Mark the three buttons with unique names.

```
MainMenu (Control)                  Full Rect         script: main_menu.gd
  CenterContainer                   Full Rect
    VBoxContainer                   separation 16
      Title (Label)                 text "Pip's Pantry", variation TitleLabel, centered
      Subtitle (Label)              text "A very small game about snacks", centered
      Spacer (Control)              custom minimum size (0, 20)
      %PlayButton (Button)          minimum width 260
      %OptionsButton (Button)
      %QuitButton (Button)
```

```gdscript
# scripts/main_menu.gd  (attached to MainMenu)
extends Control
## The title screen. It only reports button presses; Main decides what happens.

signal play_pressed
signal options_pressed
signal quit_pressed

@onready var play_button: Button = %PlayButton
@onready var options_button: Button = %OptionsButton
@onready var quit_button: Button = %QuitButton


func _ready() -> void:
    play_button.pressed.connect(play_pressed.emit)
    options_button.pressed.connect(options_pressed.emit)
    quit_button.pressed.connect(quit_pressed.emit)
    visibility_changed.connect(_on_visibility_changed)
    _on_visibility_changed()


func _on_visibility_changed() -> void:
    # Keyboard and gamepad players need a button that already has focus.
    if visible:
        play_button.grab_focus()
```

There is a design decision here worth noticing. The menu does not know what "Play" does. It declares its own signals (`play_pressed`) and forwards each button press to one of them. `Main` listens. This is the same "signals up" rule as the platformer, and it means the menu can be tested or reused without the rest of the game. The line `play_button.pressed.connect(play_pressed.emit)` connects the button's signal straight to the `emit` function of our own signal. Chapter 15 explains why that works: a signal's `emit` is itself callable.

Screen 2: options

```
OptionsMenu (Control)              Full Rect         script: options_menu.gd
  CenterContainer                  Full Rect
    PanelContainer                 (themed card)
      MarginContainer              margins 32 / 24 / 32 / 24
        VBoxContainer              separation 20
          Heading (Label)          "Options", gold, size 36, centered
          GridContainer            columns 2, h separation 24, v separation 16
            Label "Volume"
            %VolumeSlider (HSlider)         min width 260, max 1, step 0.05, value 0.8
            Label "Fullscreen"
            %FullscreenCheck (CheckButton)  Size Flags Horizontal: Shrink Begin
            Label "Difficulty"
            %DifficultyOption (OptionButton)
            Label "Player name"
            %NameEdit (LineEdit)            placeholder "Type a name", max length 12
          %BackButton (Button)
```

```gdscript
# scripts/options_menu.gd  (attached to OptionsMenu)
extends Control
## The options screen. Volume changes the Master audio bus right away.

signal closed

const DIFFICULTIES: Array[String] = ["Easy", "Normal", "Hard"]

@onready var volume_slider: HSlider = %VolumeSlider
@onready var fullscreen_check: CheckButton = %FullscreenCheck
@onready var difficulty_option: OptionButton = %DifficultyOption
@onready var name_edit: LineEdit = %NameEdit
@onready var back_button: Button = %BackButton


func _ready() -> void:
    for difficulty in DIFFICULTIES:
        difficulty_option.add_item(difficulty)
    difficulty_option.select(1)

    volume_slider.value_changed.connect(_on_volume_changed)
    fullscreen_check.toggled.connect(_on_fullscreen_toggled)
    back_button.pressed.connect(closed.emit)
    visibility_changed.connect(_on_visibility_changed)
    _on_volume_changed(volume_slider.value)


func get_difficulty() -> String:
    return DIFFICULTIES[difficulty_option.selected]


func get_player_name() -> String:
    var text: String = name_edit.text.strip_edges()
    return text if text != "" else "Pip"


func _on_volume_changed(value: float) -> void:
    # The slider is 0..1, but the audio bus wants decibels.
    AudioServer.set_bus_volume_db(0, linear_to_db(value))
    AudioServer.set_bus_mute(0, value <= 0.0)
```

```
func _on_fullscreen_toggled(pressed: bool) -> void:
    if pressed:
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_FULLSCREEN)
    else:
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_WINDOWED)


func _on_visibility_changed() -> void:
    if visible:
        volume_slider.grab_focus()
```

Points to note:

- The difficulty items are filled from a constant array. Adding "Nightmare" means one new word in the array.
- `get_player_name()` returns "Pip" when the field is empty. Always decide what happens for empty input.
- `DisplayServer.window_set_mode(...)` is how a game goes fullscreen. In Chapter 17 you will save these settings to disk so they survive restarting.
- Nothing is saved yet. If you close the game the volume is back to 0.8. That is Chapter 17's job.

Screen 3: the game screen

This screen shows the three kinds of HUD element: a bar, a counter and a row of icons, plus an inventory grid and three buttons that stand in for game events.

```
GameScreen (Control)                Full Rect      script: game_screen.gd
  Margin (MarginContainer)          Full Rect, margins 24 / 20
    VBox (VBoxContainer)
      TopBar (HBoxContainer)        separation 32
        HealthBox (VBoxContainer)
          Label "Health"
          %HealthBar (ProgressBar)  min size (280, 24), Show Percentage off
        %Hearts (HBoxContainer)     separation 8
          Heart1, Heart2, Heart3 (ColorRect)   min size (28, 28), red
        Spacer (Control)            Size Flags Horizontal: Fill + Expand
        %ScoreLabel (Label)         gold, size 32
      Middle (CenterContainer)      Size Flags Vertical: Fill + Expand
        %StatusLabel (Label)        "Press Esc to pause"
      BottomBar (HBoxContainer)
        InventoryPanel (PanelContainer)
          Padding (MarginContainer)
            InventoryBox (VBoxContainer)
              Label "Basket (click an item to use it)"
              %Inventory (GridContainer)   columns 4, separations 8
        Spacer (Control)            Size Flags Horizontal: Fill + Expand
        DemoButtons (VBoxContainer) Size Flags Vertical: Shrink End
          %DamageButton, %HealButton, %ScoreButton (Button)
```



Figure 14.6: The HUD in place. The three bars of the layout are visible: top bar, centered status text and bottom bar with the basket.

The `Middle` container expands vertically, which is what pushes the bottom bar to the bottom: it takes all the leftover height.

The hearts are three red squares (`ColorRect`). To "lose" a heart, the script simply hides the last one. An `HBoxContainer` ignores invisible children, so the row closes up by itself. A real game would use a heart picture in a `TextureRect`.

```gdscript
# scripts/game_screen.gd  (attached to GameScreen)
extends Control
## The in-game screen: a health bar, lives, a score and a clickable basket.
## Nothing here is a real game; the buttons stand in for game events.

signal lives_ran_out

const MAX_HEALTH: int = 100
const START_ITEMS: Array[String] = ["Apple", "Cheese", "Apple", "Bread"]
const SLOT_COUNT: int = 8

var health: int = MAX_HEALTH
var lives: int = 3
var score: int = 0
var items: Array[String] = []

@onready var health_bar: ProgressBar = %HealthBar
@onready var hearts: HBoxContainer = %Hearts
@onready var score_label: Label = %ScoreLabel
@onready var status_label: Label = %StatusLabel
@onready var inventory: GridContainer = %Inventory
@onready var damage_button: Button = %DamageButton
@onready var heal_button: Button = %HealButton
@onready var score_button: Button = %ScoreButton


func _ready() -> void:
    damage_button.pressed.connect(take_damage.bind(30))
    heal_button.pressed.connect(heal.bind(20))
    score_button.pressed.connect(add_score.bind(10))
```

```
    # Build the eight basket slots with code.
    for i in SLOT_COUNT:
        var slot: Button = Button.new()
        slot.custom_minimum_size = Vector2(88, 56)
        slot.pressed.connect(_on_slot_pressed.bind(i))
        inventory.add_child(slot)
    new_game()


func new_game() -> void:
    health = MAX_HEALTH
    lives = 3
    score = 0
    items = START_ITEMS.duplicate()
    status_label.text = "Press Esc to pause"
    _refresh()


func take_damage(amount: int) -> void:
    health = maxi(health - amount, 0)
    if health == 0:
        lives -= 1
        if lives <= 0:
            lives_ran_out.emit()
            new_game()
            status_label.text = "Out of lives! Starting over."
            return
        status_label.text = "Ouch! A life is gone."
        health = MAX_HEALTH
    _refresh()


func heal(amount: int) -> void:
    health = mini(health + amount, MAX_HEALTH)
    _refresh()


func add_score(amount: int) -> void:
    score += amount
    _refresh()


func _on_slot_pressed(index: int) -> void:
    if index >= items.size():
        return
    var item: String = items[index]
    items.remove_at(index)
    if item == "Apple":
        heal(20)
        add_score(5)
    else:
        add_score(10)
    status_label.text = "You ate the %s." % item
    _refresh()


func _refresh() -> void:
    # Smoothly slide the bar instead of jumping.
    var tween: Tween = create_tween()
    tween.tween_property(health_bar, "value", float(health), 0.25)
    score_label.text = "Score: %d" % score
    for i in hearts.get_child_count():
        hearts.get_child(i).visible = i < lives
    for i in inventory.get_child_count():
        var slot: Button = inventory.get_child(i)
        if i < items.size():
            slot.text = items[i]
            slot.tooltip_text = "Click to eat the %s" % items[i]
            slot.disabled = false
        else:
            slot.text = "-"
```

```
            slot.tooltip_text = "Empty"
            slot.disabled = true
```

Some ideas in this script deserve attention.

Data first, display second. The script keeps the truth in plain variables (`health`, `lives`, `score`, `items`). The function `_refresh()` copies the truth to the display. Every change ends with `_refresh()`. This pattern, *one source of truth and one function that redraws*, prevents the classic bug where the number on screen and the real number disagree.

`bind()`. `take_damage.bind(30)` creates a new callable that calls `take_damage(30)`. The button's `pressed` signal has no arguments, but our function needs one, so `bind` supplies it. In the loop, `_on_slot_pressed.bind(i)` captures a different `i` for each slot, so each button knows which slot it is.

Slots made by code. A basket of eight identical buttons is better made by a loop than by dragging eight nodes. `Button.new()` creates a button, we set its properties, and `inventory.add_child(slot)` puts it in the grid. The grid takes care of the layout.

Health arithmetic. `maxi(health - amount, 0)` stops health going below zero; `mini(health + amount, MAX_HEALTH)` stops it going above the maximum. `maxi` and `mini` are the integer versions of "take the larger" and "take the smaller".

Tooltips. Setting `tooltip_text` costs nothing and Godot shows the bubble after the mouse rests on the button.

Screen 4: pause menu

Pausing a game freezes everything in the tree: movement, timers, animation, tweens. The pause menu itself must *not* freeze, or it could never respond. The solution is the node property **Process Mode**.

Process Mode	Meaning
Inherit	use the parent's mode (the default)
Pausable	runs only while the tree is *not* paused
When Paused	runs only while the tree *is* paused
Always	always runs
Disabled	never runs

The pause menu uses `When Paused`. The game screen uses `Pausable` so it freezes. The `Main` node uses `Always`, because it must be awake to unpause the game when Esc is pressed again.

```
PauseMenu (Control)               Full Rect   Process Mode: When Paused   Visible: off
  Dim (ColorRect)                 Full Rect, black, alpha 0.6
  CenterContainer                 Full Rect
    PanelContainer
      MarginContainer             40 / 28 / 40 / 32
        VBoxContainer             separation 14
          Heading (Label)         "Paused"
          %ResumeButton (Button)  min width 240
          %OptionsButton (Button)
          %MenuButton (Button)    "Quit to menu"
```

```gdscript
# scripts/pause_menu.gd  (attached to PauseMenu)
extends Control
## The pause overlay. Its process_mode is "When Paused", so it works while the tree is paused.
```

```
signal resume_pressed
signal options_pressed
signal menu_pressed

@onready var resume_button: Button = %ResumeButton
@onready var options_button: Button = %OptionsButton
@onready var menu_button: Button = %MenuButton


func _ready() -> void:
    resume_button.pressed.connect(resume_pressed.emit)
    options_button.pressed.connect(options_pressed.emit)
    menu_button.pressed.connect(menu_pressed.emit)
    visibility_changed.connect(_on_visibility_changed)


func _on_visibility_changed() -> void:
    if visible:
        resume_button.grab_focus()
```

The `Dim` rectangle is black with alpha (transparency) 0.6. It covers the screen, darkens the game behind the menu, and its default `mouse_filter` of `Stop` swallows clicks so nothing behind it can be clicked while paused. This is a standard modal overlay.

The screen manager

`Main` ties it together. It owns no appearance; it only decides what is visible and handles the Esc key.

```
# scripts/main.gd  (attached to Main)
extends Control
## The screen manager. It owns four screens and decides which one is visible.

@onready var main_menu: Control = %MainMenu
@onready var game_screen: Control = %GameScreen
@onready var options_menu: Control = %OptionsMenu
@onready var pause_menu: Control = %PauseMenu
@onready var quit_dialog: ConfirmationDialog = %QuitDialog

var options_opened_from_pause: bool = false


func _ready() -> void:
    main_menu.play_pressed.connect(start_game)
    main_menu.options_pressed.connect(open_options.bind(false))
    main_menu.quit_pressed.connect(quit_dialog.popup_centered)
    options_menu.closed.connect(close_options)
    pause_menu.resume_pressed.connect(resume)
    pause_menu.options_pressed.connect(open_options.bind(true))
    pause_menu.menu_pressed.connect(quit_to_menu)
    quit_dialog.confirmed.connect(get_tree().quit)
    show_menu()


func _unhandled_input(event: InputEvent) -> void:
    if not event.is_action_pressed("ui_cancel"):
        return
    if options_menu.visible:
        close_options()
    elif pause_menu.visible:
        resume()
    elif game_screen.visible:
        pause()
    else:
        return
    get_viewport().set_input_as_handled()


func show_menu() -> void:
```

```
    main_menu.show()
    game_screen.hide()
    options_menu.hide()
    pause_menu.hide()


func start_game() -> void:
    main_menu.hide()
    game_screen.new_game()
    game_screen.show()


func pause() -> void:
    get_tree().paused = true
    pause_menu.show()


func resume() -> void:
    get_tree().paused = false
    pause_menu.hide()


func quit_to_menu() -> void:
    get_tree().paused = false
    show_menu()


func open_options(from_pause: bool) -> void:
    options_opened_from_pause = from_pause
    main_menu.hide()
    pause_menu.hide()
    options_menu.show()


func close_options() -> void:
    options_menu.hide()
    if options_opened_from_pause:
        pause_menu.show()
    else:
        main_menu.show()
```

Walk through the flow of Figure 14.5.

- At start `show_menu()` shows only the menu. Its script grabs focus for the Play button.
- Pressing Play hides the menu, resets the game screen and shows it.
- `_unhandled_input` receives keys that no control wanted. Esc (`ui_cancel`) is interpreted according to what is visible: close Options, or resume if paused, or pause if playing. When it did something, `set_input_as_handled()` stops other nodes from seeing the same key press.
- Pausing sets `get_tree().paused = true` and shows the overlay; resuming undoes both.
- The Options screen can be opened from two places, so we store where it came from in `options_opened_from_pause`. `Back` uses that to go to the right place. This is the simplest correct use of a *remembered state variable*.
- `open_options.bind(false)` and `.bind(true)` give one function two behaviours.
- `quit_dialog.popup_centered` is connected without parentheses: we hand over the function and the signal calls it. The dialog is a built-in `ConfirmationDialog`: it shows its `dialog_text` with OK and Cancel buttons. When OK is pressed it emits `confirmed`, which we connect to `get_tree().quit`. Asking "Really quit?" before quitting is polite and avoids accidents.

> **Warning:** `ConfirmationDialog` is a `Window`, not a `Control`. It floats above everything and has its own layout rules. Do not put it in a container.
>
> **Note:** If a signal's callable is `get_tree().quit`, Godot evaluates `get_tree()` right now, in `_ready()`, then connects to that tree's `quit` function. That is fine because the tree exists at that point and stays the same for the whole game.

Testing the screens without clicking

As in Chapter 13, the project contains a headless test, `tests/test_ui.gd`. It builds the whole `Main` scene, presses buttons by calling `button.pressed.emit()`, sends a real Esc key event, and checks results: the Play button has focus, the title is centered on screen (a check of the layout), the theme font size is what we set, volume 0.5 is about -6 dB, and pausing really sets `paused`. Run it with:

```
godot --headless --path . --script res://tests/test_ui.gd
```

You should see all lines start with `PASS` and the last line read `FAILURES: 0`.

Project 14: Pip's Pantry interface

Goal: Build the four-screen interface described in 14.7 and a theme to go with it. Time: 60 to 90 minutes. The finished project is in `projects/ch14-ui/`.

1. **Create the project** `ch14-ui`. Set the viewport to 1152 by 648, stretch mode `canvas_items`, aspect `expand`.
2. **Make the theme** `ui/theme.tres` as in 14.5: default font size 22, a `Button` with four styles (`normal`, `hover`, `pressed`, `focus`), a `PanelContainer` panel style, a `ProgressBar` background and fill, and the `TitleLabel` type variation. Run any scene with a button and check that it looks right.
3. **Build** `main_menu.tscn` and its script. Press F6 to test: the Play button should be highlighted on start and Down-arrow should move through the buttons.
4. **Build** `options_menu.tscn` and its script. Test the volume slider (it needs sound playing to be audible, but the test in step 9 checks the value), the fullscreen switch, the drop-down and the name field.
5. **Build** `game_screen.tscn` and its script. Test the three demo buttons: Take damage should drain the bar, remove a heart at zero, and refill. Click the basket items: an Apple heals, others add points.
6. **Build** `pause_menu.tscn` and its script. Set its **Process Mode** to **When Paused** and **Visible** off.
7. **Build** `main.tscn`: root `Control` (Full Rect, Process Mode **Always**, theme applied), a `ColorRect` background, the four screen instances (with unique names) and a `ConfirmationDialog`. Set the game screen's Process Mode to **Pausable**. Attach `main.gd`.
8. **Set** `main.tscn` **as the main scene** and press F5. Walk through every path in Figure 14.5: menu to options and back; menu to game; Esc to pause; pause to options and back; pause to resume; pause to quit to menu; quit with confirmation.
9. **Run the headless test** and confirm `FAILURES: 0`.
10. **Resize the window** while the game runs and check every screen stays centered and readable.

Expected result: A themed four-screen interface. The title screen has a gold title and three rounded blue buttons with a gold focus ring. Options changes the Master volume live. The game screen shows a health bar that slides, three hearts, a score, a basket of food and three test buttons. Esc dims the game and shows a Pause panel; the game stops moving (the health bar will not slide while paused) until you resume.

Chapter summary

- Interface nodes inherit from `Control` and have a rectangle, layout rules and input handling.
- Anchors tie a control to fractions of its parent and offsets add pixel distances; Full Rect is the right choice for screen roots and backgrounds.
- Containers do the layout: VBox and HBox stack, Grid makes tables, Margin pads, Center centers, Panel draws a card. Expand-flagged spacers push items apart.
- Each input control has one main value and one main signal, such as `value_changed` for a slider and `toggled` for a check button.
- A `Theme` resource styles every control under a node; `StyleBoxFlat` draws rounded colored boxes without images, and type variations give a second look to one class.
- Call `grab_focus()` when a screen appears and give the theme a visible `focus` style so keyboard and gamepad players can use it.
- Process Mode controls what keeps running during a pause: the pause menu is `When Paused`, the game is `Pausable` and the screen manager is `Always`.
- Keep the truth in variables and redraw the interface from them in one function; let screens emit signals and let a manager decide what they mean.

Review questions

1. What are the three main differences between a `Control` and a `Node2D`?
2. A button has anchors left = 1, right = 1, top = 0, bottom = 0 with small offsets. Where on the screen does it stay when the window is resized?
3. True or false: you can freely move a child of a `VBoxContainer` by changing its position in the Inspector.
4. You want a score label at the right end of a bar that also holds a health bar on the left. What do you place between them, and which size flag does it need?
5. Multiple choice: Which control would you use to let the player choose "Easy / Normal / Hard" from a drop-down? (a) `LineEdit` (b) `OptionButton` (c) `ProgressBar` (d) `GridContainer`
6. Why does a volume slider usually call `linear_to_db()`?
7. List the order in which Godot looks for a theme item, strongest first.
8. What does `grab_focus()` do and why do menus call it when they appear?
9. Multiple choice: Which Process Mode should a pause menu have? (a) Pausable (b) Disabled (c) When Paused (d) Inherit from a pausable parent
10. In `game_screen.gd` why do all changes end by calling `_refresh()` instead of updating the label inside each function?

Exercises

1. **Practice (easy):** In the theme, change the button colors to a green palette and make the title 80 points. Notice that no scene or script changes.
2. **Practice (easy):** Add a "Credits" button to the main menu that opens a simple `PanelContainer` with three `Label`s and a Back button. Reuse the "remembered state" idea from `options_opened_from_pause`.
3. **Apply (medium):** Add a `ScrollContainer` around a long `VBoxContainer` of 30 labels ("Patch notes") on its own screen. Check that the scroll bar works with the mouse wheel and that the Up/Down keys work when a button inside has focus.
4. **Apply (medium):** Make the health bar change color when health is below 30: in `_refresh()`, use `health_bar.add_theme_stylebox_override("fill", ...)` with a new `StyleBoxFlat` colored red, and remove the override when health is higher.

5. **Challenge (hard):** Add a "Controls" screen where the player can see which key is bound to `ui_accept` and `ui_cancel`. Use `InputMap.action_get_events("ui_accept")` and `event.as_text()` to build a label for each event. (Remapping keys is a larger project, but this is its first step.)

Common mistakes

- **A control ignores its position or size.** Cause: it sits inside a container, which controls layout. Fix: use minimum size and size flags, or remove it from the container.
- **The whole menu hugs the top-left corner.** Cause: the root `Control` is not Full Rect, so its size is zero or tiny. Fix: set the layout preset of the root to **Full Rect**.
- **Keyboard users cannot do anything on a new screen.** Cause: no control has focus. Fix: call `grab_focus()` on the first button when the screen is shown.
- **The pause menu freezes together with the game.** Cause: it has Process Mode Inherit or Pausable. Fix: set it to **When Paused** (and the screen manager to **Always**).
- **Clicks go through a panel to buttons behind it.** Cause: the panel's `mouse_filter` is Ignore, or there is no full-screen blocker. Fix: add a full-rect `ColorRect` with mouse filter **Stop** behind the panel, as the `Dim` node does.
- **The numbers on screen are wrong after a change.** Cause: some function changed the variable but forgot to update the label. Fix: funnel all changes through one `_refresh()`.

Further reading

- Size and anchors: https://docs.godotengine.org/en/stable/tutorials/ui/size_and_anchors.html
- Containers: https://docs.godotengine.org/en/stable/tutorials/ui/gui_containers.html
- GUI skinning: https://docs.godotengine.org/en/stable/tutorials/ui/gui_skinning.html
- Using the theme editor: https://docs.godotengine.org/en/stable/tutorials/ui/gui_using_theme_editor.html
- Control class reference: https://docs.godotengine.org/en/stable/classes/class_control.html
- Pausing games: https://docs.godotengine.org/en/stable/tutorials/scripting/pausing_games.html

Chapter 15: Signals in Depth and Changing Scenes

Learning objectives

By the end of this chapter you will be able to:

- Explain the difference between calling a function and emitting a signal, and choose between them.
- Connect signals in code with `connect()`, including extra arguments with `bind()`, one-shot connections, and lambdas, and disconnect them again.
- Declare custom signals with typed parameters and emit them from your own scripts.
- Pause a function until something happens by using `await` with a signal.
- Change scenes with `change_scene_to_file()` and `change_scene_to_packed()`, and explain what happens to the old scene.
- Build a scene manager that swaps screens with a fade and passes data into and out of them.

Key terms

- **Signal**: a named announcement that a node can make ("I was pressed", "I died"). Other code can listen for it.
- **Emit**: to fire a signal so that every connected listener is called.
- **Connect**: to register a function so that Godot calls it whenever a signal is emitted.
- **Callable**: a value that represents a function (plus, optionally, the object it belongs to and extra arguments). Connect and bind work with callables.
- **Bind**: to attach extra arguments to a callable so they are added to every call.
- **Lambda**: a small unnamed function written directly where it is used.
- **One-shot connection**: a connection that removes itself after it has been called once.
- **Await**: a keyword that pauses a function until a signal is emitted, then continues.
- **Coroutine**: a function that contains `await` and can therefore pause and resume.
- **Scene change**: replacing the running scene with a different one.
- **Scene manager**: a script that owns the job of switching screens, so that screens do not need to know about each other.
- **Group**: a tag that nodes can carry; you can call a function on every node with that tag.

15.1 Calling versus announcing

Suppose a coin is picked up. Many things might care: the score should go up, a sound should play, a particle effect should burst, an achievement might unlock, a door might open. How does the coin make all of that happen?

The first idea is to let the coin do each job itself:

```
# Fragment: the "everything inside the coin" approach (not recommended)
func _on_body_entered(body: Node2D) -> void:
    get_node("/root/Main/HUD/ScoreLabel").text = "Score: 10"
    get_node("/root/Main/Sounds/Ding").play()
    get_node("/root/Main/Door").open()
    queue_free()
```

This works for one game on one day. It is also fragile. The coin now depends on exact node paths in a scene it knows nothing about. Rename `HUD` and the coin breaks. Put the coin into a different level and it breaks. Delete the sound and it crashes. The coin has become hard to reuse.

The second idea is to let the coin **announce** what happened and let others react:

```
# Fragment: the "announce" approach
signal collected

func _on_body_entered(body: Node2D) -> void:
    collected.emit()
    queue_free()
```

Now the coin has no idea who is listening. Zero, one or ten different nodes might be. The coin works in any scene. That is the entire value of signals, and it has a name in programming: the **observer pattern**.

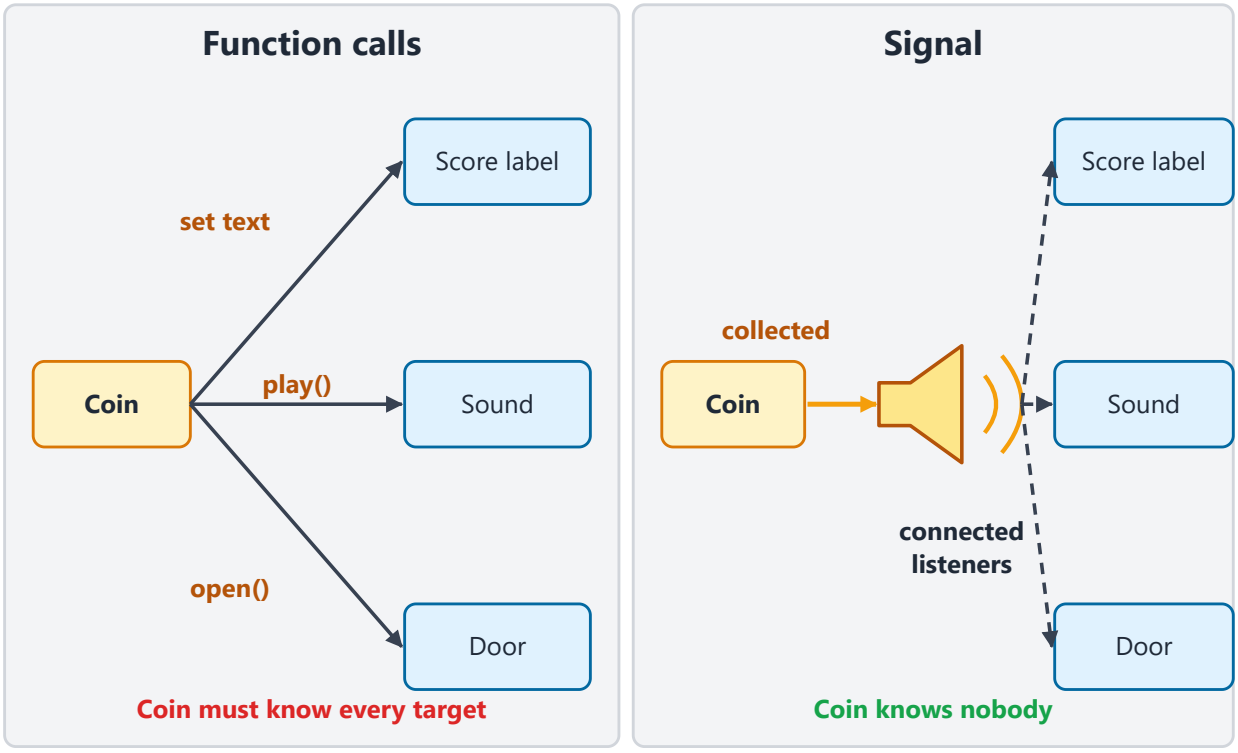


Figure 15.1: Left: the coin knows and calls each target. Right: the coin emits one signal, and the score, sound and door each subscribe on their own.

Think of a radio station. The station broadcasts; it does not know who is tuned in. Listeners choose to tune in. A function call is a phone call: you must know the number and the person must answer.

A rule of thumb

A good rule that keeps projects tidy is: **signals go up, calls go down.**

- A parent knows its children (it created them, it holds them), so it can **call** their functions: `door.open()`.
- A child should not know about its parent, so it **emits signals** and lets the parent decide what happens.

Chapter 13 followed this: the `Coin`, `Spike` and `Slime` emitted or did very little, the `Level` listened and passed events upward, and `Main` applied the rules. This chapter makes the idea precise.

Signals you already know

Every node class comes with built-in signals. You have used several:

Node	Signal	When it fires
`Button`	`pressed`	the button is clicked or activated
`Area2D`	`body_entered(body)`	a physics body overlaps the area
`Timer`	`timeout`	the countdown reaches zero
`AnimationPlayer`	`animation_finished(anim_name)`	an animation ends
`Tween`	`finished`	all tween steps are done
`HSlider`	`value_changed(value)`	the slider moves
`Node`	`tree_exited`	the node leaves the scene tree
`CanvasItem`	`visibility_changed`	shown or hidden

In the editor, select a node and open the **Signals** tab next to the Inspector. It lists every signal the node can emit, with its parameters. Double-click a signal to connect it. That is the same thing as writing `connect()` in code, which we now study.

15.2 Connecting signals in code

The editor connection dialog is fine for quick work. Code connections are better for anything made at run time, anything you want in version control as readable text, and for the reasons explained in 15.2.4. The general form in Godot 4 is:

```
emitter.signal_name.connect(callable)
```

`emitter` is the node, `signal_name` is the signal (a property-like object), and `callable` is the function to run. Notice there are **no parentheses** after the function name. We are handing over the function, not calling it.

```
# Fragment: inside _ready() of a script that has a Button child named PlayButton
$PlayButton.pressed.connect(_on_play_button_pressed)

func _on_play_button_pressed() -> void:
    print("Play was pressed")
```

By convention the receiving function is named `_on_<emitter>_<signal>`. The leading underscore says "this function is a private helper; nothing else should call it by hand".

15.2.1 Arguments

Many signals carry values. `body_entered` carries the body, `value_changed` carries the new number. The receiving function must accept the same arguments, in the same order:

```
# Fragment: a slider and its handler
$VolumeSlider.value_changed.connect(_on_volume_slider_value_changed)

func _on_volume_slider_value_changed(value: float) -> void:
    print("Volume is now ", value)
```

A function may declare **fewer** parameters than the signal sends only if you use a lambda or `unbind()`; ordinary connections require an exact match. If you do not need an argument, you still must accept it. Name it with a leading underscore (`_value`) to tell yourself and GDScript that you are ignoring it on purpose.

15.2.2 bind(): adding your own arguments

Suppose three buttons share one handler, and the handler needs to know which button was pressed. The `pressed` signal sends nothing, so we **bind** an extra argument:

```
# Fragment: three buttons, one function
$EasyButton.pressed.connect(_on_difficulty_chosen.bind("easy"))
$NormalButton.pressed.connect(_on_difficulty_chosen.bind("normal"))
$HardButton.pressed.connect(_on_difficulty_chosen.bind("hard"))

func _on_difficulty_chosen(level: String) -> void:
    print("You chose ", level)
```

`_on_difficulty_chosen.bind("easy")` makes a new callable that, when called, runs `_on_difficulty_chosen("easy")`. If the signal also sends arguments, they come first and bound arguments come after. The project's title screen uses exactly this to make "Normal" and "Quick" share one handler.

15.2.3 Lambdas: tiny functions in place

A **lambda** is a function without a name, written where you need it. It is good for short, one-off reactions:

```
# Fragment: print a message when the timer ends
$Timer.timeout.connect(func() -> void: print("Time is up"))
```

Lambdas can use variables from the surrounding function. Do not overuse them; a handler with a good name is easier to read, to search for and to disconnect. The rule of thumb is: a lambda for one short line, a named function for anything longer.

15.2.4 Flags: one-shot and deferred

`connect()` takes an optional second argument of flags. Two are worth knowing.

- `CONNECT_ONE_SHOT`: the connection removes itself after the first call. It is perfect for "do this the next time it happens, then forget it". The key room uses it for the door: finishing the room can happen only once.
- `CONNECT_DEFERRED`: the function is called at the end of the frame instead of immediately. This is a safe way to run code that must not happen in the middle of physics.

```
# Fragment
door.entered.connect(_on_door_entered, CONNECT_ONE_SHOT)
```

15.2.5 Checking and disconnecting

Connections live as long as both objects live. When either node is freed, Godot removes the connection automatically. Sometimes you need to cut a connection earlier:

```
# Fragment
if health_changed.is_connected(_on_health_changed):
    health_changed.disconnect(_on_health_changed)
```

`is_connected` asks whether a particular callable is already connected. Calling `connect()` twice with the same callable prints an error ("already connected"), so guard it with `is_connected` when a function could run more than once. Calling `disconnect()` on something that is not connected is also an error, hence the check.

> **Why it works:** A signal is basically a list of callables. `connect` adds one to the list, `disconnect` removes it, `emit` walks the list and calls each one with the same arguments. Nothing more mysterious than that.

15.2.6 The signal lab

Let us watch all of this at work. The project of this chapter contains `scenes/demo/signal_lab.tscn`, a single `Node` with the following script. Make a new project `ch15-signals-scenes`, add a `Node` and attach this script, then press **F6**.

```
# scripts/signal_lab.gd  (attached to a Node)
extends Node
## A tiny lab for every signal trick in the chapter. Open signal_lab.tscn and
## press F6 (Run Current Scene) to watch the Output panel.

signal health_changed(old_value: int, new_value: int)
signal died
signal greeted

var health: int = 100
var log_lines: Array[String] = []   # the headless test reads this


func say(line: String) -> void:
    log_lines.append(line)
    print(line)


func hurt(amount: int) -> void:
    var old_value: int = health
    health = maxi(health - amount, 0)
    health_changed.emit(old_value, health)
    if health == 0:
        died.emit()


func _ready() -> void:
    run_demo()


func run_demo() -> void:
    # 1. A normal connection to a named method.
    health_changed.connect(_on_health_changed)
    # 2. One-shot: disconnects itself after the first call.
    died.connect(_on_died, CONNECT_ONE_SHOT)
    # 3. bind(): add an extra argument at the end of the call.
    greeted.connect(_on_greeted.bind("Pip"))
    # 4. A lambda: a small unnamed function written in place.
    greeted.connect(func() -> void: say("lambda says hi"))

    greeted.emit()
    hurt(30)
    hurt(80)            # health reaches 0 -> died fires
    hurt(10)            # health is already 0, died fires again but nobody listens

    # 5. is_connected and disconnect.
    say("still connected? %s" % health_changed.is_connected(_on_health_changed))
    health_changed.disconnect(_on_health_changed)
    say("after disconnect? %s" % health_changed.is_connected(_on_health_changed))
    health = 100
    hurt(5)             # silent now
```

```
    # 6. await: pause this function until a signal fires.
    get_tree().create_timer(0.05).timeout.connect(greeted.emit)
    await greeted
    say("await finished")


func _on_health_changed(old_value: int, new_value: int) -> void:
    say("health %d -> %d" % [old_value, new_value])


func _on_died() -> void:
    say("died (one-shot)")


func _on_greeted(who: String) -> void:
    say("hello, " + who)
```

The Output panel shows:

```
hello, Pip
lambda says hi
health 100 -> 70
health 70 -> 0
died (one-shot)
health 0 -> 0
still connected? true
after disconnect? false
hello, Pip
lambda says hi
await finished
```

Follow the output against the code.

1. `greeted.emit()` calls every function connected to `greeted`, in the order they were connected: the bound handler, then the lambda.
2. `hurt(30)` sends `health_changed(100, 70)`. `hurt(80)` takes health to 0 and sends `(70, 0)`, and since health is 0 also emits `died`. The one-shot handler prints once.
3. `hurt(10)` sends `(0, 0)`. `died` fires again, but the one-shot connection is gone, so nothing prints. This is the proof that `CONNECT_ONE_SHOT` removed itself.
4. After `disconnect`, `hurt(5)` prints nothing.
5. The last lines show `await`, covered in 15.4. Nothing in this lab changes after the pause except a new message; the `greeted.emit` fires again because the timer calls it, which is why the greeting lines appear twice.

> **Try it:** Remove `CONNECT_ONE_SHOT` and run again. The line "died (one-shot)" now appears twice. Put it back.

Two details in the listing: the `signal health_changed(old_value: int, new_value: int)` line gives the parameters **types** (more in 15.3), and `get_tree().create_timer(0.05).timeout.connect(greeted.emit)` shows that a signal's `emit` function is itself a callable you can connect to another signal. We used that trick for the menu buttons in Chapter 14.

15.3 Custom signals

Your own scripts can declare their own signals. A custom signal is a promise: "when this thing happens to me, I will announce it". Declare it at the top of a script with the `signal` keyword:

```gdscript
signal died
signal health_changed(old_value: int, new_value: int)
signal item_picked_up(item_name: String)
```

Rules and habits:

- **Name it in the past tense.** `died`, `collected`, `door_opened`. The signal reports something that already happened. (Requests for future things use a name like `start_requested`, as the title screen does.)
- **Type the parameters.** `item_name: String` documents the data, and the editor shows it in autocomplete and in the connection dialog.
- **Send the facts, not the reaction.** The key says `picked_up("Red")`. It does not say `open_the_door`. Whoever listens decides what a pickup means.
- **Emit with `.emit(...)`.** In Godot 4 you write `item_picked_up.emit("Red")`. The old `emit_signal("name")` string version is Godot 3 style; avoid it.
- **Only the owner emits.** A node should emit its own signals, not another node's. That keeps the cause of an event easy to find.

The key room

The project of this chapter is a tiny top-down puzzle called **Key Room**. Pip walks around a room, collects keys, and when he has all of them the door unlocks. He walks into the door to finish. There are three scripts that matter, and they only talk through signals.

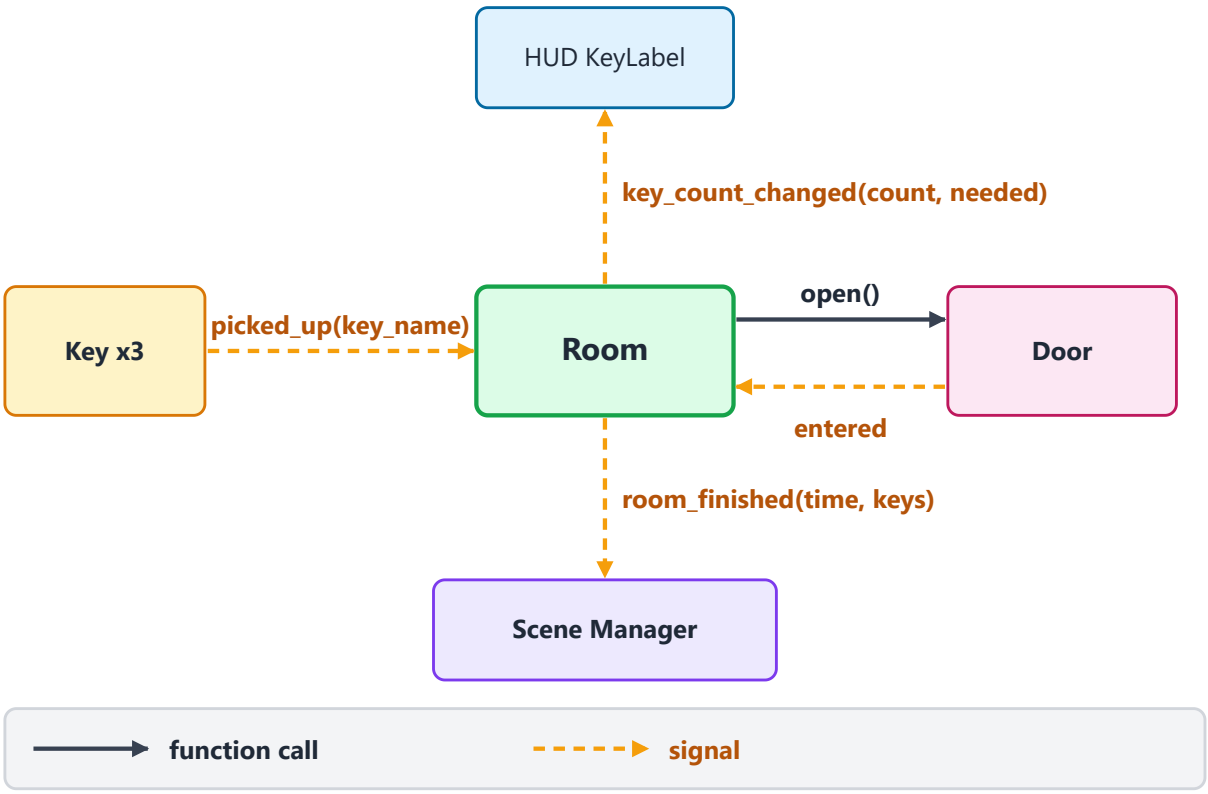


Figure 15.2: Each Key emits `picked_up`. The Room counts keys, emits `key_count_changed` for the HUD, and calls `door.open()` when done. The Door emits `entered`. The Room then emits `room_finished`, which the scene manager hears.

Start with the smallest piece, the key:

```
Key (Area2D)                        layer: none   mask: 2 (player)    script: key.gd
  CollisionShape2D                  CircleShape2D radius 16, position (5, 0)
```

```gdscript
# scripts/key.gd  (attached to Key)
extends Area2D
## A key lying on the floor. It announces its pickup with a custom signal.

signal picked_up(key_name: String)

@export var key_name: String = "Gold"
@export var key_color: Color = Color("ffd166")

var _taken: bool = false


func _ready() -> void:
    body_entered.connect(_on_body_entered)


func _on_body_entered(body: Node2D) -> void:
    if _taken or not body is Pip:
        return
    _taken = true
    picked_up.emit(key_name)
    queue_free()


func _draw() -> void:
    draw_arc(Vector2(-8, 0), 8.0, 0.0, TAU, 20, key_color, 4.0)
    draw_line(Vector2(0, 0), Vector2(18, 0), key_color, 4.0)
    draw_line(Vector2(12, 0), Vector2(12, 7), key_color, 4.0)
    draw_line(Vector2(18, 0), Vector2(18, 7), key_color, 4.0)
```

`@export` variables show up in the Inspector, so each key in the room can be given its own name and color without a new script. The `body is Pip` check refers to the hero's `class_name`.

Next the door, which both receives a call (`open()`) and announces events (`opened`, `entered`):

```
Door (StaticBody2D)                 layer: 1 (world)   mask: none    script: door.gd
  CollisionShape2D                  RectangleShape2D 96 x 32
  Trigger (Area2D)                  layer: none   mask: 2 (player)
    CollisionShape2D                RectangleShape2D 120 x 60, position (0, 20)
```

```gdscript
# scripts/door.gd  (attached to Door)
extends StaticBody2D
## A locked door. Call open() and it unlocks; walking into the trigger zone
## while it is open emits "entered".

signal opened
signal entered

var is_open: bool = false

@onready var shape: CollisionShape2D = $CollisionShape2D
@onready var trigger: Area2D = $Trigger


func _ready() -> void:
    trigger.body_entered.connect(_on_trigger_body_entered)


func open() -> void:
    if is_open:
        return
    is_open = true
    shape.set_deferred("disabled", true)
```

```
    queue_redraw()
    opened.emit()


func _on_trigger_body_entered(body: Node2D) -> void:
    if is_open and body is Pip:
        entered.emit()


func _draw() -> void:
    var color: Color = Color("3a7d44") if is_open else Color("8b5a2b")
    draw_rect(Rect2(-48, -16, 96, 32), color)
    if not is_open:
        draw_circle(Vector2.ZERO, 5.0, Color("ffd166"))
```

The door shows both directions of communication in one script. `open()` is a **function the parent calls** (calls go down). `opened` and `entered` are **signals the door emits** (signals go up). The door never mentions the room, the keys or the score.

The player is the simplest script in the chapter, because here Pip simply moves in four directions:

```
# scripts/pip.gd  (attached to Pip, a CharacterBody2D with a CircleShape2D radius 13, layer 2, mask 1)
class_name Pip
extends CharacterBody2D
## Pip, seen from above. Moves in all four directions with the arrow keys.

@export var speed: float = 220.0


func _physics_process(_delta: float) -> void:
    var direction: Vector2 = Input.get_vector("ui_left", "ui_right", "ui_up", "ui_down")
    velocity = direction * speed
    move_and_slide()


func _draw() -> void:
    draw_circle(Vector2.ZERO, 14.0, Color("ffd166"))
    draw_arc(Vector2.ZERO, 14.0, 0.0, TAU, 32, Color("b5832a"), 2.0)
    draw_circle(Vector2(-5, -3), 3.0, Color("222222"))
    draw_circle(Vector2(5, -3), 3.0, Color("222222"))
```

`Input.get_vector` combines four actions into one `Vector2`, already limited to length 1 so diagonal movement is not faster. We use the built-in `ui_*` actions, so the Input Map needs no setup.

Now the room itself. It is the middle manager: it listens to the keys and the door and it announces events to the HUD and to the scene manager.

```
Room (Node2D)                       script: room.gd
  Walls (StaticBody2D)              layer: 1
    Top, Bottom, Left, Right (CollisionShape2D)   960x32 and 32x540 rectangles at the four edges
  Keys (Node2D)
    RedKey, BlueKey, GreenKey (instances of key.tscn, different positions, names and colors)
  Door (instance of door.tscn)      position (480, 48)
  Pip (instance of pip.tscn)        position (480, 440)
  HUD (CanvasLayer)
    KeyLabel (Label)                script: hud_label.gd
```

```
# scripts/room.gd  (attached to Room)
extends Node2D
## The key room. Collect the keys, the door unlocks, walk into it to finish.

signal key_count_changed(count: int, needed: int)
signal room_finished(time_taken: float, keys: int)
```

```gdscript
var keys_needed: int = 3
var keys_collected: int = 0
var elapsed: float = 0.0
var finished: bool = false

@onready var keys_node: Node2D = $Keys
@onready var door: StaticBody2D = $Door
@onready var hud_label: Label = $HUD/KeyLabel


func setup(data: Dictionary) -> void:
    keys_needed = data.get("keys_needed", 3)


func _ready() -> void:
    # Quick mode: remove the extra keys before we start listening to them.
    var all_keys: Array[Node] = keys_node.get_children()
    for i in all_keys.size():
        if i >= keys_needed:
            all_keys[i].free()
        else:
            all_keys[i].picked_up.connect(_on_key_picked_up)

    door.entered.connect(_on_door_entered, CONNECT_ONE_SHOT)
    key_count_changed.connect(hud_label.show_keys)
    key_count_changed.emit(0, keys_needed)


func _process(delta: float) -> void:
    if not finished:
        elapsed += delta
        hud_label.show_time(elapsed)


func _on_key_picked_up(_key_name: String) -> void:
    keys_collected += 1
    key_count_changed.emit(keys_collected, keys_needed)
    if keys_collected >= keys_needed:
        door.open()


func _on_door_entered() -> void:
    finished = true
    room_finished.emit(elapsed, keys_collected)


func _draw() -> void:
    draw_rect(Rect2(0, 0, 960, 540), Color("3d405b"))
    draw_rect(Rect2(0, 0, 960, 540), Color("1b1d2e"), false, 64.0)
```

And the tiny HUD label that only displays what it is told:

```gdscript
# scripts/hud_label.gd  (attached to KeyLabel, a Label)
extends Label
## The heads-up text. It does not know about keys or doors; it only shows what it is told.

var _keys_text: String = ""
var _time_text: String = ""


func show_keys(count: int, needed: int) -> void:
    _keys_text = "Keys: %d / %d" % [count, needed]
    _refresh()


func show_time(seconds: float) -> void:
    _time_text = "Time: %.1f" % seconds
    _refresh()
```

```
func _refresh() -> void:
    text = _keys_text + "    " + _time_text
```

Study `_ready` in the room, because it uses nearly everything in this section.

- `all_keys[i].picked_up.connect(_on_key_picked_up)` connects a signal from a node we found in a loop.
- The `door.entered` connection is one-shot.
- `key_count_changed.connect(hud_label.show_keys)` connects **our own custom signal to a method of another node**. The label's function takes `(count, needed)` which matches the signal. The room does not reach into the label to change text; it announces the count, and the label updates itself.
- `key_count_changed.emit(0, keys_needed)` fires once at the start so the HUD shows "Keys: 0 / 3" from the beginning.
- `_on_key_picked_up(_key_name: String)` receives the key's name but does not need it (hence the leading underscore). We still have to accept it, because the signal sends it.

Look also at the beginning of `_ready()`: in "Quick" mode the room frees the surplus keys. We use `free()` rather than `queue_free()` here, because the keys have not been connected or used yet and we need them gone before we count and connect the rest. In general prefer `queue_free()`, which waits for a safe moment; use `free()` only when you are sure nothing else is using the node right now.

> **Warning:** Do not create connections in a function that runs more than once, such as `_process`, without checking `is_connected`. You would add the same callable again every frame.



Figure 15.3: The key room: Pip in the middle, one key left, the brown locked door at the top and the HUD "Keys: 2 / 3".

15.4 Waiting for signals with await

Some things take time: a one-second pause, an animation, a button press, a network answer. Game code often reads like a story: "wait one second, then show the message, then wait until the player presses OK, then continue". Writing that with only connected functions splits the story into many small functions. The keyword `await` keeps the story in one place.

```gdscript
# Fragment: show a message for two seconds, then clear it
func show_message(text: String) -> void:
    message_label.text = text
    await get_tree().create_timer(2.0).timeout
    message_label.text = ""
```

`get_tree().create_timer(2.0)` makes a one-off timer. It has a `timeout` signal. `await` followed by a signal means: "stop this function here; resume when that signal is emitted". The rest of the game keeps running. A function that contains `await` is called a **coroutine**.

You used this in Chapter 13: `await get_tree().create_timer(1.0).timeout` before reloading a level after a death.

Other things you can await:

```gdscript
# Fragment: the same idea with other signals
await animation_player.animation_finished      # an animation ends
await $StartButton.pressed                     # the player clicks
await tween.finished                           # a tween ends
var chosen: int = await dialog.option_selected # a custom signal that carries a value
```

When a signal carries one value, `await` returns it. With several values it returns an array of them.

Rules for await

1. **The caller continues immediately.** Calling a coroutine does not block the caller. If you want the caller to wait too, await the call: `await show_message("Hi")`. The scene manager in 15.6 does this.
2. **Anything may happen while you wait.** The node might be freed, the scene might change, the player might do something else. After an `await`, assume the world has moved on. If a node can disappear, check `is_instance_valid(node)` after the await.
3. **A signal that never comes means a function that never continues.** There is no error; the function just sleeps forever. If something seems to "stop working", check that the awaited signal is really emitted.
4. **Do not use `await` in `_process`.** It would start a new waiting copy every frame.

> **Why it works:** `await` does not freeze the game. Godot remembers where the function stopped, returns to the game loop, and later resumes the function from that exact line when the signal arrives.

15.5 Signals across the tree: groups and buses

Sometimes the listener is not close to the emitter. A coin deep inside a level wants to tell the HUD, which is a sibling of the level's parent. Three tools exist, in order of preference.

1. Pass events up one level at a time

This is what the platformer and the key room do. The coin tells the level, the level tells the main scene. It needs a few lines of forwarding code per level of the tree, but every link is visible, searchable and easy to understand. This is the best choice for most projects.

2. Groups

A **group** is a tag. Add the tag `enemies` to every slime (**Groups** tab next to the Inspector, or `add_to_group("enemies")` in code). Then anyone can talk to the whole group:

```gdscript
# Fragment: freeze every enemy, wherever it lives in the tree
get_tree().call_group("enemies", "set_physics_process", false)

# Fragment: count living enemies
var count: int = get_tree().get_nodes_in_group("enemies").size()
```

Groups are great for broadcasting a command *down* to many nodes ("all enemies, stop") or for finding nodes of a kind. They are not a replacement for signals, because groups are about commanding, not announcing.

3. A signal bus

If many distant nodes need to hear about the same events, you can create one shared object that holds the signals and nothing else, an **event bus**. Both emitters and listeners connect to the bus, so they do not need to know each other. The usual way to make a shared object that every script can reach is an **autoload** (also called a singleton). You will build one in Chapter 16 and use it in Chapter 18. The shape looks like this (a preview; do not use it yet):

```gdscript
# Preview, for Chapter 16: events.gd registered as an autoload named "Events"
extends Node

signal coin_collected
signal player_died
```

```gdscript
# Preview: the coin emits through the bus, the HUD listens to the bus
Events.coin_collected.emit()
Events.coin_collected.connect(_on_coin_collected)
```

A bus is convenient but has a cost: events come from nowhere. A reader of the HUD script cannot tell which node emits `coin_collected`. Use a bus for a handful of truly global events, not as the default.

15.6 Changing scenes

A game is made of screens: title, game, game over, options. Each is a scene. Moving between them is called a **scene change**. Godot gives you a built-in way and lets you build your own.

The built-in way

```gdscript
# Fragment: inside any script, for example a button handler
get_tree().change_scene_to_file("res://scenes/level_2.tscn")
```

`change_scene_to_file(path)` loads the scene file at the path, removes the **current scene** (the one Godot is running as the main scene) from the tree, frees it, and puts the new one in its place. The switch happens at the end of the frame, so the code that called it runs to the end first. There are two close relatives:

```gdscript
# Fragment
get_tree().change_scene_to_packed(my_packed_scene)   # you already have a PackedScene (e.g. from preload)
get_tree().reload_current_scene()                    # restart the current scene from scratch
```

`change_scene_to_packed` is useful when you already preloaded a scene, because it avoids loading from disk again. The function `reload_current_scene()` is the one-line "Try Again" button for a small game.

The project contains a tiny demonstration. `scenes/demo/demo_a.tscn` and `demo_b.tscn` use the same script, `demo_scene.gd`, which is attached to a `Control` that has a heading label and one button:

```gdscript
# scripts/demo_scene.gd  (attached to the root Control of demo_a.tscn and demo_b.tscn)
extends Control
## Used by demo_a.tscn and demo_b.tscn: one button that changes scene with the built-in method.

@export_file("*.tscn") var next_scene: String = ""
@export var heading: String = "Demo"

@onready var heading_label: Label = %HeadingLabel
@onready var go_button: Button = %GoButton


func _ready() -> void:
    heading_label.text = heading
    go_button.pressed.connect(_on_go_pressed)


func _on_go_pressed() -> void:
    get_tree().change_scene_to_file(next_scene)
```

`@export_file("*.tscn")` makes the Inspector show a file picker that only lists scene files. In `demo_a.tscn` the `next_scene` is `demo_b.tscn`, and in `demo_b.tscn` it points back to `main.tscn`. Two scenes, one script, no code duplication. Run `demo_a.tscn` with F6 and click through.

What happens to the old scene

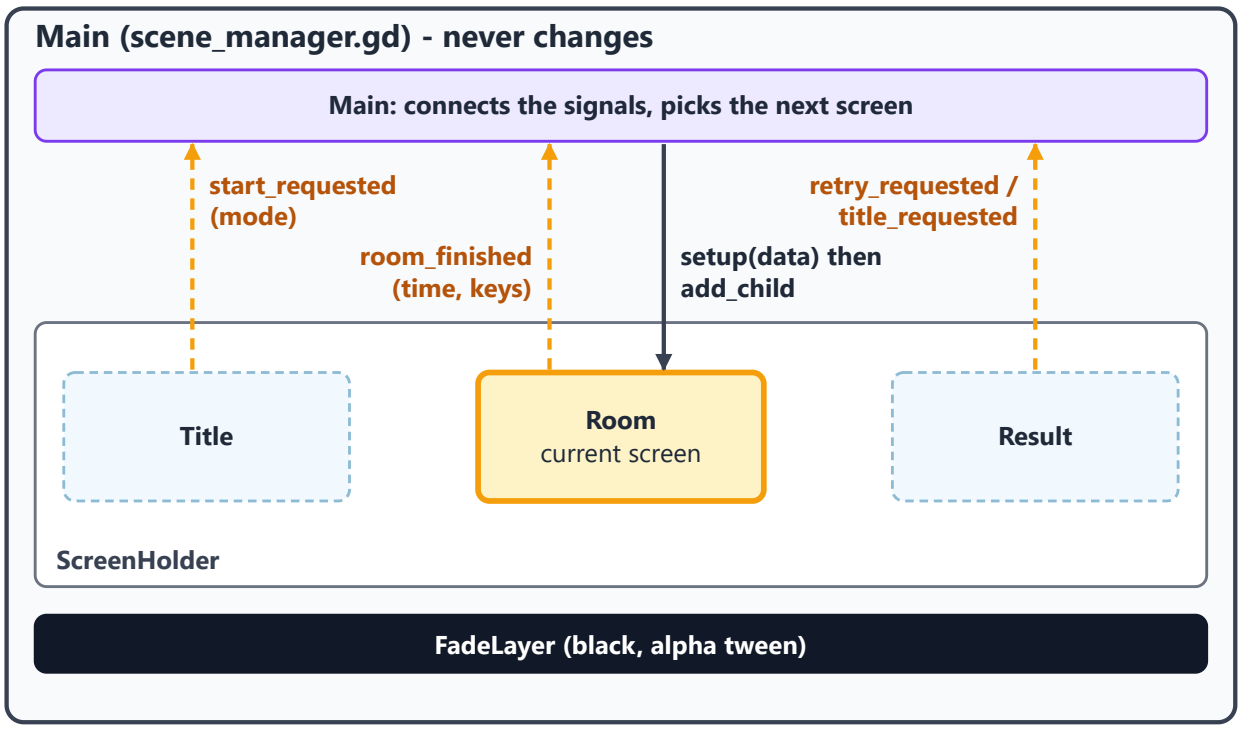
This is the part beginners trip over. The old scene is **freed**. All its variables, positions, unfinished timers and connections are gone. If you leave level 1 and come back, it starts new. Anything that should survive a scene change (score, lives, which level you reached) must live **somewhere that is not inside the scene**. There are three places:

1. **In the arguments**: you pass the data to the new scene when you create it (15.7).
2. **In a parent that does not change**: for example a `Main` node that stays and swaps only its child.
3. **In an autoload**: a global object that lives for the whole game (Chapter 16).

The built-in `change_scene_*` replaces the whole current scene, so it cannot pass data and it does not allow a fade between screens without extra tricks. For a game with more than a couple of screens a manager of your own is cleaner.

A scene manager

The idea: have one scene that **never changes** (the main scene), with an empty holder node inside it. The manager creates screens, adds them to the holder and removes the old one. Because the manager survives, it can keep data and run a fade.



dashed = signal from a screen up to Main solid = call from Main down to a screen

Figure 15.4: Main stays loaded the whole time. Screens are added to ScreenHolder one at a time. Screens report with signals; the manager decides what to load next and what data to hand over.

The tree:

```
Main (Node)                       script: scene_manager.gd
  ScreenHolder (Node)             empty at start; the current screen is its only child
  FadeLayer (CanvasLayer)         Layer: 10
    Fade (ColorRect)              Full Rect, black with alpha 0, Mouse Filter: Ignore
```

The `Fade` rectangle covers the screen. Its alpha is zero (see-through) most of the time. We tween the alpha to 1 to fade to black and back to 0 to fade in. **Mouse Filter: Ignore** lets clicks pass through it, so an invisible rectangle never blocks the buttons.

Here is the complete manager.

```gdscript
# scripts/scene_manager.gd  (attached to Main)
extends Node
## Swaps whole screens (title, room, result) inside one permanent scene,
## fading to black between them, and passes data along.

const TITLE_SCENE: PackedScene = preload("res://scenes/title.tscn")
const ROOM_SCENE: PackedScene = preload("res://scenes/room.tscn")
const RESULT_SCENE: PackedScene = preload("res://scenes/result.tscn")

const FADE_TIME: float = 0.2

var current: Node = null
var best_time: float = -1.0   # -1 means "no finished run yet"
var _busy: bool = false

@onready var holder: Node = $ScreenHolder
@onready var fade: ColorRect = $FadeLayer/Fade


func _ready() -> void:
```

```
    show_title()


func show_title() -> void:
    if _busy:
        return
    var screen: Node = await _swap(TITLE_SCENE, {"best_time": best_time})
    screen.start_requested.connect(show_room)
    screen.demo_requested.connect(_on_demo_requested)


func show_room(mode: String) -> void:
    if _busy:
        return
    var keys_needed: int = 3 if mode == "normal" else 1
    var screen: Node = await _swap(ROOM_SCENE, {"keys_needed": keys_needed})
    screen.room_finished.connect(show_result)


func show_result(time_taken: float, keys: int) -> void:
    if _busy:
        return
    var is_record: bool = best_time < 0.0 or time_taken < best_time
    if is_record:
        best_time = time_taken
    var screen: Node = await _swap(RESULT_SCENE, {
        "time": time_taken, "keys": keys, "is_record": is_record})
    screen.retry_requested.connect(show_room.bind("normal"))
    screen.title_requested.connect(show_title)


func _on_demo_requested() -> void:
    # The simple, built-in way to change scenes. It replaces this whole Main scene.
    get_tree().change_scene_to_file("res://scenes/demo/demo_a.tscn")


func _swap(scene: PackedScene, data: Dictionary) -> Node:
    _busy = true
    if current != null:
        var fade_out: Tween = create_tween()
        fade_out.tween_property(fade, "color:a", 1.0, FADE_TIME)
        await fade_out.finished
        current.queue_free()
        current = null
    var next: Node = scene.instantiate()
    next.setup(data)               # hand over data BEFORE the node enters the tree
    holder.add_child(next)
    current = next
    var fade_in: Tween = create_tween()
    fade_in.tween_property(fade, "color:a", 0.0, FADE_TIME)
    _busy = false
    return next
```

Take it in four steps.

The three `show_*` functions are the manager's menu of destinations. Each prepares the data for one screen, calls `_swap`, and then **connects that screen's signals**. Notice `await _swap(...)`: `_swap` is a coroutine (it awaits the fade-out), so the caller awaits it as well and receives the new screen when it is ready.

`_swap` does the real work, in this order:

1. Mark the manager busy.
2. If a screen exists, fade to black, wait for the tween to finish (`await fade_out.finished`), and free the old screen.
3. Instantiate the new screen and call its `setup(data)` function.

4. Add it to the holder, and start fading in. We deliberately do **not** await the fade-in, so `_swap` returns immediately and the caller connects signals in the same frame, before the player can click anything.

The `_busy` guard. While a fade runs, the old screen is still on the screen and its buttons still work. A player who double-clicks would trigger two swaps at once and end up with two screens. The first lines of every `show_*` function return early when `_busy` is true. This pattern, a boolean that blocks re-entry while something is in progress, is common wherever an action takes time.

The demo button. `_on_demo_requested` shows the built-in alternative: `change_scene_to_file` replaces `Main` itself, manager and all. Click "Scene-change demo", then through scenes A and B; "Back to the game" loads `main.tscn` fresh and the best time is forgotten. That forgotten time is a live demonstration of what happens to data in a replaced scene.

> **Note:** Because `_swap` calls `next.setup(data)`, every screen scene must have a `setup` function (even an empty one). This is a simple convention. If the function were missing the game would stop with an error at the first swap, which is the helpful direction to fail: loudly and early.

Screens: the title

The title screen announces the player's choice and displays the best time it was given.

```
Title (Control)                    Full Rect    script: title.gd
  Background (ColorRect)           Full Rect
  CenterContainer                  Full Rect
    VBoxContainer                  separation 14
      Heading (Label)              "Key Room"
      %BestLabel (Label)
      %NormalButton (Button)       "Normal (3 keys)"
      %QuickButton (Button)        "Quick (1 key)"
      %DemoButton (Button)         "Scene-change demo"
```

```gdscript
# scripts/title.gd  (attached to Title)
extends Control
## The title screen. It announces what the player chose; it does not change scenes itself.

signal start_requested(mode: String)
signal demo_requested

var best_time: float = -1.0

@onready var normal_button: Button = %NormalButton
@onready var quick_button: Button = %QuickButton
@onready var demo_button: Button = %DemoButton
@onready var best_label: Label = %BestLabel


func setup(data: Dictionary) -> void:
    best_time = data.get("best_time", -1.0)


func _ready() -> void:
    # bind() adds an extra argument to the call, so one function serves two buttons.
    normal_button.pressed.connect(_on_start_pressed.bind("normal"))
    quick_button.pressed.connect(_on_start_pressed.bind("quick"))
    demo_button.pressed.connect(demo_requested.emit)
    if best_time >= 0.0:
        best_label.text = "Best time: %.1f s" % best_time
    else:
        best_label.text = "No best time yet"
    normal_button.grab_focus()
```

```
func _on_start_pressed(mode: String) -> void:
    start_requested.emit(mode)
```

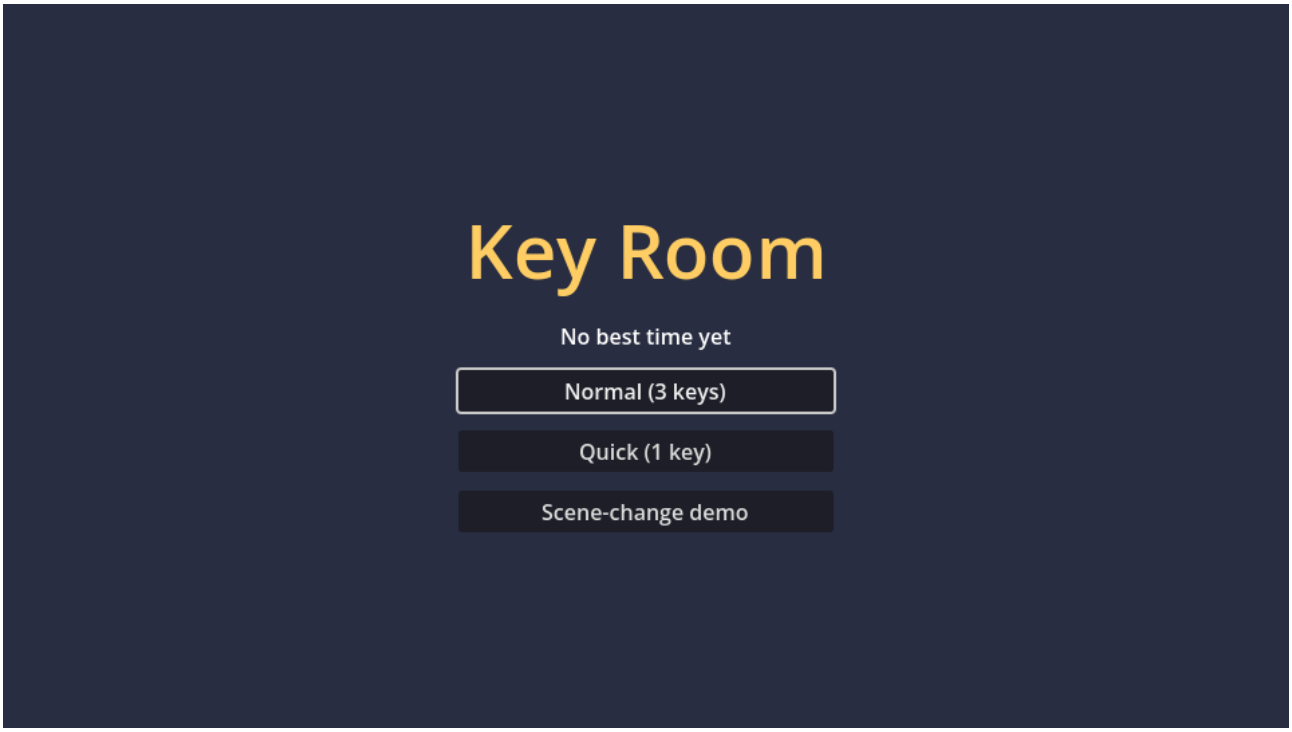


Figure 15.5: The title screen on a first start. The line under the heading comes from the scene manager, which passes it in through `setup()`; after a finished run it shows the best time instead of "No best time yet".

15.7 Passing data between scenes

Look at how information travels in the key room:

- **Into a screen:** the manager builds a `Dictionary` and calls `setup(data)` *before* adding the screen to the tree. A dictionary is flexible: add a key and a new screen can read it, without changing every function signature. Use `data.get("key", default)` so a missing key gives a safe default instead of an error.
- **Out of a screen:** the screen emits a signal with the results (`room_finished(time, keys)`). The manager is the listener.

This is a clean two-way street, and it needs no global variables.

The timing trap

There is one trap that catches almost everybody. Compare these two orders:

```
# Order A (what we do): data first, then add to the tree
var screen = RESULT_SCENE.instantiate()
screen.setup(data)
holder.add_child(screen)       # _ready() runs now

# Order B (what looks natural, but is wrong for @onready)
var screen = RESULT_SCENE.instantiate()
holder.add_child(screen)       # _ready() runs now
screen.setup(data)
```

`instantiate()` creates the nodes but does not run `_ready()`. That happens when the node enters the tree. The `@onready` variables are assigned at that moment too. So if `setup()` runs **before** `add_child`, the `@onready` variables are still empty, and `setup()` must not use them (it would be a null error). The right habit is shown in `result.gd`: `setup()` only **stores** the data in plain variables, and `_ready()`, which runs later with everything ready, **displays** it.

```
Result (Control)                    Full Rect    script: result.gd
  Background (ColorRect)            Full Rect
  CenterContainer
    VBoxContainer
      Heading (Label)               "Door unlocked!"
      %SummaryLabel (Label)
      %RecordLabel (Label)          green
      %RetryButton (Button)         "Play again"
      %TitleButton (Button)         "Back to title"
```

```gdscript
# scripts/result.gd  (attached to Result)
extends Control
## The result screen. Data arrives through setup(), BEFORE the node is in the tree,
## so we store it and show it later in _ready().

signal retry_requested
signal title_requested

var time_taken: float = 0.0
var keys: int = 0
var is_record: bool = false

@onready var summary_label: Label = %SummaryLabel
@onready var record_label: Label = %RecordLabel
@onready var retry_button: Button = %RetryButton
@onready var title_button: Button = %TitleButton


func setup(data: Dictionary) -> void:
    time_taken = data.get("time", 0.0)
    keys = data.get("keys", 0)
    is_record = data.get("is_record", false)


func _ready() -> void:
    summary_label.text = "You found %d key(s) in %.1f seconds." % [keys, time_taken]
    record_label.text = "New best time!" if is_record else ""
    retry_button.pressed.connect(retry_requested.emit)
    title_button.pressed.connect(title_requested.emit)
    retry_button.grab_focus()
```

The room uses the same convention: its `setup` stores `keys_needed`, and `_ready` uses it to decide which keys to keep.

> **Why it works:** All three screens answer to the same two-function protocol, `setup(data)` and "emit signals when done". The manager does not need to know anything else about them. This is a very small example of an *interface*: agreement about how to talk without knowing what is inside.

A map of the whole flow

Put it all together and the data flows like this:

1. Manager starts, `show_title()` passes the `best_time`.
2. The player presses "Quick". The title emits `start_requested("quick")`.

3. The manager's `show_room("quick")` passes `keys_needed = 1` into the room.
4. The player collects the key; the key emits `picked_up`; the room counts it, emits `key_count_changed` (the HUD updates) and opens the door.
5. The player walks into the door; the door emits `entered`; the room emits `room_finished(time, keys)`.
6. The manager's `show_result` compares the time with the record, updates `best_time` and passes the results to the result screen.
7. "Back to title" emits `title_requested`; the manager loads the title again, now with a best time to display.

Seven steps, five signals, no node ever touched another node's variables.

Project 15: Key Room

Goal: Build the three-screen Key Room game with a scene manager, custom signals and data passing. Time: 60 to 90 minutes. The finished project is in `projects/ch15-signals-scenes/`.

File list (all in `res://`):

```
project.godot, icon.svg
scenes/main.tscn, title.tscn, room.tscn, result.tscn, pip.tscn, key.tscn, door.tscn
scenes/demo/demo_a.tscn, demo_b.tscn, signal_lab.tscn
scripts/scene_manager.gd, title.gd, room.gd, result.gd, hud_label.gd,
        pip.gd, key.gd, door.gd, demo_scene.gd, signal_lab.gd
tests/test_flow.gd
```

1. **Create the project** `ch15-signals-scenes`. Set the viewport to 960 by 540 and name physics layers 1 `world` and 2 `player`.
2. **Do the signal lab first.** Make `signal_lab.tscn` and its script from 15.2.6. Run with F6 and compare the Output with the listing.
3. **Make Pip** (`pip.tscn`, `pip.gd`). Test him in a throwaway scene with walls.
4. **Make the key** (`key.tscn`, `key.gd`) and the **door** (`door.tscn`, `door.gd`). In the throwaway scene, connect the key's `picked_up` signal to a function that prints the key name. Check it prints when Pip touches the key.
5. **Build the room** (`room.tscn`, `room.gd`, `hud_label.gd`). Place three keys. Press F6: collect them; watch the HUD change; see the door turn green; walk into it. Nothing happens on entering yet except that `finished` becomes true. Print a line in `_on_door_entered` to check.
6. **Make the title and result screens** with their scripts. Test each alone with F6: they show, the buttons have focus, the labels are filled (with default values).
7. **Write** `scene_manager.gd` and build `main.tscn`. Set it as the main scene. Press F5 and play a full round: Quick mode, collect the key, enter the door, see the result, go back to the title, see the best time.
8. **Add the demos.** Build `demo_a.tscn` and `demo_b.tscn`, set their exported properties, and check the "Scene-change demo" button.
9. **Run the headless test** `tests/test_flow.gd` and confirm `FAILURES: 0`.

```
godot --headless --path . --script res://tests/test_flow.gd
```

Expected result: Title screen with three buttons. "Quick" fades to the room, where one key lies in the room; collecting it opens the door; stepping into the door fades to the result screen showing the time and "New best time!". "Back to title" shows "Best time: ... s". "Normal" needs all three keys. Clicking a button twice quickly never produces two screens. "Scene-change demo" moves through scenes A and B and back to a fresh title with no best time.

Chapter summary

- A function call needs to know its target; a signal is an announcement that any number of listeners can subscribe to. Signals go up, calls go down.
- `emitter.signal.connect(callable)` connects with no parentheses; arguments must match; `bind()` adds extra ones; lambdas suit short one-liners.
- `CONNECT_ONE_SHOT` removes the connection after one call; `is_connected()` and `disconnect()` manage connections explicitly.
- Declare custom signals with the `signal` keyword, name them in the past tense, type their parameters and emit with `.emit(...)`.
- `await some_signal` pauses a function without freezing the game; after an await, the world may have changed.
- Groups command many nodes at once; a signal bus (an autoload, Chapter 16) serves a few global events; forwarding signals up the tree is the default.
- `change_scene_to_file()` replaces the whole current scene and frees it, so data in it is lost.
- A scene manager keeps a permanent scene, swaps screens in a holder, fades with a tween, guards against double clicks, and passes data through `setup()` before `add_child()`.

Review questions

1. Explain in your own words why a coin that emits `collected` is more reusable than a coin that updates the score label directly.
2. Which is correct for connecting a button handler? (a) `button.pressed.connect(_on_pressed())` (b) `button.pressed.connect(_on_pressed)` (c) `button.pressed.connect("_on_pressed")` (d) `connect(button.pressed, _on_pressed)`
3. A `Button`'s `pressed` signal sends no arguments. How do you make `_on_choice(index: int)` run with `3` when the button is pressed?
4. True or false: connecting the same function to the same signal twice is silently allowed and is harmless.
5. What does `CONNECT_ONE_SHOT` do, and give one situation where it is useful.
6. What is wrong with this declaration, and what should it be? `signal Died()`
7. What does `await get_tree().create_timer(1.5).timeout` do to (a) the function containing it and (b) the rest of the game?
8. After `get_tree().change_scene_to_file("res://level_2.tscn")` what happens to the old scene and the variables inside it? Name two ways to keep a score across the change.
9. Multiple choice: In the scene manager, why is `setup(data)` called before `add_child(next)`? (a) because `add_child` is slow (b) so the data is in place before `_ready()` runs (c) because Godot requires it (d) so the screen is hidden
10. Why does `result.gd` store the data in `setup()` instead of writing it straight into the labels?

Exercises

1. **Practice (easy):** In the signal lab, add a signal `healed(amount: int)`, a `heal()` function that emits it, and two connected handlers: a named function and a lambda. Print what each one receives.
2. **Practice (easy):** Make the HUD turn red when `keys_collected` equals `keys_needed - 1` ("one key left!"). The room should not touch the label's color; instead add a new signal to the room and a new function to `hud_label.gd`.
3. **Apply (medium):** Add a fourth screen `Credits` to the scene manager. It has a Back button, its own `setup`, and is opened by a fourth button on the title screen. Follow the same recipe as the other screens.

4. **Apply (medium):** Keep a list of the last five times (not only the best) in the scene manager and pass them to the title screen, which shows them in a `VBoxContainer` made of labels created in `_ready()`.
5. **Challenge (hard):** Put a short intro line before the room starts: the room shows "Find the keys!" for two seconds using `await`, and Pip cannot move until the message is gone. Hint: give `Pip` a `can_move` variable, and have the room set it after the `await`. Add a check for it to the headless test.

Common mistakes

- **"Invalid call. Nonexistent function" when a signal fires.** Cause: the handler's name was misspelled, or it takes different parameters from those the signal sends. Fix: compare the signal's parameters in the Signals tab with your function.
- **Error "Signal … is already connected".** Cause: `connect()` ran twice for the same callable, for example in a function that runs more than once. Fix: connect in `_ready()` once, or guard with `is_connected`.
- **The handler runs, but with the wrong value.** Cause: `bind()` in a loop with a loop variable that changes, or the order of bound and emitted arguments confused (emitted first, bound last). Fix: print the arguments in the handler and check the order.
- **Null instance error in `setup()`.** Cause: `setup()` uses an `@onready` variable but runs before `add_child()`. Fix: store the data in `setup()` and show it in `_ready()`.
- **A scene change loses the score.** Cause: the score lived in the scene being replaced. Fix: keep it in a permanent node such as the scene manager, or in an autoload (Chapter 16).
- **Two screens appear after a quick double-click.** Cause: nothing blocks a second swap while the first one fades. Fix: add a `_busy` flag and return early.
- **A function with `await` never continues.** Cause: the awaited signal is never emitted, or the node was freed meanwhile. Fix: print before and after the `await`; check that the emitter still exists.

Further reading

- Using signals: https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html
- GDScript basics (signals and await): https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
- Signal class reference: https://docs.godotengine.org/en/stable/classes/class_signal.html
- Callable class reference: https://docs.godotengine.org/en/stable/classes/class_callable.html
- SceneTree class reference (change_scene_to_file and groups): https://docs.godotengine.org/en/stable/classes/class_scenetree.html
- Groups: https://docs.godotengine.org/en/stable/tutorials/scripting/groups.html
- Change scenes manually: https://docs.godotengine.org/en/stable/tutorials/scripting/change_scenes_manually.html

Chapter 16: Autoloads, Game State, and Custom Resources

Learning objectives

By the end of this chapter you will be able to:

- Explain why data stored inside a scene disappears when the scene changes, and how an autoload solves that problem.
- Create an autoload (singleton) script, register it in Project Settings, and use it from any other script.
- Design a small "game state" autoload that holds the player's progress and announces changes with signals.
- Build an event bus: an autoload that contains only signals, so unrelated scenes can talk without knowing each other.
- Write a custom `Resource` class with exported properties and create `.tres` data files from it in the editor.
- Decide when to use a node, an autoload, or a resource for a piece of data, and avoid the common traps of each.

Key terms

- **Autoload**: a script or scene that Godot loads automatically when the game starts and keeps alive for the whole run, no matter which scene is showing.
- **Singleton**: a thing that exists exactly once. An autoload is Godot's version of a singleton.
- **Global state**: data that many parts of a game can read and change, such as the player's gold.
- **Game state**: the facts about the current play-through: score, lives, inventory, the level reached.
- **Event bus**: an autoload that holds only signals, used as a shared message board between scenes.
- **Resource**: a Godot object that holds data and can be saved to a file. Textures, sounds, and scenes are all resources.
- **Custom resource**: a resource type you define yourself by writing a script that `extends Resource`.
- `class_name`: a line that gives a script a global name so it can be used as a type anywhere in the project.
- `@export`: a keyword that makes a script variable appear in the Inspector, so it can be edited without code.
- `.tres` **file**: a text file that stores one resource ("text resource").
- **Shared reference**: when two variables point to the very same object rather than to two copies.
- **Coupling**: how much one piece of code depends on the details of another. Less is better.

16.1 The problem: scenes forget

In Chapter 15 you learned to change from one scene to another with `get_tree().change_scene_to_file()`. That works well, but it hides a surprise that catches every beginner sooner or later.

When Godot changes scenes, it **frees the old scene**. Every node in it is deleted, and every variable stored in those nodes' scripts is deleted with them. That is usually what you want: when you leave the title screen, you do not need its buttons any more. But think about what else lives in a scene:

- The player's score.
- The number of lives left.
- The items in the player's bag.
- Which doors have been opened.

If the score is a variable inside the level scene's script, then going from level 1 to level 2 throws the score away. The new level starts with a fresh `score = 0`, because it is a brand-new copy of the scene with brand-new variables.

Think of a scene as a stage play. When the curtain falls, the actors leave and the props are carried off. If the hero was holding a sword, it is gone. The next play starts with an empty stage. What you need is something that **stays** between plays: a notebook kept in the theater office, where the hero's sword, gold, and health are written down. Every play can read from the notebook when it starts and write to it when something changes.

In Godot, that notebook is called an **autoload**.

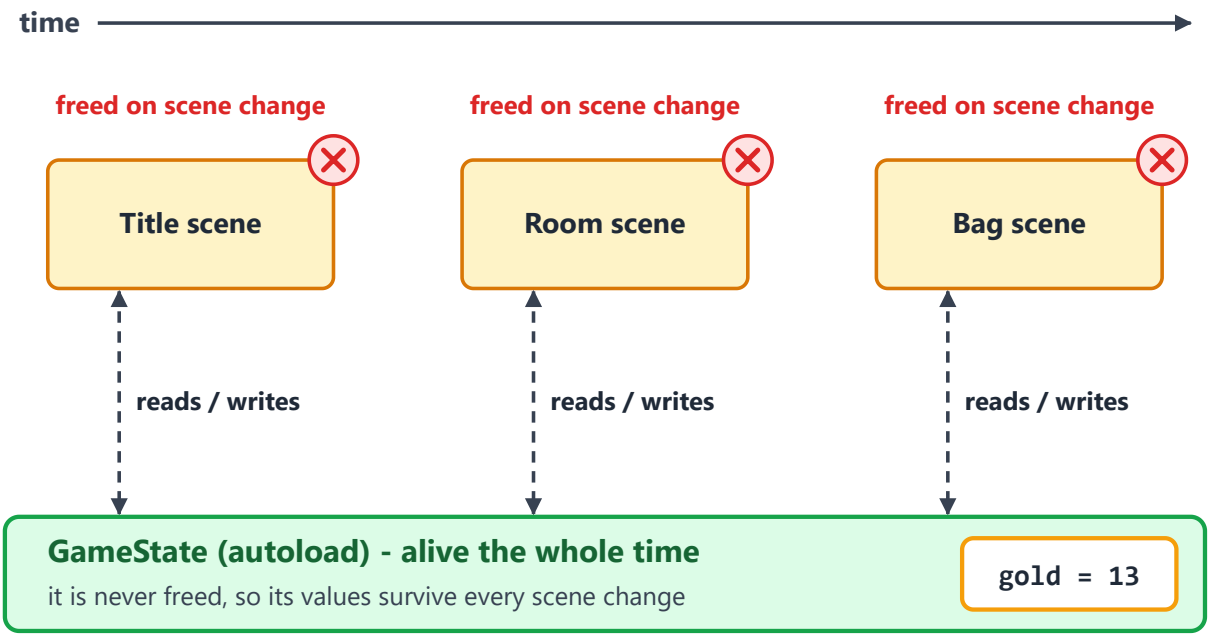


Figure 16.1: Scenes come and go. The autoload stays, so it can remember things between scenes.

How the scene tree really looks

Every running Godot game has a **scene tree**. At the very top is a node called `root` (a `Window`). Normally your current scene is added as a child of `root`. When you change scenes, Godot removes the old scene from `root` and adds the new one.

An autoload is simply a node that Godot adds to `root` **before** your main scene, and never removes. While your game runs, the tree looks like this:

```
root
├── GameState        (autoload, stays forever)
├── Events           (autoload, stays forever)
└── Main             (your current scene, replaced when you change scenes)
```

You can see this yourself. Run any project that has an autoload, then look at the **Remote** tab at the top of the Scene dock. You will see your autoloads sitting next to your scene. (Chapter 22 explores the Remote tree in detail.)

Try it: prove that scenes forget

Try it: Create a new project. Make a scene `a.tscn` with a root `Control`, a `Button` child, and this script:

```
extends Control

var clicks: int = 0

func _ready() -> void:
    $Button.text = "Clicks: %d" % clicks
    $Button.pressed.connect(_on_button_pressed)

func _on_button_pressed() -> void:
    clicks += 1
    $Button.text = "Clicks: %d" % clicks
    if clicks == 3:
        get_tree().change_scene_to_file("res://a.tscn")
```

Click three times. The scene reloads itself and the counter goes back to zero. The variable `clicks` belonged to the scene, and the scene was replaced. Keep this experiment in mind; at the end of the next section we fix it.

16.2 Autoloads (singletons)

An **autoload** is a script (or a whole scene) that you register in the project's settings. Godot then:

1. Creates it automatically when the game starts, before the main scene.
2. Adds it to `root` and keeps it there until the game closes.
3. Gives it a global name, so that **any script in the project can use it directly**.

That last point is the magic. You never use `get_node()` to find an autoload. If you register a script under the name `GameState`, then in every other script you can simply write `GameState.gold`.

Step by step: your first autoload

Let us fix the click counter. Make a new script file called `counter.gd`. A script that will be an autoload usually `extends Node`, because it does not need a position on screen.

```
# counter.gd  (autoload; extends Node)
extends Node

var clicks: int = 0
```

Now register it:

1. Open **Project > Project Settings**.
2. Click the **Globals** tab, then the **Autoload** sub-tab. (In older Godot 4 versions the tab is simply called "Autoload".)
3. Click **Select Script/Scene** (the other two buttons, **Create Script** and **Create Scene**, make a new file instead) and choose `res://counter.gd`.
4. The script appears as a row in the table. The **Name** column shows `Counter` (Godot uses the file name in PascalCase); the columns are **Name**, **Path** and **Global Variable**.
5. Leave the **Enable** box in the **Global Variable** column ticked.

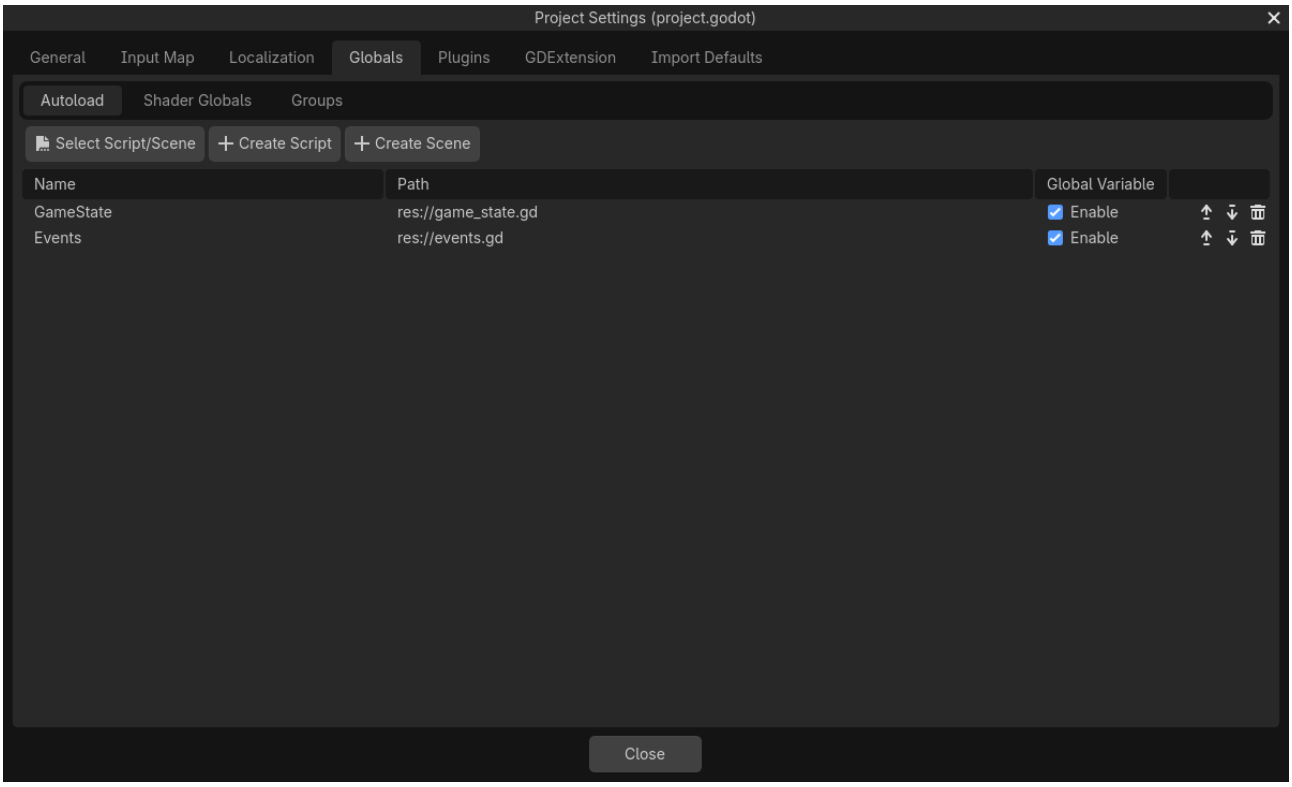


Figure 16.2: Project Settings, Globals, Autoload. Each row is a script that Godot loads for you. The Global Variable box must be ticked for the name to work in code.

Now change the scene's script to use it:

```gdscript
# a.gd  (attach to the root Control of a.tscn)
extends Control


func _ready() -> void:
    $Button.text = "Clicks: %d" % Counter.clicks
    $Button.pressed.connect(_on_button_pressed)


func _on_button_pressed() -> void:
    Counter.clicks += 1
    $Button.text = "Clicks: %d" % Counter.clicks
    if Counter.clicks % 3 == 0:
        get_tree().change_scene_to_file("res://a.tscn")
```

Run it and click. Every third click the scene reloads, but the button text continues from where it was: 3, 4, 5… The number lives in `Counter`, which was never removed. The scene forgets, the autoload remembers.

> **Why it works:** The scene tree contains `root/Counter` the whole time. When `a.tscn` is replaced by a fresh copy of itself, the new copy runs `_ready()` again and reads `Counter.clicks`, which still holds the old value.

Naming and the project file

When you add an autoload, Godot writes a line into `project.godot`:

```
[autoload]

GameState="*res://game_state.gd"
```

The leading `*` means "Global Variable is on". You normally never edit this by hand, but it is good to know where it lives, especially when you use version control (Chapter 22).

Naming rules:

- Use PascalCase for the autoload name: `GameState`, `Events`, `SaveManager`. It will read like a class name in code, which is what it is.
- The name is a *global identifier*. It must not be the same as a built-in class. Do not call an autoload `Input`, `Time`, or `Node`.
- Do **not** give an autoload script a `class_name` that is the same as its autoload name. Godot reports an error ("class hides an autoload singleton"). Pick one of the two. Almost always, the autoload name is enough.
- The file name is usually snake_case (`game_state.gd`) and the autoload name PascalCase (`GameState`). Both point to the same thing.

Autoloads run first, and they run once

Two facts matter when you write code that depends on autoloads:

- **Order.** Autoloads load in the order they appear in the Autoload list, top to bottom. If `GameState` uses `Events` in its `_ready()`, then `Events` must be listed above it. You can drag rows in the list to reorder them.
- **They are created once.** The `_ready()` function of an autoload runs one time, when the game starts. It does not run again when you change scenes.

Scene autoloads

An autoload can also be a **scene** (`.tscn`) instead of a script. That is useful when the always-present thing needs child nodes: for example a "transition" screen with a `ColorRect` that fades in and out between levels, or a music player holding an `AudioStreamPlayer`. In Project Settings you simply choose the `.tscn` file instead of the `.gd` file. Everything else works the same.

Running one scene on its own still works

When you press **F6** (Run Current Scene), Godot still loads all autoloads first. So a scene that reads `GameState.gold` runs fine when started alone. This is one of the big practical benefits of autoloads: you can test any level by itself.

What belongs in an autoload, and what does not

Autoloads are powerful, and with power comes a temptation: put everything in one. Do not. A useful rule is to ask, "Does this need to outlive the scene that uses it?"

Put it in an autoload	Keep it in the scene
Score, lives, gold, inventory	The player's position on screen
Settings (volume, language)	A door's open/closed animation
The list of unlocked levels	A slime's speed
A message bus for signals	Anything one scene alone uses

Beginners often make the opposite mistake: they put a variable in an autoload because it is *easy to reach*, when it really belongs to one scene. The result is a tangle where any script can change any value, and a bug in one place is hard to trace. We come back to this in section 16.7.

16.3 A game state autoload

Now we build the real thing. Most games benefit from one central script that answers the question "How is the player doing right now?" We call it the **game state**. For the chapter project, Pip has a pocket, and the game state holds:

- `gold`: how many gold pieces Pip owns.
- `inventory`: how many of each item Pip carries.
- A way to add items, sell items, and reset everything.

Signals tell the world when something changes

In Chapter 15 you learned custom signals. They are the perfect tool here. The game state should not know about the screen. It should not say "update the label named GoldLabel." Instead it says, in effect, "my gold just changed to 25." Anyone who cares (a label, a sound, a shop door) listens.

```
# game_state.gd  (autoload "GameState"; extends Node)
extends Node
## Autoload "GameState": the one copy of the player's progress.

signal gold_changed(new_gold: int)
signal inventory_changed

var gold: int = 0
var inventory: Dictionary = {}  # item id (String) -> how many (int)


func add_gold(amount: int) -> void:
    gold += amount
    gold_changed.emit(gold)


func reset() -> void:
    gold = 0
    inventory.clear()
    gold_changed.emit(gold)
    inventory_changed.emit()
```

Read this slowly. The data (`gold`, `inventory`) is at the top. The functions that change the data come next, and each one **announces** the change by emitting a signal. The full version appears in the project at the end of the chapter; first let us learn the two ideas it uses.

Rule: change data through functions

You could write `GameState.gold += 5` from any script. It would work, but the signal would not be emitted, so the label showing the gold would not update. Instead, always call a function: `GameState.add_gold(5)`. Functions are the *doorway* into the data. Because every change goes through one doorway, you can add rules in a single place later (for example, "gold can never go below zero") and every caller benefits.

```
func add_gold(amount: int) -> void:
    gold = maxi(0, gold + amount)  # gold can never be negative
    gold_changed.emit(gold)
```

`maxi(a, b)` returns the larger of two integers. (`maxf` is the version for floats.)

A display that listens

A user-interface label that shows the gold should show the right number from the very first moment, and then update whenever the signal arrives. That means two steps in `_ready()`: show the current value, then connect.

```gdscript
# hud.gd  (attach to a Label; extends Label)
extends Label
## Shows the gold from GameState. Works in any scene that contains it.


func _ready() -> void:
    _show_gold(GameState.gold)
    GameState.gold_changed.connect(_show_gold)


func _show_gold(amount: int) -> void:
    text = "Gold: %d" % amount
```

Notice what is missing: there is no `get_node("../Player")` or any path through the scene tree. The label talks only to the autoload. You can drop this `Hud` scene into the title screen, the shop, or level 7, and it will just work.

> **Note:** Why show the current value first? Because the label is created *after* the gold was earned (for example, when level 2 loads). It missed all earlier signals. Reading the current value in `_ready()` catches it up. Forgetting this is the number one reason a freshly loaded scene shows "Gold: 0" for a moment, or forever.

Where does the game state get its list of items?

Our game state should know what kinds of items exist (a coin, a gem, a key). We could write the details into the script, like this:

```gdscript
var coin_value: int = 1
var gem_value: int = 10
var key_value: int = 5
```

That works for three items and becomes unmanageable at thirty. It also mixes **data** (what is a gem worth?) with **logic** (what happens when you sell). Godot has a better tool for data: the **resource**, which is the subject of the next two sections.

16.4 Resources: data with a name and a file

You have already used resources without knowing the word. Every time you dragged a texture into a `Sprite2D`, set a sound on an `AudioStreamPlayer`, or saved a scene, you worked with resources. Here is the definition:

> A **resource** is a chunk of data that Godot can load from a file, keep in memory, and save back to a file.

The base class is called `Resource`. Many built-in classes extend it: `Texture2D`, `AudioStream`, `PackedScene`, `Theme`, `Font`, `Animation`, `StyleBoxFlat`. They differ from nodes in important ways:

Node	Resource
Lives in the scene tree	Does **not** live in the tree
Has a position, can draw, can process each frame	Just holds data (and can have helper functions)

Node	Resource
Usually one per thing on screen	Can be shared by many users
Saved inside a `.tscn`	Saved on its own in a `.tres` file (or inside a scene)

A good mental picture: a node is an **actor**, a resource is a **card in a card catalog**. Ten slimes on screen are ten actors, but they can all read the same card that says "Slime: health 3, speed 80, worth 5 points."

Why custom resources?

Games are full of cards: items, enemies, weapons, levels, dialogue lines, recipes. If you write each one as a block of numbers inside a script, then designing the game means editing code. If each is a resource file, then designing the game means filling in a form in the Inspector. Artists and designers (and future you, at midnight) can add a new item without touching a single line of code.

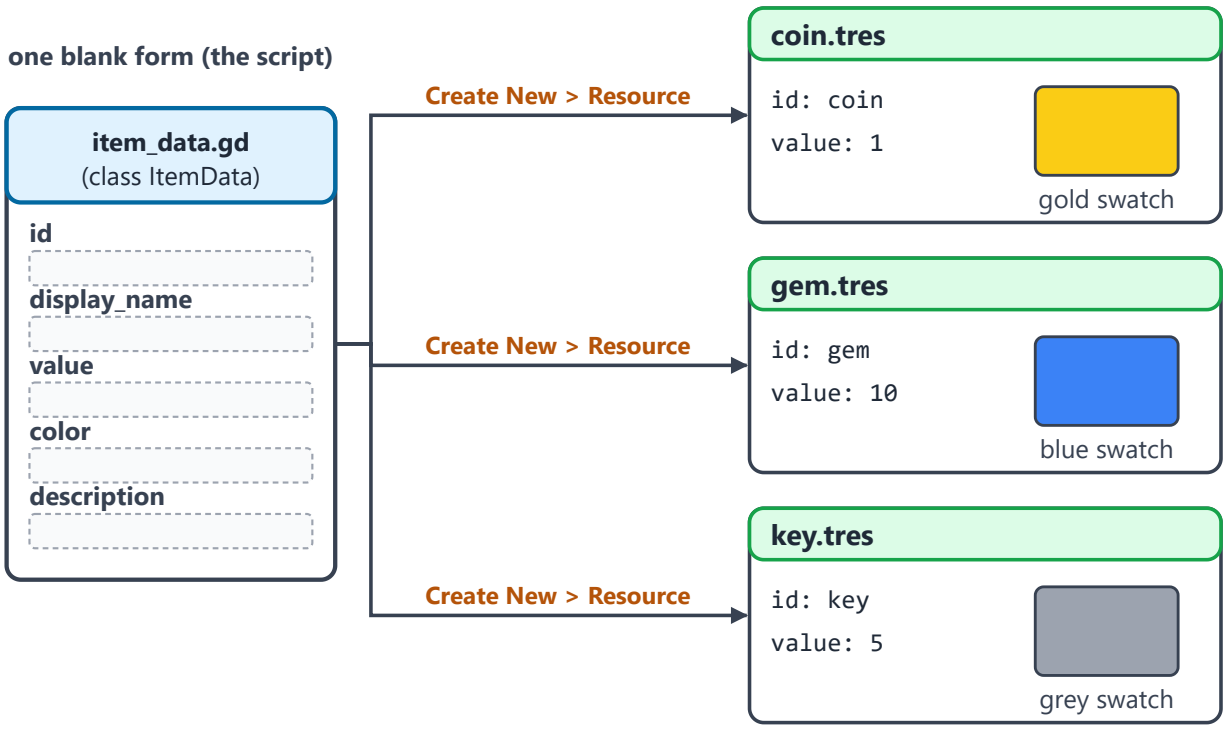


Figure 16.3: One script describes the fields (the blank form). Each `.tres` file is a filled-in form.

Writing a custom resource

A custom resource is a script that `extends Resource`. Create a new script `item_data.gd`:

```gdscript
# item_data.gd  (a Resource, not a Node)
class_name ItemData
extends Resource
## One kind of item. Each .tres file in res://items/ is one ItemData.

@export var id: String = ""
@export var display_name: String = "Item"
@export var value: int = 1
@export var color: Color = Color.WHITE
@export_multiline var description: String = ""
```

Line by line:

- `class_name ItemData` gives the script a global name. Now `ItemData` can be used as a type in any script (`var item: ItemData`), and the editor lists it in the "Create New Resource" dialog. Remember rule 7 from our style guide: never reuse an existing Godot class name. `ItemData` is safe.
- `extends Resource` makes it a resource rather than a node.
- Each `@export var` creates a field that shows in the Inspector. `@export_multiline` gives a big text box for long text.
- Notice there is **no** `_ready()`. Resources have no scene-tree life. They are loaded, they sit there, they hold values.

> **Warning:** Exported variables need an explicit type or a default value so the Inspector knows which editor to show. `@export var value: int = 1` is perfect. Without a type or a default, Godot cannot choose the right control.

Making `.tres` files in the editor

1. In the **FileSystem** dock, right-click the folder where you want the data (make a folder `items` first), and choose **Create New > Resource...**.
2. A dialog opens with a big list of resource types. Type `ItemData` in the search box, select it, and click **Create**.
3. Name the file `coin.tres` and save.
4. The new resource opens in the Inspector. Fill in: Id `coin`, Display Name `Coin`, Value `1`, pick a gold Color, and add a description.
5. Press **Ctrl+S** to save. Repeat for `gem`, `key`, and `potion`.

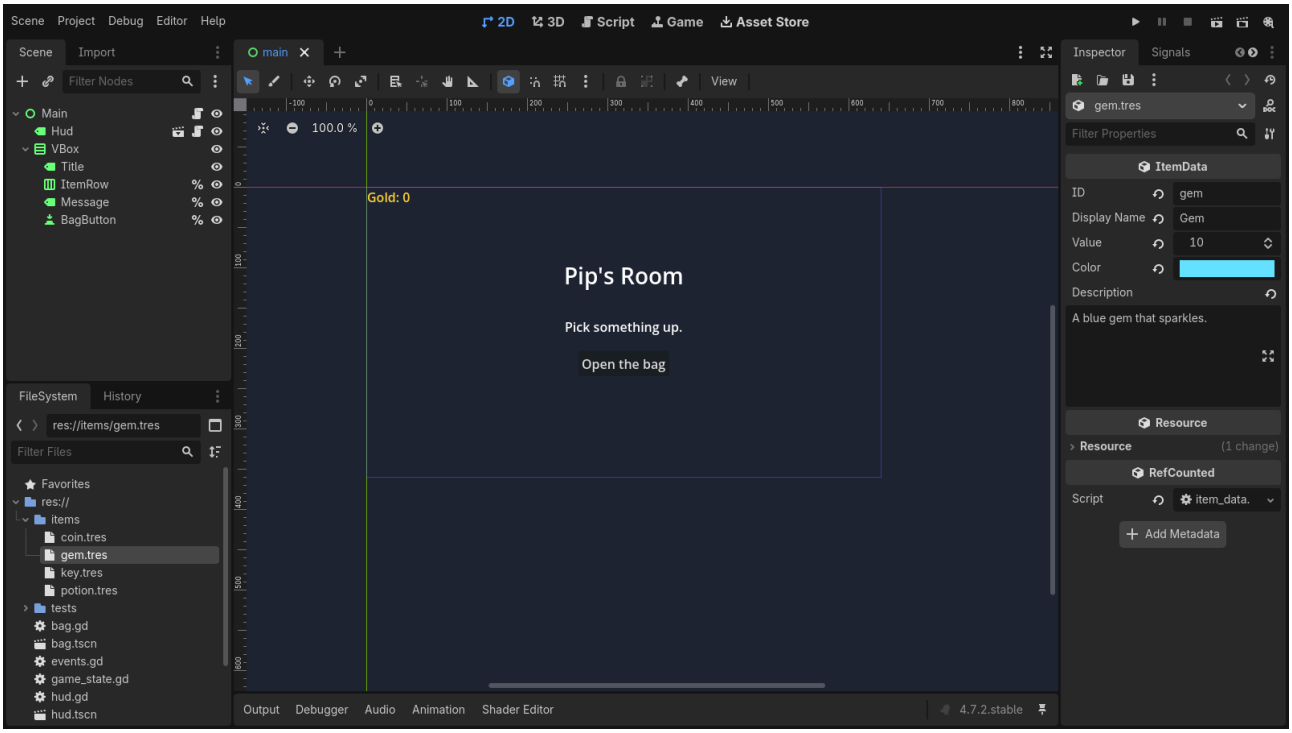


Figure 16.4: A custom resource in the Inspector. The fields are the `@export` variables of the script.

> **Note:** If `ItemData` does not appear in the Create New Resource list, the editor has not noticed the new `class_name` yet. Save the script, then try again; if it still does not show, close and reopen the project. Godot rebuilds its list of global class names when the project loads.

Open `coin.tres` in a text editor (Godot can show it too: right-click the file in the FileSystem dock). It is plain, readable text:

```
[gd_resource type="Resource" script_class="ItemData" load_steps=2 format=3]

[ext_resource type="Script" path="res://item_data.gd" id="1_item"]

[resource]
script = ExtResource("1_item")
id = "coin"
display_name = "Coin"
value = 1
color = Color(1, 0.84, 0.2, 1)
description = "A shiny round coin."
```

This is the same file format as a scene. It names the script that gives it meaning, then lists each saved field. Because it is text, it works nicely with version control: you can see exactly what changed between two versions of the game.

Using a resource in code

There are two common ways to get a resource into a script.

Way 1: let the Inspector fill it in. Export a variable of your resource type, then drag a `.tres` file onto it:

```
@export var item: ItemData
```

Way 2: load it by path. `load()` reads the file; `preload()` does the same while the script is being compiled (so the path must be a plain constant string).

```
var coin: ItemData = load("res://items/coin.tres") as ItemData
const GEM: ItemData = preload("res://items/gem.tres")
```

The `as ItemData` part is a **cast**: it tells the type system "I expect this to be an `ItemData`." If the file is something else, the result is `null` instead of a crash.

Once you have it, use it like any object:

```
print(coin.display_name)   # Coin
print(coin.value * 3)      # 3
```

Resources are shared

This is the most important thing to understand about resources, and the source of the most puzzling bugs.

When you call `load("res://items/coin.tres")` twice, you do **not** get two coins. Godot keeps one copy in memory (a *cache*) and hands out the **same object** both times. If any code changes a field, every other user sees the change.

```
var a: ItemData = load("res://items/coin.tres")
var b: ItemData = load("res://items/coin.tres")
print(a == b)      # true: the very same object

a.value = 99
print(b.value)     # 99: b sees it too!
```

For *read-only data cards* this is wonderful. It saves memory and everything agrees. But it means: **never change a shared resource to track the state of one thing.** If every slime reads "Slime: health 3" and slime A takes damage by doing `data.health -= 1`, then *every* slime loses health.

The fix is to keep the **changing numbers on the node** and the **unchanging facts in the resource**:

```
# slime.gd  (fragment; goes in a script attached to the slime)
@export var data: EnemyData        # shared, never changed
var health: int = 0                # belongs to this one slime

func _ready() -> void:
    health = data.max_health       # copy the starting number out
```

If you truly need a private, changeable copy of a resource, ask for one with `duplicate()`:

```
var my_copy: ItemData = coin.duplicate()
my_copy.value = 99    # only my_copy changes
```

> **Warning:** Another quirk: changes you make to a resource while the game runs are **not** written back to the `.tres` file unless you call `ResourceSaver.save()`. That is usually what you want. We use `ResourceSaver` in Chapter 17.

Resources inside the game state

Now our game state can use item cards. In `game_state.gd` we list the item files and load them when the game starts:

```
const ITEM_PATHS: Array[String] = [
    "res://items/coin.tres",
    "res://items/gem.tres",
    "res://items/key.tres",
    "res://items/potion.tres",
]

var catalog: Array[ItemData] = []


func _ready() -> void:
    for path in ITEM_PATHS:
        var item := load(path) as ItemData
        catalog.append(item)
```

`Array[ItemData]` is a **typed array**: it only accepts `ItemData` objects, and the editor can then autocomplete `catalog[0].display_name`. Note that the inventory stores *ids* (text) and counts, not the resources themselves. Ids are easy to save to a file (Chapter 17), while a resource is a heavier object.

> **Note:** Why a hard-coded list instead of scanning the `items` folder with `DirAccess`? In an exported game, `res://` files are packed inside the game file and are renamed during export, so folder scanning of resources can fail in surprising ways. A short explicit list is dependable. Larger games often put the list into one more resource (a "database").

16.5 An event bus: signals without strings attached

Our `GameState` emits `gold_changed`. That works when the listener already knows `GameState` exists, which is always true for an autoload. But some messages are not about state at all. They are *events*: "the player picked up something," "a door opened," "the boss was defeated," "show this message on the screen."

Imagine a coin in a level. When Pip touches it, the coin wants to say "a coin was collected." Who is listening? The score display, a sound player, a quest tracker, an achievement system. The coin should not need a path to any of them. If the coin script says `get_node("../../UI/ScoreLabel").add(1)`, then moving the label breaks the coin.

The usual solution is the **event bus** (also called a global signal hub): an autoload that does nothing except declare signals.

```gdscript
# events.gd  (autoload "Events"; extends Node)
extends Node
## An "event bus": a shared place for signals that many scenes care about.
## It has no logic of its own, only signals.

@warning_ignore("unused_signal")
signal toast(text: String)
```

Whoever has news calls `Events.toast.emit("Hello!")`. Whoever wants news calls `Events.toast.connect(my_function)`. The two never meet.

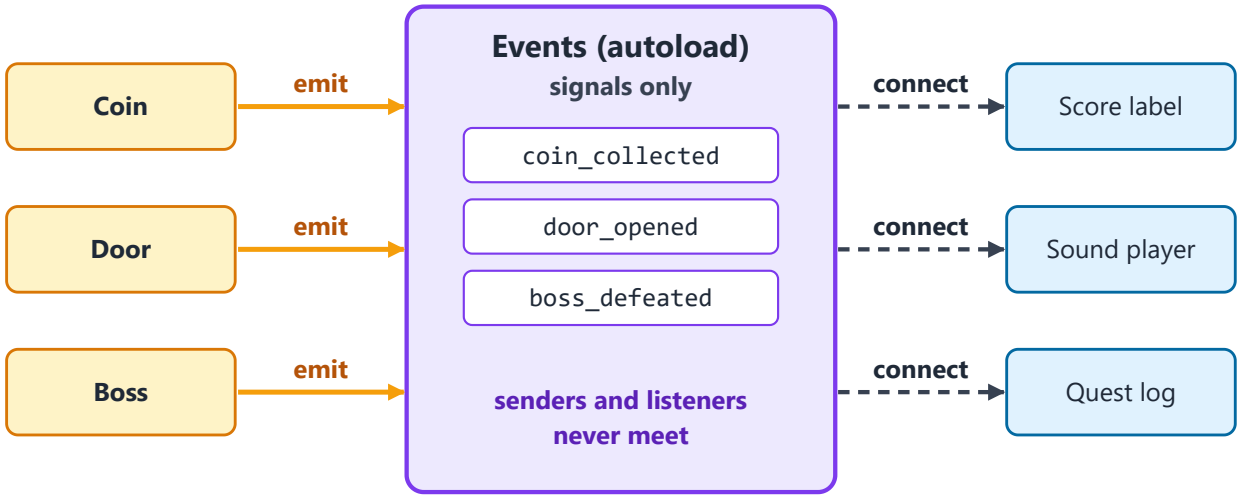


Figure 16.5: Many senders, many listeners, one shared bus in the middle. No sender knows who is listening.

Why the `@warning_ignore("unused_signal")` line? Godot's script checker warns when a script declares a signal but never emits it itself. On an event bus the signals are emitted by *other* scripts, so the warning is a false alarm. The annotation silences it for the next line.

Game state or event bus?

They look similar, so here is how to choose:

- **Game state** answers questions about **now**: "How much gold do I have?" It stores data, and also emits signals when the data changes.

- **Event bus** announces things that **happened**: "A toast message was requested." It stores nothing.

If you can phrase it as a noun ("gold", "lives", "inventory"), it is state. If you phrase it as a past-tense verb ("coin collected", "level finished"), it is an event.

> **Why it works:** Signals are the *only* connection between sender and receiver. Neither imports, names, or finds the other. That is **loose coupling**, and it is the reason big games stay editable. When two scenes are loosely coupled, you can rewrite one of them without touching the other.

Cleaning up

What happens to a connection when the listener scene is deleted (because of a scene change) but the bus lives on? Godot is kind here: when an object is freed, its connections are removed automatically, as long as the callable belongs to that object (a normal method like `_on_toast`). You do not have to disconnect by hand in the common case.

The case that does need care: connecting to the bus from an *autoload* or from a node that is **not** freed, but then adding the same connection again and again (for instance, on every `_ready()` of a node that stays alive). Each call adds one more connection and your function then runs twice, three times... If you suspect this, check first with `is_connected()`:

```
if not Events.toast.is_connected(_on_toast):
    Events.toast.connect(_on_toast)
```

16.6 Putting it together: how scenes use the autoloads

Let us see the shape of the whole program before we build it. Our project has two scenes and two autoloads:

```
Autoloads (always alive)
├── GameState    gold, inventory, catalog of ItemData, add/sell/reset
└── Events       toast(text)

Scenes (replaced when you change scene)
├── main.tscn    "Pip's Room": buttons to pick up items
└── bag.tscn     "Pip's Bag": list of items with a Sell button

Shared pieces
├── hud.tscn     a Label that shows the gold (placed in both scenes)
└── items/*.tres the ItemData cards
```

Follow one click through the system:

1. In the room, you click **Pick up Gem**.
2. `main.gd` calls `GameState.add_item(gem)`.
3. The game state raises the gem count and emits `inventory_changed`, then sends `Events.toast.emit("Picked up Gem (you now have 1)")`.
4. The room's message label hears the toast and shows the text.
5. You click **Open the bag**. The room is deleted; the bag scene appears. It reads the inventory from `GameState` and draws one row per item.
6. You click **Sell all** on the gem. `GameState.sell_all()` removes the gems and calls `add_gold(10)`, which emits `gold_changed`.

7. The Hud label (instanced in the bag scene) hears `gold_changed` and shows "Gold: 10". Go back to the room and the Hud there starts with "Gold: 10" too, because it reads the state in its `_ready()`.

Nobody ever asked a scene for its gold. There is exactly one place that knows it.

16.7 Pitfalls and good habits

Autoloads and resources save a lot of work, and they can also hide trouble. Keep these habits.

1. Do not make everything global. If only the shop screen needs a value, keep it in the shop screen. A rule of thumb: if you can delete the autoload and only one scene breaks, it should not have been an autoload.

2. Autoloads must not depend on a particular scene. Never write `get_node("/root/Main/Player")` inside an autoload. The autoload is alive when `Main` does not exist (during a scene change, or when you run a different scene), and the code will fail. If an autoload needs to talk to a scene, use a signal.

3. Reset state when a new game begins. Because autoloads live through the whole run, the gold from your last play-through is still there when the player presses "New Game" on the title screen. Give your state a `reset()` function and call it when a new run starts. (This is exactly the "Why do I start with 40 gold?" bug.)

4. Know that resources are shared. Keep the changing numbers on nodes or in the game state; keep the resource as read-only data. Use `duplicate()` when you need a private copy.

5. Type your variables. `var item: ItemData` lets the editor catch mistakes such as `item.valeu`. It also makes the code easier to read.

6. Order matters in the Autoload list. If autoload A uses autoload B in its `_ready()`, put B first.

7. Test in isolation. Because autoloads load for every scene, you can run a single scene (F6) and see it work. If it does not work without the "right" starting scene, you have hidden coupling. Fix it before it grows.

> **Note:** Some programmers dislike singletons because global data is easy to misuse. They are right that misuse is easy. They are wrong that you should avoid them: Godot's own `Input`, `Time`, `OS`, and `AudioServer` are singletons. Use autoloads sparingly, for truly global things, and give them tidy functions and signals.

Project 16: Pip's Pocket

Goal: A two-scene game-state demo. In "Pip's Room" you pick up items; in "Pip's Bag" you sell them. The gold counter and item counts survive every scene change, and items are data files (`.tres`), not code.

Time: 45 to 60 minutes.

Files you will create (the finished project is in `projects/ch16-autoloads-resources/`):

```
ch16-autoloads-resources/
├── project.godot
├── icon.svg
├── item_data.gd        custom Resource class
├── items/
│   ├── coin.tres
│   ├── gem.tres
│   ├── key.tres
│   └── potion.tres
├── events.gd           autoload "Events"
```

```
├── game_state.gd       autoload "GameState"
├── hud.gd / hud.tscn   gold label
├── main.gd / main.tscn the room
├── bag.gd / bag.tscn   the bag
└── tests/test_state.gd headless check (optional)
```

Step 1: Create the project

1. In the Project Manager, click **Create**, name it `PipsPocket`, pick an empty folder, and choose the **Compatibility** or **Forward+** renderer (either works).
2. Open **Project > Project Settings > Display > Window** and set Viewport Width `640` and Height `360`. Under **Stretch**, set Mode to `canvas_items`.

Step 2: The item resource

1. Create `item_data.gd` with the code from section 16.4.
2. Create a folder `items`. Right-click it, **Create New > Resource...**, choose `ItemData`, and make four files with these values:

File	Id	Display Name	Value	Color
`coin.tres`	`coin`	Coin	1	gold (1, 0.84, 0.2)
`gem.tres`	`gem`	Gem	10	light blue (0.4, 0.9, 1)
`key.tres`	`key`	Key	5	pale grey (0.85, 0.85, 0.9)
`potion.tres`	`potion`	Potion	3	pink (0.9, 0.35, 0.8)

Step 3: The event bus

Create `events.gd` exactly as shown in section 16.5.

Step 4: The game state

Create `game_state.gd`:

```gdscript
# game_state.gd  (autoload "GameState"; extends Node)
extends Node
## Autoload "GameState": the one copy of the player's progress.

signal gold_changed(new_gold: int)
signal inventory_changed

const ITEM_PATHS: Array[String] = [
    "res://items/coin.tres",
    "res://items/gem.tres",
    "res://items/key.tres",
    "res://items/potion.tres",
]

var gold: int = 0
var catalog: Array[ItemData] = []
var inventory: Dictionary = {}  # item id (String) -> how many (int)


func _ready() -> void:
    for path in ITEM_PATHS:
        var item := load(path) as ItemData
        catalog.append(item)
```

```gdscript
func get_count(item_id: String) -> int:
    return inventory.get(item_id, 0)


func add_item(item: ItemData, amount: int = 1) -> void:
    inventory[item.id] = get_count(item.id) + amount
    inventory_changed.emit()
    Events.toast.emit("Picked up %s (you now have %d)" % [item.display_name, get_count(item.id)])


func sell_all(item: ItemData) -> void:
    var count: int = get_count(item.id)
    if count == 0:
        return
    inventory.erase(item.id)
    add_gold(count * item.value)
    inventory_changed.emit()
    Events.toast.emit("Sold %d %s for %d gold" % [count, item.display_name, count * item.value])


func add_gold(amount: int) -> void:
    gold += amount
    gold_changed.emit(gold)


func reset() -> void:
    gold = 0
    inventory.clear()
    gold_changed.emit(gold)
    inventory_changed.emit()
```

Some details worth noticing:

- `inventory.get(item_id, 0)` asks a Dictionary for a key and gives `0` if it is missing. That is how a never-collected item counts as zero.
- `"%s ... %d" % [a, b]` is string formatting from Chapter 3: `%s` is replaced by text and `%d` by a whole number.
- `sell_all` returns early (`return`) when there is nothing to sell. This guard keeps the toast from saying "Sold 0 Gem."

Step 5: Register both autoloads

Open **Project > Project Settings > Globals > Autoload**. Add `res://events.gd` with name `Events` first, then `res://game_state.gd` with name `GameState`. Make sure the Events row is above the GameState row.

Step 6: The Hud scene

1. Make a new scene with a **Label** as the root. Rename it `Hud`. Save as `hud.tscn`.
2. In the Inspector, set Text to `Gold: 0`. Under **Theme Overrides > Colors**, set Font Color to gold.
3. Set its Position to about (12, 8).
4. Attach `hud.gd` (code in section 16.3).

Step 7: The room scene

Build this scene and save it as `main.tscn`:

```
Main (Control)                main.gd, Layout: Full Rect
├── Hud (instance of hud.tscn)
└── VBox (VBoxContainer)      anchored to the center, about 600 x 180
    ├── Title (Label)         text "Pip's Room", font size 28, centered
    ├── ItemRow (HBoxContainer)   unique name %ItemRow, alignment Center
```

```
    ├── Message (Label)        unique name %Message, centered
    └── BagButton (Button)     unique name %BagButton, text "Open the bag"
```

To give a node a **unique name**, right-click it in the Scene dock and choose **Access as Unique Name**. It shows a `%` beside the node. In code you can then write `%ItemRow` from anywhere in the same scene, no matter where the node sits in the tree.

Attach this script to `Main`:

```gdscript
# main.gd  (attach to Main, a Control)
extends Control
## The "room": pick up items. Scene tree is in main.tscn.

@onready var item_row: HBoxContainer = %ItemRow
@onready var message: Label = %Message


func _ready() -> void:
    for item in GameState.catalog:
        var button := Button.new()
        button.text = "Pick up %s" % item.display_name
        button.add_theme_color_override("font_color", item.color)
        button.pressed.connect(_on_pick_pressed.bind(item))
        item_row.add_child(button)
    Events.toast.connect(_on_toast)
    %BagButton.pressed.connect(_on_bag_pressed)


func _on_pick_pressed(item: ItemData) -> void:
    GameState.add_item(item)


func _on_toast(text: String) -> void:
    message.text = text


func _on_bag_pressed() -> void:
    get_tree().change_scene_to_file("res://bag.tscn")
```

Two ideas are new here:

- **Buttons built in code.** The room does not contain four hand-placed buttons. It loops over the catalog and makes one button per item. Add a fifth `.tres` file and a fifth button appears, with no change to this script (only a new line in `ITEM_PATHS`).
- **`.bind(item)`** attaches an extra argument to a signal connection. The `pressed` signal carries no information, so `bind(item)` tells the function *which* item's button was pressed. Without it, all four buttons would be indistinguishable.

Step 8: The bag scene

Save this scene as `bag.tscn`:

```
Bag (Control)                  bag.gd, Layout: Full Rect
├── Hud (instance of hud.tscn)
└── VBox (VBoxContainer)       anchored to the center, about 500 x 260
    ├── Title (Label)          text "Pip's Bag", font size 28, centered
    ├── List (VBoxContainer)   unique name %List, Size Flags Vertical: Expand
    └── BackButton (Button)    unique name %BackButton, text "Back to the room"
```

```gdscript
# bag.gd  (attach to Bag, a Control)
extends Control
```

```gdscript
## The bag: lists what you own and lets you sell it. Scene tree is in bag.tscn.

@onready var list: VBoxContainer = %List


func _ready() -> void:
    GameState.inventory_changed.connect(_rebuild)
    %BackButton.pressed.connect(_on_back_pressed)
    _rebuild()


func _rebuild() -> void:
    for child in list.get_children():
        child.queue_free()
    var shown: int = 0
    for item in GameState.catalog:
        var count: int = GameState.get_count(item.id)
        if count > 0:
            list.add_child(_make_row(item, count))
            shown += 1
    if shown == 0:
        var empty := Label.new()
        empty.text = "Your bag is empty."
        list.add_child(empty)


func _make_row(item: ItemData, count: int) -> HBoxContainer:
    var row := HBoxContainer.new()
    row.add_theme_constant_override("separation", 12)
    var swatch := ColorRect.new()
    swatch.color = item.color
    swatch.custom_minimum_size = Vector2(24, 24)
    row.add_child(swatch)
    var label := Label.new()
    label.text = "%s x%d  (%d gold each)" % [item.display_name, count, item.value]
    label.custom_minimum_size = Vector2(300, 0)
    row.add_child(label)
    var sell := Button.new()
    sell.text = "Sell all"
    sell.pressed.connect(GameState.sell_all.bind(item))
    row.add_child(sell)
    return row


func _on_back_pressed() -> void:
    get_tree().change_scene_to_file("res://main.tscn")
```

Notice `_rebuild()`: it throws the old rows away and builds new ones whenever the inventory changes. For a small list this "rebuild everything" approach is simplest and hard to get wrong. `queue_free()` deletes the old rows safely at the end of the frame.

Step 9: Set the main scene and run

In **Project Settings > Application > Run**, set **Main Scene** to `res://main.tscn`. Press **F5**.

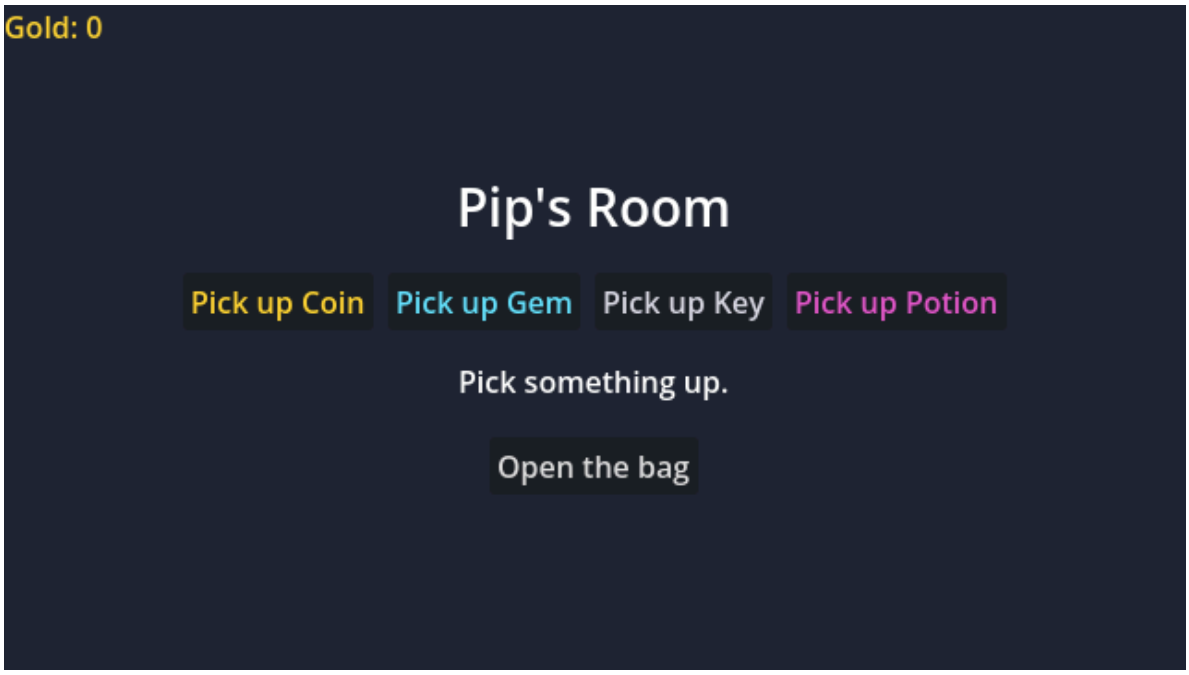


Figure 16.6: Pip's Room. The buttons are made from the ItemData files.

Step 10: Try the checks

1. Pick up 3 coins and 1 gem. The message line follows each click.
2. Click **Open the bag**. You see "Coin x3" and "Gem x1" and the Hud still says "Gold: 0".
3. Sell the gem: Gold becomes 10 and the gem row vanishes.
4. Sell the coins: Gold becomes 13.
5. Click **Back to the room**. The Hud there says "Gold: 13": the state survived two scene changes.

Expected result

You can move between the room and the bag as often as you like, and the gold and items are always right. Picking up more of an item adds to its count. The bag shows "Your bag is empty." when it is empty. If you add a fifth item file (for example `apple.tres`, value 2) and add its path to `ITEM_PATHS`, a new button appears without any other code change.

Optional: a headless self-test

The project includes `tests/test_state.gd`, a script that checks the game state without any window. Run it from a terminal in the project folder:

```
godot --headless --path . -s tests/test_state.gd
```

It prints `TEST PASSED` when adding, selling, and resetting all behave. The test does its work in `_process()` rather than `_initialize()`, because autoloads only finish their own `_ready()` after the tree starts running. (Testing is a good habit; Chapter 22 returns to it.)

Chapter summary

- Scenes are deleted when you change scenes, taking their variables with them. Data that must survive belongs somewhere else.

- An **autoload** is a script or scene Godot creates at startup and keeps alive for the whole run. Register it in Project Settings > Globals > Autoload and use it by name from any script.
- A **game state** autoload holds the player's progress. Change it through functions and announce changes with signals, so displays update themselves.
- An **event bus** is an autoload with only signals. It lets unrelated scenes communicate without knowing about each other (loose coupling).
- A **custom resource** is a script that `extends Resource` with `@export` variables. Each `.tres` file made from it is a data card you edit in the Inspector.
- Resources are **shared**: `load()` returns the same object every time. Treat them as read-only data, keep changing values elsewhere, and use `duplicate()` for a private copy.
- Keep autoloads few and focused. Reset them when a new game starts, and never let them depend on a particular scene.

Review questions

1. Why does a variable in a scene's script lose its value when you call `change_scene_to_file()`? (Short answer)
2. Name the three things Godot does for a script registered as an autoload. (Short answer)
3. Where in the editor do you register an autoload? (Short answer)
4. True or false: you need `get_node("/root/GameState")` to use an autoload in another script.

5. In which situation must the order of the Autoload list matter? (Multiple choice) a) when two autoloads use the same signal name b) when one autoload uses another in its `_ready()` c) when an autoload extends `Control` d) it never matters

6. Why should other scripts call `GameState.add_gold(5)` instead of writing `GameState.gold += 5`? (Short answer)

7. What is the difference between "game state" and an "event bus"? Give one example of each. (Short answer)
8. A resource script declares `@export var damage: int = 3`. You `load()` the same `.tres` file in two different scripts and change `damage` in one. What does the other see, and why? (Short answer)

9. Which method gives you a private, changeable copy of a resource? (Multiple choice) a) `load()` b) `preload()` c) `duplicate()` d) `queue_free()`

10. True or false: a freshly created Hud label should only wait for the `gold_changed` signal; reading `GameState.gold` in `_ready()` is unnecessary.

Exercises

1. **Practice (easy):** Add a fifth item, `apple.tres`, with value 2 and a red color. Register it in `ITEM_PATHS` and confirm that a button appears and can be sold in the bag.
2. **Practice (easy):** Add `@export var rarity: String = "common"` to `ItemData`. Set the gem's rarity to `"rare"` in the Inspector. Show the rarity in the bag rows, e.g. `Gem x1 [rare]`.
3. **Apply (medium):** Add a `New Game` button to the room that calls `GameState.reset()`. Check that the Hud shows 0 and that the bag is empty afterwards. Then make `add_gold` refuse to let gold fall below zero, and add a `spend_gold(amount) -> bool` function that returns `false` (and changes nothing) when the player cannot afford it.

4. **Apply (medium):** Add a second signal to `Events` called `item_sold(item_id: String, total: int)`. Emit it from `sell_all`. Make a new small scene `Achievements` with a Label that listens and shows "Big seller!" the first time a sale is 10 gold or more.
5. **Challenge (hard):** Add a third scene `shop.tscn` where the player can **buy** items for twice their value. Think about where the "buy price" should live (hint: a new `@export` field on `ItemData`). The shop must use `spend_gold()`, show a toast when the player cannot afford something, and the Hud must update in all three scenes.

Common mistakes

- **Symptom:** `Identifier "GameState" not declared in the current scope`. **Cause:** the autoload is not registered, the name is misspelled, or the Global Variable box is unticked. **Fix:** open Project Settings > Globals > Autoload, check the name's spelling and capitalization, and make sure it is enabled.
- **Symptom:** the error `Class "GameState" hides an autoload singleton`. **Cause:** the script has `class_name GameState` and is also an autoload named `GameState`. **Fix:** remove the `class_name` line from the autoload script.
- **Symptom:** a label shows `Gold: 0` after a scene change even though you have gold. **Cause:** the label only listens for the signal and never reads the current value. **Fix:** in `_ready()`, display the current value first, then connect.
- **Symptom:** changing one enemy's data changes all the enemies. **Cause:** they share one resource and you modified it. **Fix:** keep per-enemy values in variables on the enemy node, or call `duplicate()` on the resource.
- **Symptom:** `ItemData` is not found in the Create New Resource dialog. **Cause:** the editor has not rebuilt its class list since you added `class_name`. **Fix:** save the script and reopen the project; check the script has no syntax error.
- **Symptom:** starting a new game gives you the old game's gold. **Cause:** autoloads live for the whole run and nothing reset them. **Fix:** give the state a `reset()` function and call it when a new game starts.

Further reading

- Singletons (Autoload): https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html
- Resources: https://docs.godotengine.org/en/stable/tutorials/scripting/resources.html
- Scene organization (decoupling with signals): https://docs.godotengine.org/en/stable/tutorials/best_practices/scene_organization.html
- Autoloads versus regular nodes: https://docs.godotengine.org/en/stable/tutorials/best_practices/autoloads_versus_internal_nodes.html
- GDScript exports: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_exports.html

Chapter 17: Saving and Loading

Learning objectives

By the end of this chapter you will be able to:

- Explain the difference between data in memory and data on disk, and why a game must save explicitly.
- Find the `user://` folder on your computer and explain why games write there instead of `res://`.
- Store and read settings with `ConfigFile`, including sensible defaults for a first run.
- Write and read game progress as JSON using `FileAccess`, `JSON.stringify`, and `JSON.parse_string`.
- Check loaded data before trusting it, so a damaged or outdated file cannot crash the game.
- Build a settings screen (volume, fullscreen, player name) that applies changes immediately and remembers them.
- Compare three ways to save (ConfigFile, JSON, custom Resource) and choose one for a given job.

Key terms

- **Memory (RAM)**: the fast, temporary storage where a running program keeps its variables. It is emptied when the game closes.
- **Disk**: permanent storage (a hard drive or SSD) where files stay after the game closes.
- **Save file**: a file the game writes so it can restore progress later.
- `user://`: Godot's path prefix for a per-user, writable folder that belongs to your game.
- `res://`: the path prefix for your project's own files. It is read-only in an exported game.
- **ConfigFile**: a Godot class that reads and writes simple `[section] key=value` text files, ideal for settings.
- **JSON**: a popular text format for structured data made of dictionaries, arrays, numbers, strings, booleans, and null.
- **Serialize**: to turn data in memory into text or bytes that can be stored. **Deserialize** is the reverse.
- `FileAccess`: the class for opening, reading, and writing files.
- **Validation**: checking that loaded data has the shape and types you expect before using it.
- **Save version**: a number stored in the save file so a newer game can recognize (and convert) older files.
- **Atomic write**: writing to a temporary file and then renaming it, so a crash never leaves a half-written save.

17.1 Why saving is a separate job

In Chapter 16 the gold and the inventory survived scene changes, because an autoload kept them in memory. But close the game and open it again, and everything is gone. Memory is like a whiteboard: fast and handy, wiped when the power goes off. To keep anything, you must copy it from the whiteboard to a notebook. In computer words: you must **serialize** the data and write it to a **file on disk**.

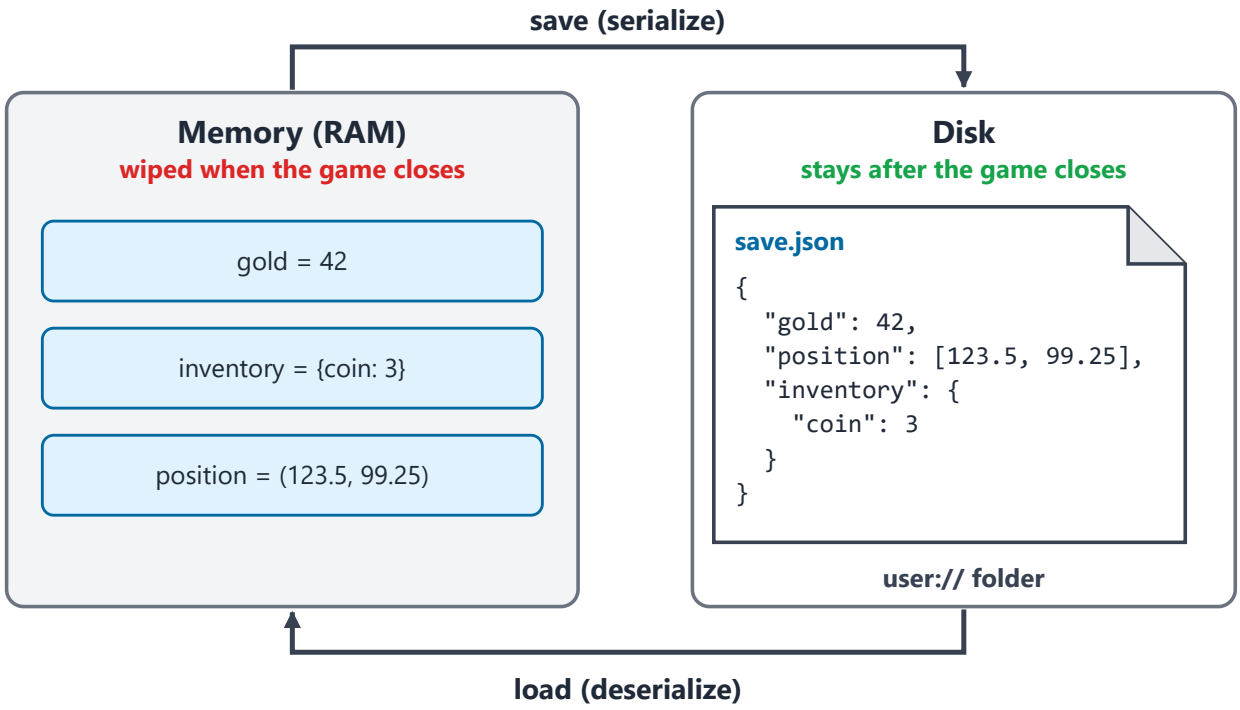


Figure 17.1: Saving copies values from memory into a file. Loading reads the file and puts the values back into variables.

Notice that Godot does not do this for you. There is no "save everything" button. You decide:

1. **What** to save (gold, position, high score... but not, say, the position of every particle).
2. **Where** to save it (which file, in which folder).
3. **In what format** to save it (settings file, JSON, resource).
4. **When** to save and load (on a button, at a checkpoint, when the game quits).

Deciding what to save is a design job, and a good one: the smaller the list, the fewer bugs. A rule that works well is to save **facts**, not **objects**. Save `"gold": 42` and `"position": [120, 80]`, not the whole Player node. When loading, the game rebuilds the nodes from the facts.

Two kinds of saved data

Games usually keep two groups of data separately:

- **Settings**: volume, fullscreen, language, key bindings, the player's chosen name. They belong to the person and the computer, not to a particular play-through. They load when the game starts.
- **Progress**: gold, level reached, inventory, high score. They belong to a play-through (or a "save slot"). They load when the player continues.

Keeping them in two files means resetting a broken save does not also reset the volume to full blast. In this chapter we use `ConfigFile` for settings and JSON for progress, because each fits its job.

17.2 Where files go: `res://` and `user://`

Godot uses two special path prefixes (you met `res://` in Chapter 1).

Prefix	Meaning	Can the game write to it?
`res://`	Your project folder. Scenes, scripts, art.	In the editor yes, but in an exported game **no**: the files are packed inside the game and are read-only.
`user://`	A folder made for your game's saved data.	**Yes**, always.

So: **everything the game saves goes in `user://`**. Never try to save to `res://`. It may seem to work while you test in the editor, and then silently fail in the exported game.

Where is `user://` on my computer?

Godot picks a sensible, hidden location for each operating system. The folder name is based on your project's name (**Project Settings > Application > Config > Name**):

System	Location
Windows	`%APPDATA%\Godot\app_userdata\<project name>\`
macOS	`~/Library/Application Support/Godot/app_userdata/<project name>/`
Linux	`~/.local/share/godot/app_userdata/<project name>/`
Web	A browser storage area (IndexedDB), not a normal folder

You do not need to memorize this. In the editor, choose **Project > Open User Data Folder** and your file manager opens at the right place. In code, you can ask for the real path like this:

```
print(ProjectSettings.globalize_path("user://"))
```

`globalize_path` converts a Godot path into the full operating-system path. Our chapter project prints this at the bottom of the screen so you can find your save files.

> **Note:** The folder is created for you the first time you write something. A friendly detail: if you rename the project, the folder name changes, and old saves seem to vanish. They are still in the old folder. For a game you plan to ship, you can fix the folder name in **Project Settings > Application > Config > Use Custom User Dir** and **Custom User Dir Name**.

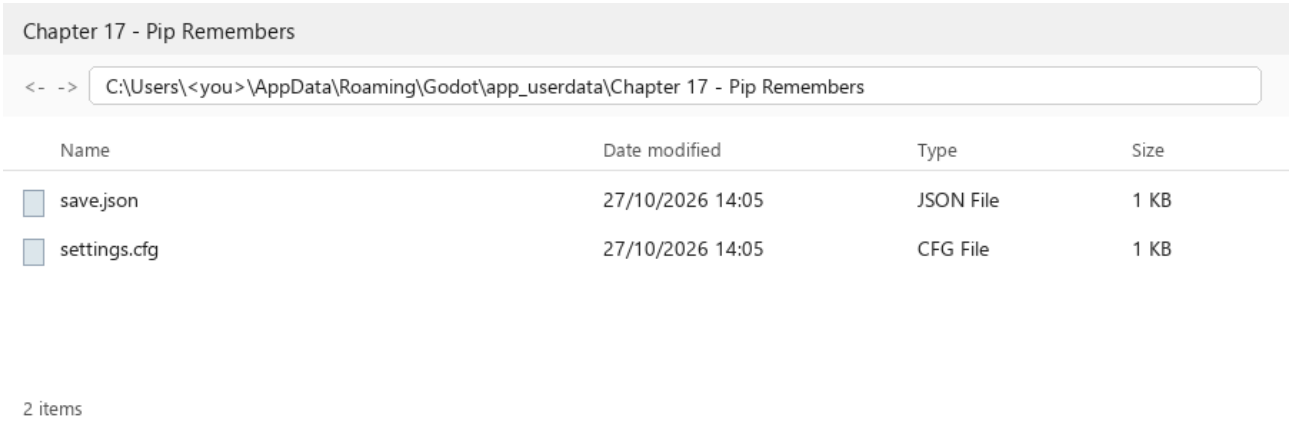


Figure 17.2: Open User Data Folder shows the two files this chapter's project writes.

Never save to a place the player cannot find

A common beginner idea is to save beside the executable or to hard-code `C:/Users/me/...`. Both break on other computers. Always use `user://`; Godot translates it for every system, including the web and mobile.

17.3 Settings with ConfigFile

`ConfigFile` is Godot's tool for the classic settings file. The file looks like this, and you can open it in any text editor:

```
[player]

name="Mallow"

[audio]

volume=0.35

[video]

fullscreen=true
```

There are **sections** (in square brackets) and **keys** with values inside them. It is neat for settings, because humans can read and edit it, and because each value remembers its type (the quotes around `"Mallow"` tell Godot it is text; `true` is a bool; `0.35` is a float).

Writing a ConfigFile

```gdscript
# fragment: put it inside any function
var config := ConfigFile.new()
config.set_value("player", "name", "Mallow")
config.set_value("audio", "volume", 0.35)
config.set_value("video", "fullscreen", true)
var err: Error = config.save("user://settings.cfg")
if err != OK:
    push_error("Could not save settings: " + error_string(err))
```

- `ConfigFile.new()` makes an empty settings object in memory.
- `set_value(section, key, value)` stores a value. The section is created if needed.
- `save(path)` writes the file and **returns an `Error` code**. `OK` (which is the number 0) means success.
- `error_string(err)` turns an error code into readable words, like "File not found."

Always look at the returned error. Disks fill up, folders are locked, and antivirus programs interfere. A game that ignores the error says "Saved!" and has saved nothing.

Reading a ConfigFile, with defaults

```gdscript
# fragment: put it inside any function
var config := ConfigFile.new()
var err: Error = config.load("user://settings.cfg")
if err != OK:
    print("No settings file yet. Using defaults.")
    return
var player_name: String = config.get_value("player", "name", "Pip")
var volume: float = config.get_value("audio", "volume", 0.8)
```

The third argument of `get_value` is a **default**: what you get when that key does not exist. This is the safety net for two common situations:

- **First run.** There is no file at all (`load` returns `ERR_FILE_NOT_FOUND`). The game should not crash or complain; it should simply use its defaults.
- **A newer game, an older file.** You added a `language` setting in version 2. A player's version-1 file has no such key. The default fills the gap.

Other useful ConfigFile functions: `has_section_key(section, key)`, `get_sections()`, `get_section_keys(section)`, `erase_section(section)`, and `erase_section_key(section, key)`.

The SaveManager autoload

We put the saving code in an autoload called `SaveManager`, using what you learned in Chapter 16. The settings part looks like this:

```gdscript
# save_manager.gd  (autoload "SaveManager"; extends Node) -- settings part
extends Node

signal settings_changed

# Paths are variables (not constants) so a test can point them somewhere else.
var settings_path: String = "user://settings.cfg"

var player_name: String = "Pip"
var volume: float = 0.8  # 0.0 (silent) to 1.0 (full)
var fullscreen: bool = false


func _ready() -> void:
    load_settings()
    apply_settings()


func save_settings() -> Error:
    var config := ConfigFile.new()
    config.set_value("player", "name", player_name)
    config.set_value("audio", "volume", volume)
    config.set_value("video", "fullscreen", fullscreen)
    return config.save(settings_path)


func load_settings() -> Error:
    var config := ConfigFile.new()
    var err: Error = config.load(settings_path)
    if err != OK:
        return err  # first run, or no file: keep the defaults
    player_name = str(config.get_value("player", "name", player_name))
    volume = clampf(float(config.get_value("audio", "volume", volume)), 0.0, 1.0)
    fullscreen = bool(config.get_value("video", "fullscreen", fullscreen))
    return OK
```

Look at how `load_settings` treats what it reads. `get_value` returns a **Variant** (any type), because the file could contain anything, even if someone edited it by hand and typed `volume="loud"`. Wrapping the result in `float(...)`, `str(...)`, and `bool(...)` converts it to the type we want, and `clampf(x, 0.0, 1.0)` keeps the volume inside the legal range. The principle: **never trust a file**. It was written by a program you cannot see right now, perhaps an older version of your own game.

Applying settings

Reading a setting is not enough; the game must *act* on it. Two built-in singletons do the work:

```
func apply_settings() -> void:
    AudioServer.set_bus_volume_db(0, linear_to_db(volume))
    if fullscreen:
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_FULLSCREEN)
    else:
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_WINDOWED)
    settings_changed.emit()
```

- `AudioServer.set_bus_volume_db(0, ...)` sets the volume of audio bus number 0, which is the **Master** bus (Chapter 11). Buses use **decibels**, a logarithmic scale where 0 dB is full and -80 dB is nearly silent. Our setting is a friendly number between 0 and 1, and `linear_to_db()` converts it. (Passing `0.0` gives `-inf`, which Godot treats as silence.)
- `DisplayServer.window_set_mode()` switches the game window between windowed and fullscreen.

We keep "load" and "apply" as two separate functions on purpose. Loading changes variables; applying changes the world. Separating them lets the settings screen apply a change instantly without saving it to disk on every slider tick.

17.4 Progress with JSON and FileAccess

For game progress, a structured format is better than flat key/value pairs, because progress has lists and nested pieces: an inventory is a dictionary inside the save. **JSON** (JavaScript Object Notation) is the world's most common format for this, and it maps directly onto what you already know:

JSON	GDScript
object `{ "a": 1 }`	`Dictionary`
array `[1, 2, 3]`	`Array`
number `3.5`	`float` (or `int`, see below)
string `"hi"`	`String`
`true` / `false`	`bool`
`null`	`null`

A save file might look like this:

```
{
    "version": 1,
    "gold": 42,
    "inventory": {
        "coin": 3,
        "gem": 1
    },
    "position": [
        123.5,
        99.25
    ],
    "play_time": 61.5
}
```

Step 1: turn your data into a Dictionary

```
func to_dict() -> Dictionary:
    return {
        "version": SAVE_VERSION,
        "gold": gold,
```

```
        "inventory": inventory,
        "position": [pip_position.x, pip_position.y],  # JSON has no Vector2
        "play_time": play_time,
    }
```

JSON can hold only the types in the table. A `Vector2`, `Color`, or `Node` cannot go in directly. You convert them to plain pieces: a `Vector2` becomes an array of two numbers; a `Color` could become `[r, g, b, a]` or a hex string (`color.to_html()`).

The `"version": SAVE_VERSION` line is your future self's best friend. When you change the game and the save layout changes, the version number tells your loader which layout it is looking at.

Step 2: turn the Dictionary into text and write it

```
func save_game() -> Error:
    var text: String = JSON.stringify(to_dict(), "\t")
    var file := FileAccess.open(save_path, FileAccess.WRITE)
    if file == null:
        return FileAccess.get_open_error()
    file.store_string(text)
    file.close()
    return OK
```

- `JSON.stringify(value, indent)` converts data to JSON text. The `"\t"` argument asks for tab-indented, human-readable output. Without it the whole save is one long line (shorter, harder to read).
- `FileAccess.open(path, mode)` opens a file. `FileAccess.WRITE` creates the file, or empties it if it exists. It returns `null` if opening failed, and `FileAccess.get_open_error()` then tells you why.
- `store_string` writes the text. `close()` finishes the job and lets the operating system flush the data. (The file also closes itself when the variable goes out of scope, but closing explicitly makes the intention clear.)

Step 3: read the file and parse it

```
func load_game() -> bool:
    if not has_save():
        return false
    var text: String = FileAccess.get_file_as_string(save_path)
    var json := JSON.new()
    if json.parse(text) != OK:
        push_warning("Save file is damaged (line %d): %s" % [json.get_error_line(),
json.get_error_message()])
        return false
    if typeof(json.data) != TYPE_DICTIONARY:
        return false
    return from_dict(json.data)
```

- `FileAccess.file_exists(path)` checks for the file (we wrapped it as `has_save()`).
- `FileAccess.get_file_as_string(path)` reads the whole file as text in one call.
- `JSON.new()` and `json.parse(text)` parse step by step and return an error code; on failure `get_error_line()` and `get_error_message()` say where and why. There is also a one-line shortcut, `JSON.parse_string(text)`, which returns `null` on failure. It is handy for quick tests, but you learn nothing about *why* it failed.
- `json.data` holds the result once parsing has succeeded. We insist that it is a Dictionary before going on.

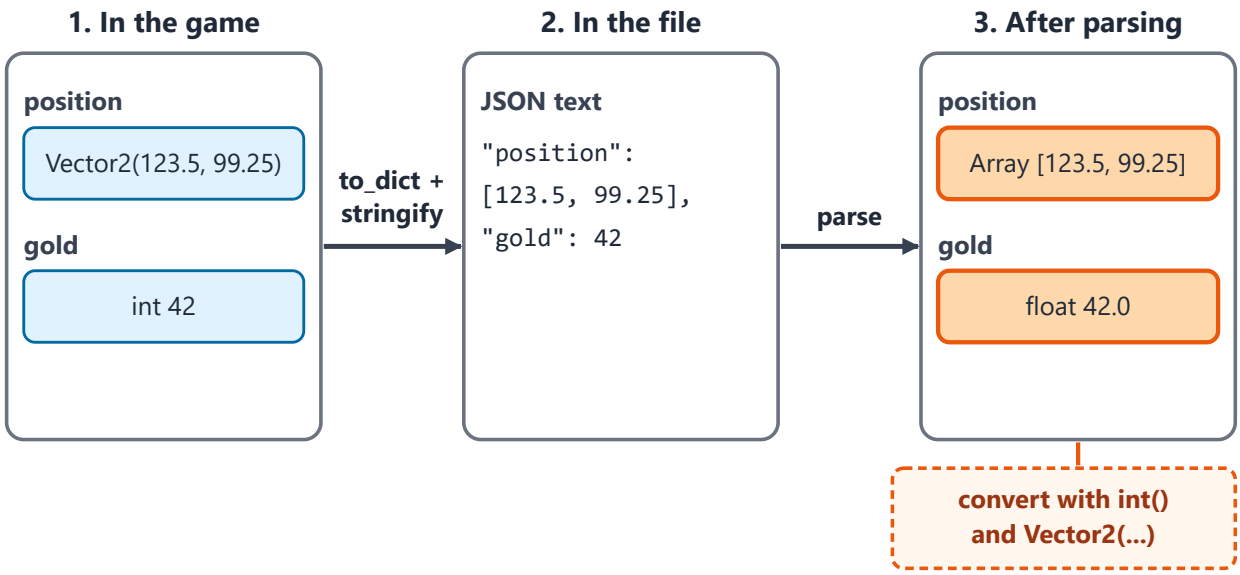


Figure 17.3: Two JSON surprises. Vectors must be converted by hand, and every number comes back from the parser as a float.

Gotcha 1: all JSON numbers come back as floats

JSON does not distinguish whole numbers from decimals; it just has "number". Godot's parser reads them all as `float`. So a saved `"gold": 42` is loaded as `42.0`. If your code then does `str(gold)` or uses it as an array index, you may see `42.0` or an error. The fix is to convert explicitly when loading: `gold = int(data["gold"])`. This applies to dictionary values too: a saved inventory `{"coin": 3}` loads as `{"coin": 3.0}`. In our `from_dict` we rebuild the inventory with `int(...)` on every count.

Gotcha 2: validate everything

A file from disk can be: missing a key, from an old version, hand-edited, empty, or half-written after a crash. If your loader assumes a perfect file, then `data["gold"]` on a missing key crashes the game, and worse, a *half-applied* load leaves the game in a broken, mixed-up state.

The strategy that handles all of this is **validate first, apply last**:

```gdscript
# Returns true if the data was valid and has been applied.
# Nothing is changed unless EVERYTHING checks out.
func from_dict(data: Dictionary) -> bool:
    if int(data.get("version", 0)) != SAVE_VERSION:
        return false
    var new_gold: Variant = data.get("gold")
    var new_inventory: Variant = data.get("inventory")
    var new_position: Variant = data.get("position")
    var new_time: Variant = data.get("play_time")
    if not _is_number(new_gold) or not _is_number(new_time):
        return false
    if typeof(new_inventory) != TYPE_DICTIONARY:
        return false
    if typeof(new_position) != TYPE_ARRAY or new_position.size() != 2:
        return false
    if not _is_number(new_position[0]) or not _is_number(new_position[1]):
        return false
    var clean_inventory: Dictionary = {}
    for key in new_inventory:
        if not _is_number(new_inventory[key]):
            return false
        clean_inventory[str(key)] = int(new_inventory[key])  # JSON numbers come back as floats
```

```
    gold = int(new_gold)
    inventory = clean_inventory
    pip_position = Vector2(float(new_position[0]), float(new_position[1]))
    play_time = float(new_time)
    return true


func _is_number(value: Variant) -> bool:
    return typeof(value) == TYPE_INT or typeof(value) == TYPE_FLOAT
```

Read it in two halves. The top half only *looks*: it copies each piece into a local variable and checks its type, returning `false` at the first problem. The bottom four lines (starting at `gold = int(new_gold)`) are the only ones that touch the real game, and they run only if every check passed. `Variant` is the type name for "anything"; `typeof()` returns a constant such as `TYPE_INT` that names what a value really is. `data.get("gold")` returns `null` for a missing key, which fails `_is_number`, so a missing key is handled by the same check.

The caller gets a simple `true` or `false` and can show "Could not load that save" instead of crashing.

Gotcha 3: a crash during saving

Opening a file with `WRITE` empties it immediately. If the game (or the computer) dies before the new text is fully written, the player has lost both the new save *and* the old one. A professional habit is the **atomic write**: write the new data to a temporary file first, and only when that has fully succeeded, rename it over the real file.

```
func save_game() -> Error:
    var text: String = JSON.stringify(to_dict(), "\t")
    # Write to a temporary file first, then rename. If the game crashes
    # half way through writing, the old save is still safe.
    var temp_path: String = save_path + ".tmp"
    var file := FileAccess.open(temp_path, FileAccess.WRITE)
    if file == null:
        return FileAccess.get_open_error()
    file.store_string(text)
    file.close()
    var dir := DirAccess.open(save_path.get_base_dir())
    if dir == null:
        return DirAccess.get_open_error()
    return dir.rename(temp_path.get_file(), save_path.get_file())
```

`DirAccess` is the folder counterpart of `FileAccess`. `DirAccess.open("user://")` gives you a handle to the folder, and `rename(from, to)` renames a file in it; if the target name already exists, the rename replaces it (the project's headless test checks this, see below). `get_base_dir()` returns the folder part of a path, and `get_file()` the file-name part. Deleting a save is also a `DirAccess` job:

```
func delete_save() -> Error:
    var dir := DirAccess.open(save_path.get_base_dir())
    if dir == null:
        return DirAccess.get_open_error()
    return dir.remove(save_path.get_file())
```

An alternative: `var_to_str` and `str_to_var`

If you want to keep Godot types such as `Vector2` and `Color` without converting them by hand, Godot has its own text format. These are *fragments* showing the idea:

```
# fragment
var text: String = var_to_str({"pos": Vector2(1, 2), "tint": Color.RED})
var back: Variant = str_to_var(text)   # Vector2 and Color come back as themselves
```

It is convenient, but the format is Godot-only, so other tools cannot read it, and (important) it is meant for your own files. JSON is the better choice when the data might be read by other programs, a website, or a friend's tool.

> **Warning:** Never load a save file with `FileAccess.get_var(true)` or `bytes_to_var_with_objects()` unless you wrote that file yourself and no one else can supply it. Those variants can create objects, including objects with scripts. The safe forms (JSON, ConfigFile, `str_to_var`) only create plain data.

17.5 The third way: saving a custom Resource

In Chapter 16 you made `ItemData` resources. A resource can also be a **save file**. You define a class that holds your progress, and Godot's `ResourceSaver` writes it as a `.tres` (text) or `.res` (binary) file:

```gdscript
# save_bundle.gd  (a Resource)
class_name SaveBundle
extends Resource
## A save file made from a custom Resource (the third way to save).

@export var gold: int = 0
@export var pip_position: Vector2 = Vector2.ZERO
@export var inventory: Dictionary = {}
```

```gdscript
# fragment: saving and loading a SaveBundle
var bundle := SaveBundle.new()
bundle.gold = 11
bundle.pip_position = Vector2(4, 5)
bundle.inventory = {"key": 1}
ResourceSaver.save(bundle, "user://bundle.tres")

var loaded := ResourceLoader.load("user://bundle.tres", "", ResourceLoader.CACHE_MODE_IGNORE) as
SaveBundle
print(loaded.gold)   # 11
```

Advantages: very little code, types are preserved (`int` stays `int`, `Vector2` stays `Vector2`, no manual conversion), and the Inspector can open the file for debugging. The `CACHE_MODE_IGNORE` argument tells the loader to read the file again instead of returning the object cached in memory. Without it, loading right after saving in the same run could return the old in-memory object (remember: resources are shared).

The disadvantages are serious for a *shipped game*:

- A `.tres` file can contain **script references**. A player who downloads a "save file" from the internet and drops it into their folder could run code you never wrote. JSON and ConfigFile files cannot do that.
- Any change to the class (renaming a variable) can break old saves in confusing ways.
- Only Godot can read them.

	ConfigFile	JSON + FileAccess	Custom Resource
Best for	Settings	Game progress, shared saves	Quick prototypes, editor tools
Human readable	Yes, very	Yes	Yes (`.tres`)
Keeps Godot types	Yes	No (convert by hand)	Yes
Safe against hand-made files	Yes	Yes	**No** (scripts)
Readable by other programs	Somewhat	Yes	No

A sensible default for a beginner shipping a game: **ConfigFile for settings, JSON for progress.** That is what we use.

> **Why it works:** All three do the same job: turn memory into a file and back. What differs is who does the converting. With JSON you do it by hand (and so you control and check everything); with Resource Godot does it (and so you also accept whatever the file says).

17.6 A settings screen

Settings change the world immediately: drag a volume slider and you hear the music get quieter. Our screen follows three rules:

1. **Show the current values** when the screen opens.
2. **Apply changes at once** as the player moves a control.
3. **Save to disk** when the player leaves the screen (not on every tick of the slider, which would write the file hundreds of times).

The scene tree for `settings.tscn`:

```
Settings (Control)                settings.gd, Layout: Full Rect
└── VBox (VBoxContainer)          centered, about 400 x 260
    ├── Title (Label)             "Settings", font size 28, centered
    ├── NameLabel (Label)         "Your name"
    ├── NameEdit (LineEdit)       unique name %NameEdit, Max Length 16
    ├── VolumeLabel (Label)       unique name %VolumeLabel
    ├── VolumeSlider (HSlider)    unique name %VolumeSlider, Min 0, Max 1, Step 0.05
    ├── FullscreenCheck (CheckButton)  unique name %FullscreenCheck, text "Fullscreen"
    └── Buttons (HBoxContainer)   centered
        ├── ResetButton (Button)  unique name %ResetButton, "Reset to defaults"
        └── BackButton (Button)   unique name %BackButton, "Save and go back"
```

```gdscript
# settings.gd  (attach to Settings, a Control)
extends Control
## Settings screen. Scene tree is in settings.tscn.

@onready var name_edit: LineEdit = %NameEdit
@onready var volume_slider: HSlider = %VolumeSlider
@onready var volume_label: Label = %VolumeLabel
@onready var fullscreen_check: CheckButton = %FullscreenCheck


func _ready() -> void:
    _show_current_values()
    # Connect AFTER filling in the values, so filling them does not trigger the handlers.
    volume_slider.value_changed.connect(_on_volume_changed)
    fullscreen_check.toggled.connect(_on_fullscreen_toggled)
    name_edit.text_changed.connect(_on_name_changed)
    %ResetButton.pressed.connect(_on_reset_pressed)
    %BackButton.pressed.connect(_on_back_pressed)


func _show_current_values() -> void:
    name_edit.text = SaveManager.player_name
    volume_slider.value = SaveManager.volume
    fullscreen_check.button_pressed = SaveManager.fullscreen
    _update_volume_label()


func _update_volume_label() -> void:
    volume_label.text = "Volume: %d%%" % roundi(SaveManager.volume * 100.0)
```

```
func _on_volume_changed(value: float) -> void:
    SaveManager.volume = value
    SaveManager.apply_settings()
    _update_volume_label()


func _on_fullscreen_toggled(pressed: bool) -> void:
    SaveManager.fullscreen = pressed
    SaveManager.apply_settings()


func _on_name_changed(new_text: String) -> void:
    SaveManager.player_name = new_text


func _on_reset_pressed() -> void:
    SaveManager.reset_settings()
    _show_current_values()


func _on_back_pressed() -> void:
    SaveManager.save_settings()
    get_tree().change_scene_to_file("res://main.tscn")
```

Several details deserve attention:

- **Connect after filling.** If we connected `value_changed` first and then set `volume_slider.value`, the handler would fire during setup. Setting up in the right order avoids a confusing extra call. (A `CheckButton`'s `button_pressed` property, by contrast, can also be set silently with `set_pressed_no_signal(true)`; both approaches work.)
- **`%d%%`** in a format string: `%d` is the number, and `%%` prints a literal percent sign.
- **`roundi()`** rounds a float to the nearest integer. Without it, 0.35 * 100 can display as `35.00000000000001`.
- **`HSlider`** emits `value_changed(value: float)` while the player drags. Setting Step to `0.05` makes it move in 5% jumps.
- **`reset_settings()`** (in the SaveManager) puts defaults in all three variables and applies them. The screen then re-reads the values so the controls match.

The reset function is short:

```
func reset_settings() -> void:
    player_name = "Pip"
    volume = 0.8
    fullscreen = false
    apply_settings()
```

> **Note:** A very common improvement is to also save the settings the moment the player *closes the game* rather than only when leaving this screen. In a script on an autoload, handle `NOTIFICATION_WM_CLOSE_REQUEST` in `_notification()` and call `save_settings()` there. Our project saves when leaving the screen, which is simple and enough for now.

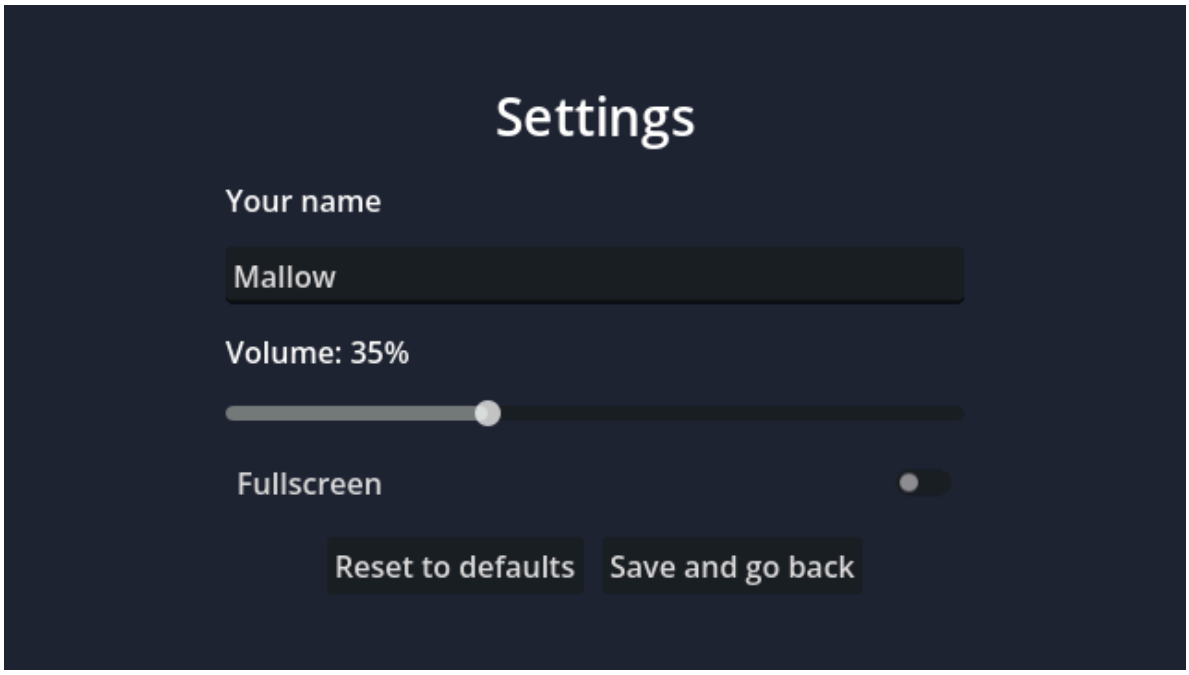


Figure 17.4: The finished settings screen.

17.7 When to save, and good habits

When to save. A good game saves at moments the player expects: after a button press ("Save"), at a checkpoint, on finishing a level, or when the game closes. Saving every frame is wasteful and can wear out disks; saving only when the game closes loses everything in a crash. A middle path is common: autosave at checkpoints plus a manual save.

Keep a version. The `"version"` key is a one-line insurance policy. When version 2 arrives with a different layout, `from_dict` can say "if version is 1, upgrade the data like this; if 2, load directly; otherwise refuse." Without a version, you can only guess.

Tell the player. Show a message when a save succeeds or fails (our main scene does). Silent failure is the worst kind.

Never delete without a plan. The Delete button in the project removes the save file. In a real game you would ask "Are you sure?" first (a `ConfirmationDialog` node), and might move the file to a backup instead.

Test with a bad file. Break your own save on purpose: delete a key, type nonsense, make the file empty. Your loader should refuse politely and leave the game as it was. We do exactly that in the headless test below.

Do not store secrets or trust the player's file. A save file on a player's computer can be edited by the player. For a single-player game that is normally fine (their game, their rules). Never use a local save file as proof of anything in an online game.

Mind the web. In a web export, `user://` is stored in the browser, and some browsers clear it when the player clears site data. If you export to the web (Chapter 23), tell the player their progress lives in the browser.

Project 17: Pip Remembers

Goal: A small program where Pip can move around, collect coins into a counter, and have the position, gold, inventory, and play time saved to a JSON file and restored later. A settings screen stores name, volume, and fullscreen in a ConfigFile. A headless test proves that saving and loading give back the same values.

Time: 45 to 75 minutes.

Files (the finished project is in `projects/ch17-saving-loading/`):

```
ch17-saving-loading/
├── project.godot
├── icon.svg
├── save_manager.gd     autoload "SaveManager" (settings + progress)
├── save_bundle.gd      class SaveBundle (the Resource save, used by the test)
├── pip.gd              Pip, drawn with code, moves with the arrow keys
├── main.gd / main.tscn the play screen
├── settings.gd / settings.tscn
└── tests/test_save.gd  headless round-trip test
```

Step 1: Create the project and the autoload

1. Create a project `PipRemembers`. Set the window size to 640 x 360 and Stretch Mode `canvas_items`.
2. Create `save_manager.gd` with the full script below, then register it as an autoload named `SaveManager` (Project Settings > Globals > Autoload).

```gdscript
# save_manager.gd  (autoload "SaveManager"; extends Node)
extends Node
## Autoload "SaveManager": settings (ConfigFile) and game progress (JSON).

signal settings_changed

const SAVE_VERSION: int = 1

# Paths are variables (not constants) so a test can point them somewhere else.
var settings_path: String = "user://settings.cfg"
var save_path: String = "user://save.json"

# Settings (kept between runs, shared by every save)
var player_name: String = "Pip"
var volume: float = 0.8  # 0.0 (silent) to 1.0 (full)
var fullscreen: bool = false

# Progress (the game itself)
var gold: int = 0
var inventory: Dictionary = {}  # item id (String) -> how many (int)
var pip_position: Vector2 = Vector2(320, 180)
var play_time: float = 0.0  # seconds


func _ready() -> void:
    load_settings()
    apply_settings()


func _process(delta: float) -> void:
    play_time += delta


# ---------- Settings: ConfigFile ----------

func save_settings() -> Error:
    var config := ConfigFile.new()
    config.set_value("player", "name", player_name)
```

```
    config.set_value("audio", "volume", volume)
    config.set_value("video", "fullscreen", fullscreen)
    return config.save(settings_path)


func load_settings() -> Error:
    var config := ConfigFile.new()
    var err: Error = config.load(settings_path)
    if err != OK:
        return err  # first run, or no file: keep the defaults
    player_name = str(config.get_value("player", "name", player_name))
    volume = clampf(float(config.get_value("audio", "volume", volume)), 0.0, 1.0)
    fullscreen = bool(config.get_value("video", "fullscreen", fullscreen))
    return OK


func reset_settings() -> void:
    player_name = "Pip"
    volume = 0.8
    fullscreen = false
    apply_settings()


func apply_settings() -> void:
    AudioServer.set_bus_volume_db(0, linear_to_db(volume))
    if fullscreen:
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_FULLSCREEN)
    else:
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_WINDOWED)
    settings_changed.emit()


# ---------- Progress: JSON ----------

func to_dict() -> Dictionary:
    return {
        "version": SAVE_VERSION,
        "gold": gold,
        "inventory": inventory,
        "position": [pip_position.x, pip_position.y],  # JSON has no Vector2
        "play_time": play_time,
    }


# Returns true if the data was valid and has been applied.
# Nothing is changed unless EVERYTHING checks out.
func from_dict(data: Dictionary) -> bool:
    if int(data.get("version", 0)) != SAVE_VERSION:
        return false
    var new_gold: Variant = data.get("gold")
    var new_inventory: Variant = data.get("inventory")
    var new_position: Variant = data.get("position")
    var new_time: Variant = data.get("play_time")
    if not _is_number(new_gold) or not _is_number(new_time):
        return false
    if typeof(new_inventory) != TYPE_DICTIONARY:
        return false
    if typeof(new_position) != TYPE_ARRAY or new_position.size() != 2:
        return false
    if not _is_number(new_position[0]) or not _is_number(new_position[1]):
        return false
    var clean_inventory: Dictionary = {}
    for key in new_inventory:
        if not _is_number(new_inventory[key]):
            return false
        clean_inventory[str(key)] = int(new_inventory[key])  # JSON numbers come back as floats
    gold = int(new_gold)
    inventory = clean_inventory
    pip_position = Vector2(float(new_position[0]), float(new_position[1]))
    play_time = float(new_time)
    return true
```

```
func _is_number(value: Variant) -> bool:
    return typeof(value) == TYPE_INT or typeof(value) == TYPE_FLOAT


func has_save() -> bool:
    return FileAccess.file_exists(save_path)


func save_game() -> Error:
    var text: String = JSON.stringify(to_dict(), "\t")
    # Write to a temporary file first, then rename. If the game crashes
    # half way through writing, the old save is still safe.
    var temp_path: String = save_path + ".tmp"
    var file := FileAccess.open(temp_path, FileAccess.WRITE)
    if file == null:
        return FileAccess.get_open_error()
    file.store_string(text)
    file.close()
    var dir := DirAccess.open(save_path.get_base_dir())
    if dir == null:
        return DirAccess.get_open_error()
    return dir.rename(temp_path.get_file(), save_path.get_file())


func load_game() -> bool:
    if not has_save():
        return false
    var text: String = FileAccess.get_file_as_string(save_path)
    var json := JSON.new()
    if json.parse(text) != OK:
        push_warning("Save file is damaged (line %d): %s" % [json.get_error_line(),
json.get_error_message()])
        return false
    if typeof(json.data) != TYPE_DICTIONARY:
        return false
    return from_dict(json.data)


func delete_save() -> Error:
    var dir := DirAccess.open(save_path.get_base_dir())
    if dir == null:
        return DirAccess.get_open_error()
    return dir.remove(save_path.get_file())
```

The `_process` function adds up play time; because the autoload lives for the whole run, the time keeps counting through scene changes. Note that the progress variables (`gold`, `pip_position`...) are plain variables in the autoload, exactly like the game state of Chapter 16.

Step 2: Pip

Make `pip.gd`. It will be attached to a `Node2D` in the main scene. It reads the starting position from the SaveManager and writes its position back every frame, so the SaveManager always knows where Pip is.

```
# pip.gd  (attach to a Node2D)
extends Node2D
## Pip, drawn with code. Move with the arrow keys. Position lives in SaveManager.

const SPEED: float = 180.0
const BOUNDS: Rect2 = Rect2(20, 60, 600, 230)


func _ready() -> void:
    position = SaveManager.pip_position


func _process(delta: float) -> void:
```

```
    var direction: Vector2 = Input.get_vector("ui_left", "ui_right", "ui_up", "ui_down")
    position += direction * SPEED * delta
    position = position.clamp(BOUNDS.position, BOUNDS.end)
    SaveManager.pip_position = position


func _draw() -> void:
    draw_circle(Vector2.ZERO, 18.0, Color(0.49, 0.78, 0.89))
    draw_circle(Vector2(-6, -4), 3.0, Color(0.13, 0.15, 0.2))
    draw_circle(Vector2(6, -4), 3.0, Color(0.13, 0.15, 0.2))
    draw_arc(Vector2(0, 2), 8.0, 0.4, PI - 0.4, 12, Color(0.13, 0.15, 0.2), 2.0)
```

`ui_left`, `ui_right`, `ui_up`, and `ui_down` are built-in input actions that already respond to the arrow keys (and a gamepad's D-pad and left stick), so no Input Map editing is needed. `Input.get_vector` combines four actions into one direction (Chapter 7), and `clamp` keeps Pip inside the rectangle.

Step 3: The main scene

```
Main (Control)                main.gd, Layout: Full Rect
├── Pip (Node2D)              pip.gd, Position (320, 180)
├── Info (Label)              unique name %Info, top-left
├── Where (Label)             unique name %Where, small grey text under Info
├── Status (Label)            unique name %Status, centered above the buttons
└── Buttons (HBoxContainer)   along the bottom, Alignment Center
    ├── CoinButton (Button)   unique name %CoinButton, "Collect coin"
    ├── SaveButton (Button)   unique name %SaveButton, "Save"
    ├── LoadButton (Button)   unique name %LoadButton, "Load"
    ├── DeleteButton (Button) unique name %DeleteButton, "Delete save"
    └── SettingsButton (Button) unique name %SettingsButton, "Settings"
```

```
# main.gd  (attach to Main, a Control)
extends Control
## Main screen. Scene tree is in main.tscn.

@onready var info: Label = %Info
@onready var status: Label = %Status
@onready var pip: Node2D = $Pip


func _ready() -> void:
    %CoinButton.pressed.connect(_on_coin_pressed)
    %SaveButton.pressed.connect(_on_save_pressed)
    %LoadButton.pressed.connect(_on_load_pressed)
    %DeleteButton.pressed.connect(_on_delete_pressed)
    %SettingsButton.pressed.connect(_on_settings_pressed)
    %Where.text = "Files live in: " + ProjectSettings.globalize_path("user://")
    status.text = "A save file exists. Press Load." if SaveManager.has_save() else "No save file yet."


func _process(_delta: float) -> void:
    info.text = "Hello, %s!   Gold: %d   Time played: %d s" % [
        SaveManager.player_name, SaveManager.gold, int(SaveManager.play_time)]


func _on_coin_pressed() -> void:
    SaveManager.gold += 1


func _on_save_pressed() -> void:
    var err: Error = SaveManager.save_game()
    status.text = "Saved." if err == OK else "Could not save: " + error_string(err)


func _on_load_pressed() -> void:
    if SaveManager.load_game():
        pip.position = SaveManager.pip_position
        status.text = "Loaded."
```

```
    else:
        status.text = "Nothing to load (or the file is damaged)."


func _on_delete_pressed() -> void:
    var err: Error = SaveManager.delete_save()
    status.text = "Save deleted." if err == OK else "No save to delete."


func _on_settings_pressed() -> void:
    get_tree().change_scene_to_file("res://settings.tscn")
```

(The `status.text = "..." if condition else "..."` form is GDScript's one-line if/else, called a *ternary expression*.)

Notice how little the scene knows. When the player presses Load, the SaveManager fills its variables; the main scene then moves Pip to the loaded position. The label rebuilds itself every frame from the SaveManager, so gold and time update with no extra work.

Step 4: The settings scene

Build `settings.tscn` as shown in section 17.6, attach `settings.gd`, and make sure the main scene's Settings button leads there. Set the Main Scene (Project Settings > Application > Run) to `res://main.tscn`.

Step 5: Run and test by hand

1. Press **F5**. Move Pip with the arrow keys. Click **Collect coin** five times.
2. Click **Save**. The status line says "Saved." Click **Project > Open User Data Folder**; you find `save.json`. Open it in a text editor and read it.
3. Move Pip somewhere else, collect two more coins, then click **Load**. Pip jumps back and the gold drops to 5.
4. Close the game, run it again, and the status line says "A save file exists." Click **Load**: everything is restored.
5. Open **Settings**. Drag the slider (the percentage changes), tick Fullscreen (the window changes), change the name to `Mallow`, and click **Save and go back**. The greeting now says "Hello, Mallow!". Close and reopen the game: the name, volume, and fullscreen are remembered, because `_ready()` of the autoload loaded `settings.cfg`.
6. Break the save on purpose: open `save.json`, delete the last curly brace `}`, save the file, and press **Load**. The game shows "Nothing to load (or the file is damaged)." and the Output panel shows a warning with the line number. The gold did not change.

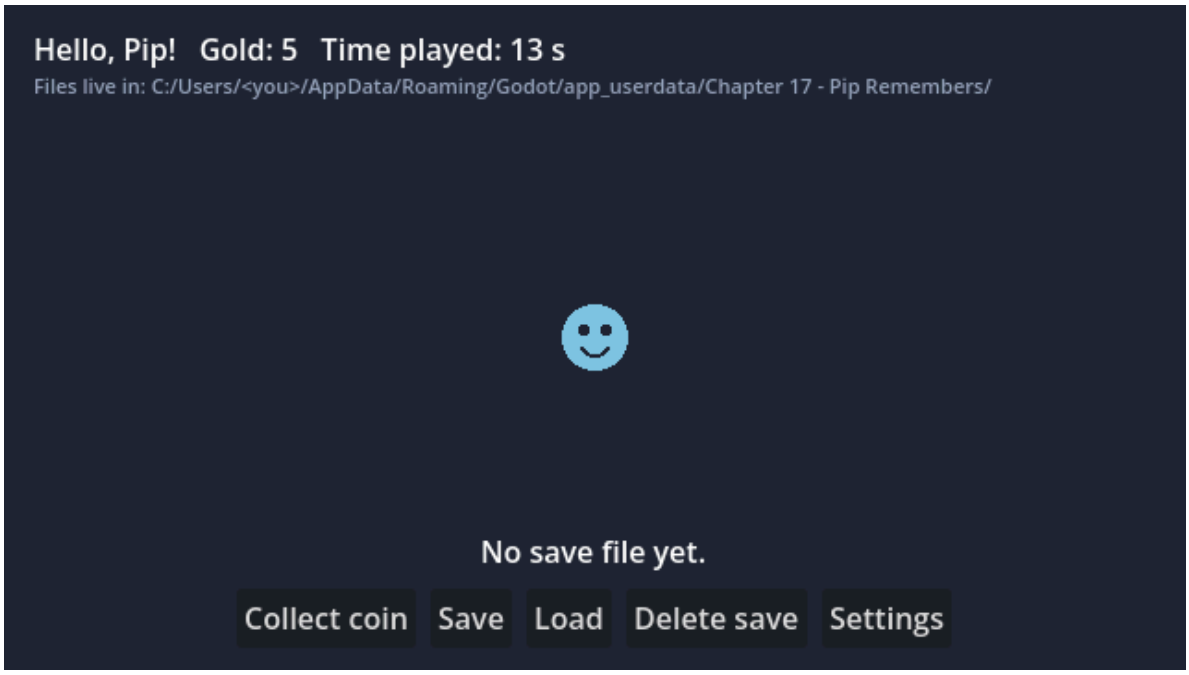


Figure 17.5: The play screen of Pip Remembers. The path line is shortened to a neutral example; yours shows your own user folder.

Step 6: The headless test

Manual testing is slow and easy to forget. The project includes `tests/test_save.gd`, a script that runs without a window. It points the SaveManager at test files (so your real save is untouched), saves, wipes the variables, loads, and compares:

```
# tests/test_save.gd  (a SceneTree script; fragment shown, full file in the project)
extends SceneTree

var _failures: int = 0


func check(condition: bool, label: String) -> void:
    if condition:
        print("  ok    ", label)
    else:
        print("  FAIL  ", label)
        _failures += 1


func _process(_delta: float) -> bool:
    var sm: Node = root.get_node("SaveManager")
    sm.settings_path = "user://test_settings.cfg"
    sm.save_path = "user://test_save.json"

    sm.gold = 42
    sm.inventory = {"coin": 3, "gem": 1}
    sm.pip_position = Vector2(123.5, 99.25)
    check(sm.save_game() == OK, "game saved")
    sm.gold = 0
    sm.inventory = {}
    sm.pip_position = Vector2.ZERO
    check(sm.load_game(), "game loaded")
    check(sm.gold == 42 and typeof(sm.gold) == TYPE_INT, "gold is back as an int")
    check(sm.inventory == {"coin": 3, "gem": 1}, "inventory restored")
    check(sm.pip_position == Vector2(123.5, 99.25), "position restored")

    print("TEST ", "PASSED" if _failures == 0 else "FAILED (%d)" % _failures)
```

```
    quit(0 if _failures == 0 else 1)
    return true
```

Run the full version from a terminal inside the project folder:

```
godot --headless --path . -s tests/test_save.gd
```

You will see a line per check and `TEST PASSED` at the end. The full test also covers: a second save replacing the first, a damaged file, a file with the wrong type for `gold`, an unknown version, a missing file, and a `SaveBundle` resource round trip.

> **Why it works:** The script extends `SceneTree`, which lets Godot run it as the whole program instead of a normal scene (`-s` means "run this script"). It does its work in `_process` because autoloads finish their `_ready()` only after the tree starts running; in `_initialize()` they would not be ready yet. Looking up `root.get_node("SaveManager")` with a name (no leading slash) is the dependable way to reach an autoload from such a script.

Expected result

- Pip moves; the coin button raises the gold; the time counter ticks.
- Save writes `save.json` to the user data folder; Load restores gold, inventory, position, and time exactly.
- Settings persist between runs in `settings.cfg`.
- A damaged or wrong-typed save is refused with a message, and nothing in the game changes.
- The headless test prints `TEST PASSED`.

Chapter summary

- Variables live in memory and vanish when the game closes. To keep data you serialize it and write it to a file in `user://`.
- `res://` is read-only in an exported game. `user://` is the writable per-user folder; **Project > Open User Data Folder** takes you there.
- `ConfigFile` stores `[section] key=value` settings. `get_value` with a default handles first runs and older files. Always check the `Error` that `save` and `load` return.
- Progress fits JSON: build a Dictionary, `JSON.stringify` it, write it with `FileAccess`. Read it with `FileAccess.get_file_as_string` and `JSON.new().parse`.
- JSON has no `Vector2`, and returns every number as a float. Convert explicitly (`int(...)`, arrays for vectors).
- Never trust a file. Validate every type first and apply the data only if everything passes; store a `version`; write through a temporary file and rename it so a crash cannot destroy the old save.
- Custom Resources are the least code but the least safe to load from untrusted files. Choose: ConfigFile for settings, JSON for progress.
- A settings screen shows the current values, applies changes at once, and saves when the player leaves.

Review questions

1. Why does a game lose its variables when it closes, and what must you do to keep them? (Short answer)
2. Why must saved data go in `user://` and not `res://`? (Short answer)
3. True or false: `user://` is the same folder on every operating system.

4. What does the third argument of `ConfigFile.get_value("audio", "volume", 0.8)` do, and in which two situations does it matter? (Short answer)
5. `config.save("user://settings.cfg")` returns a value. What is it, and what does the value `OK` mean? (Short answer)

6. You save `{"gold": 42}` as JSON and load it again. What is the type of `data["gold"]`? (Multiple choice) a) `int` b) `float` c) `String` d) `Variant` only, with no definite type

7. Which of these cannot be stored in JSON without converting it first? (Multiple choice) a) an `int` b) a `Dictionary` c) a `Vector2` d) a `bool`

8. In `from_dict`, why are all the checks done before any game variable is changed? (Short answer)

9. Explain the "atomic write" trick in your own words, and say what problem it prevents. (Short answer)
10. True or false: loading a custom `.tres` Resource from a file the player can edit is exactly as safe as loading a JSON file.

Exercises

1. **Practice (easy):** Add a `high_score` key to the JSON save (an int). Add it to `to_dict`, validate it in `from_dict`, and show it in the info label. Test with the headless test by adding a check.
2. **Practice (easy):** Add a text field to the settings file: `[player] favorite_color="blue"`. Add a `LineEdit` for it in the settings screen and make sure it survives a restart.
3. **Apply (medium):** Add three save slots. Change `save_path` to be built from a slot number (`"user://save_%d.json" % slot`), add an `OptionButton` to choose the slot on the main screen, and show on each slot's entry whether a file exists.
4. **Apply (medium):** Add autosave: a `Timer` in `main.tscn` (wait time 10 seconds, autostart on) that calls `SaveManager.save_game()` and shows "Autosaved." in the status label. Think about what should happen if the save fails.
5. **Challenge (hard):** Implement save **migration**. Make version 2 of the save add a `"lives"` key (default 3). Write `from_dict` so that a version-1 file (no `lives`) still loads, filling in 3, while a version-2 file loads `lives` normally. Add two headless test checks, one for each version.

Common mistakes

- **Symptom:** saving works in the editor but not in the exported game. **Cause:** the path starts with `res://`. **Fix:** change it to `user://`.
- **Symptom:** the game crashes with "Invalid get index 'gold' (on base: 'Dictionary')" while loading. **Cause:** the file lacks a key and the code used `data["gold"]` without checking. **Fix:** validate with `data.get("gold")` and type checks; refuse the file on failure.
- **Symptom:** after loading, numbers print as `42.0` or an item count cannot be used as an array index. **Cause:** JSON returns every number as a float. **Fix:** convert with `int(...)` when loading.
- **Symptom:** `file.store_string(...)` fails with "Invalid call. Nonexistent function on a null instance". **Cause:** `FileAccess.open` returned `null` (bad path or no permission) and the result was not checked. **Fix:** test `if file == null:` and report `FileAccess.get_open_error()`.
- **Symptom:** settings reset to defaults on every start. **Cause:** `load_settings()` is never called (or it runs before the file has been saved), or the project name changed, which changed the `user://` folder. **Fix:** call `load_settings()` in the autoload's `_ready()`, and keep the project name stable (or set a custom user dir).

- **Symptom:** the slider handler runs while the screen is opening and the volume jumps. **Cause:** the signal was connected before the starting value was set. **Fix:** set the values first, connect afterwards.

Further reading

- Saving games: https://docs.godotengine.org/en/stable/tutorials/io/saving_games.html
- File paths in Godot (`res://` and `user://`): https://docs.godotengine.org/en/stable/tutorials/io/data_paths.html
- ConfigFile: https://docs.godotengine.org/en/stable/classes/class_configfile.html
- FileAccess: https://docs.godotengine.org/en/stable/classes/class_fileaccess.html
- JSON: https://docs.godotengine.org/en/stable/classes/class_json.html

Chapter 18: Project: Pip's Coin Run, a Complete Game

Learning objectives

By the end of this chapter you will be able to:

- Plan a small game as a list of screens, a flow between them, and a short list of shared data.
- Combine scenes, signals, autoloads, and saving from the earlier chapters into one working game.
- Build a title screen, a play screen, a pause menu, and a game over screen, and connect them with scene changes.
- Pause a game correctly using `get_tree().paused` and node process modes, while the pause menu keeps working.
- Spawn coins and enemies from timers, and make a game gradually harder.
- Store the high score in a file, and show a "new best" message when the player beats it.
- Test the game's logic with a headless script and find your way around a project of about fifteen files.

Key terms

- **Game loop (design sense)**: the repeating cycle the player lives in, here: move, collect, dodge, score, try again.
- **Screen flow**: a map showing which screen leads to which other screen and what triggers each move.
- **Pause**: freezing the game's processing and physics without closing it. In Godot, `get_tree().paused = true`.
- **Process mode**: a per-node setting that decides whether the node keeps running while the tree is paused.
- **Spawner**: code (often a Timer) that creates new game objects while the game runs.
- **Difficulty curve**: how the challenge grows as time passes or as the player improves.
- **Invulnerability frames**: a short period after a hit during which the player cannot be hit again.
- **High score**: the best score ever reached, kept in a file so it survives closing the game.
- **Placeholder art**: simple shapes drawn with code that stand in for finished art.
- **Scope (of a project)**: how much a project tries to do. Keeping scope small is how games get finished.

18.1 What we are building

For seventeen chapters you have collected skills one at a time. This chapter puts them together into one finished, small game, the kind you could show a friend. It is called **Pip's Coin Run**, and it grows out of Coin Dash from Chapter 9.

The pitch. Pip is trapped in a walled arena. Gold coins appear at random. Slimes crawl in from the edges and chase Pip. Collect coins to score points; each slime that touches you costs one of your three lives. When the lives run out, the run is over. Beat your best score and the game remembers it, even after you close it.

What the player sees.

1. A **title screen** with the game name, the best score, and Play, Mute, and Quit.
2. The **play screen**: the arena, Pip, coins, slimes, and a bar showing score, lives, and best.
3. A **pause menu** over the play screen (Esc or P): Resume, Mute, Quit to title.

4. A **game over screen**: your score, the best score, a "NEW BEST SCORE!" banner when earned, Play again, and Back to title.

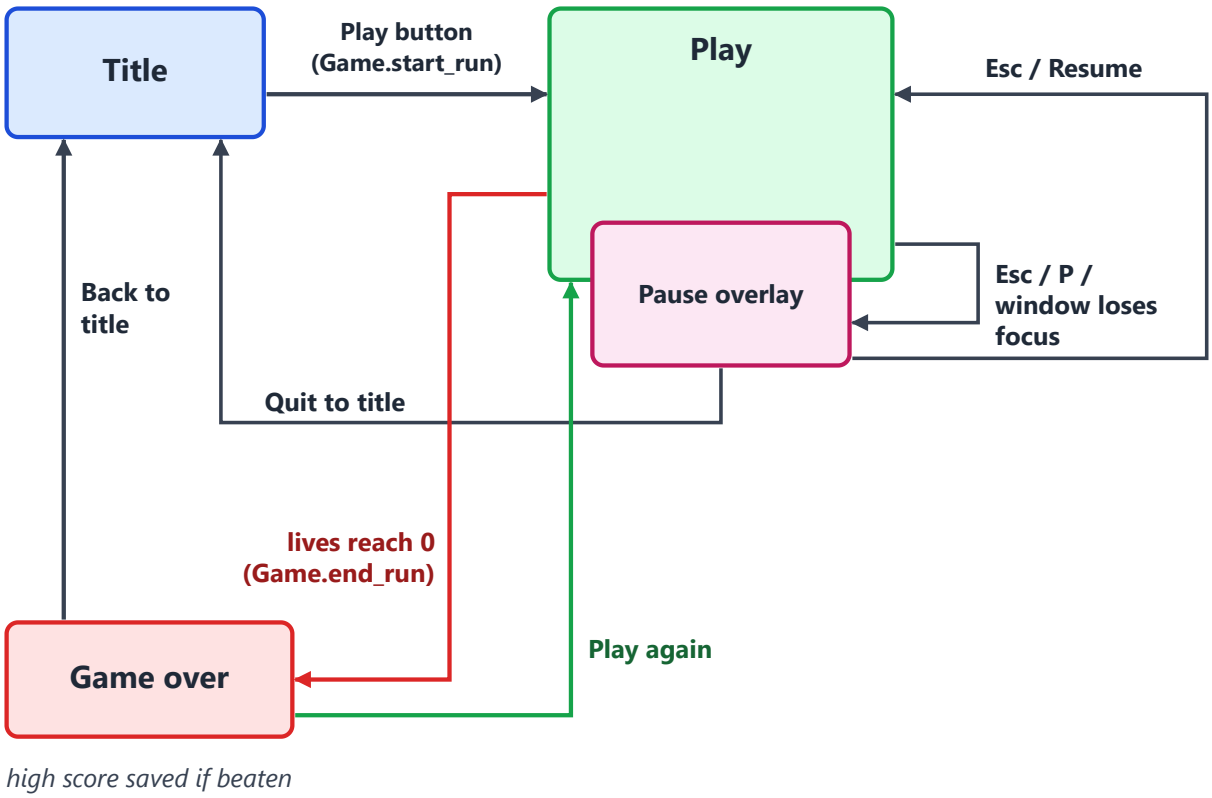


Figure 18.1: The screen flow. Before writing code, draw this map; every arrow becomes one line of code.

Plan on paper first

A game of this size has about a dozen moving parts. If you start by typing, you will quickly lose track. Professional developers start with three small lists, and so should you.

List 1: the screens and the arrows between them (Figure 18.1). Each arrow is a scene change or an overlay.

List 2: the rules.

- Pip moves in four directions at a fixed speed and cannot leave the arena.
- A coin gives 10 points. Coins vanish by themselves after 8 seconds (blinking at the end), and at most 6 exist at once.
- A slime moves straight toward Pip. Touching Pip costs 1 life and the slime pops.
- After being hit, Pip blinks and cannot be hit again for 1.2 seconds.
- Slimes appear on the edge of the arena, away from Pip. The longer the run lasts, the faster they come and the faster they crawl.
- At 0 lives the run ends. If the score beats the best score, the new best is saved.

List 3: the shared data, and who owns it.

Data	Who owns it	Why
Score, lives, "is a run in progress", "was it a record"	`Game` autoload	Needed by the play screen, the HUD, and the game over screen.
High score, volume, mute	`Save` autoload (a ConfigFile)	Must survive closing the game.
Position of Pip, slime positions, timers	The play scene's nodes	Only meaningful while playing; thrown away with the scene.
Beeps	`Sfx` autoload	Any scene may make a sound, even while paused.

This is exactly the reasoning of Chapter 16: *does it need to outlive the scene?* Only the first, second, and fourth rows do.

> **Note:** "Scope" is the most important word in game making. Every feature you add costs time, and unfinished games teach you less than finished small ones. This design is deliberately small. When you are done, the Exercises show where to grow it.

18.2 The project map

The finished project is in `projects/ch18-pips-coin-run/`. Here is every file and what it is for. You will recognize most ideas from earlier chapters.

```
ch18-pips-coin-run/
├── project.godot        settings, autoloads, input actions
├── icon.svg
│
├── save.gd              autoload "Save":  high score, volume, mute (ConfigFile)
├── game.gd              autoload "Game":  score, lives, run state, scene changes
├── sfx.gd               autoload "Sfx":   generated beeps
│
├── pip_face.gd          draws Pip's face (used by the player AND the title)
├── player.gd / player.tscn   CharacterBody2D, movement, being hurt
├── coin.gd / coin.tscn       Area2D, +10 points
├── slime.gd / slime.tscn     Area2D, chases Pip
│
├── play_scene.gd / game.tscn   the play screen: arena, spawning
├── hud.gd / hud.tscn           CanvasLayer: score, lives, best
├── pause_menu.gd / pause_menu.tscn   CanvasLayer overlay
│
├── title.gd / title.tscn         title screen (main scene)
├── game_over.gd / game_over.tscn game over screen
└── tests/test_run.gd             headless check of the logic
```

One naming note. The autoload script is `game.gd` (autoload name `Game`), and the play screen's scene is `game.tscn`. The play screen's *script* therefore has a different name, `play_scene.gd`. A script and a scene may share a base name, but two scripts both called `game.gd` in one folder would be impossible.

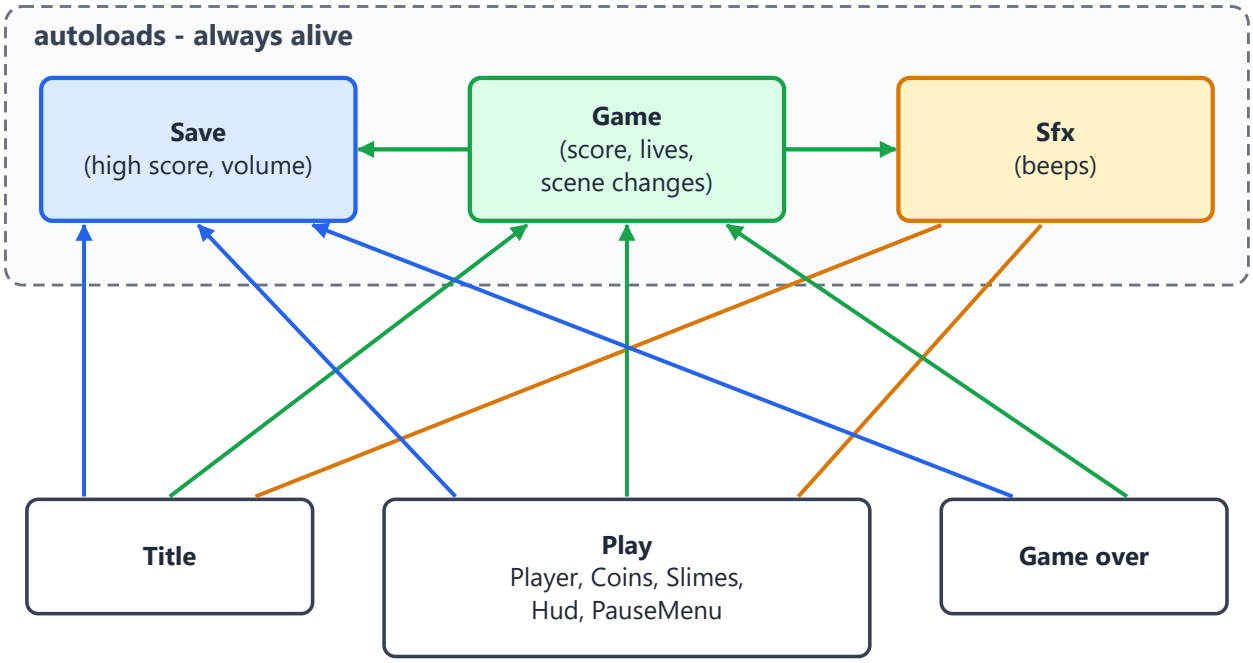


Figure 18.2: Who talks to whom. Scenes use the autoloads; the autoloads never reach into a scene.

18.3 The autoloads: Save, Game, Sfx

We start with the parts that hold the game together, because everything else uses them.

Save: the high score and the sound settings

This is Chapter 17 in miniature. A `ConfigFile` holds three values. The file is read when the game starts (`_ready`), and written when a new record is set or the mute setting is changed.

```
# save.gd  (Save, an autoload that extends Node)
extends Node
## Autoload "Save": the high score and the sound settings, kept in a ConfigFile.

var path: String = "user://pips_coin_run.cfg"  # a variable so a test can change it

var high_score: int = 0
var volume: float = 0.8
var muted: bool = false


func _ready() -> void:
    load_data()
    apply_audio()


func load_data() -> void:
    var config := ConfigFile.new()
    if config.load(path) != OK:
        return  # first run: keep the defaults
    high_score = maxi(0, int(config.get_value("scores", "high_score", 0)))
    volume = clampf(float(config.get_value("audio", "volume", volume)), 0.0, 1.0)
    muted = bool(config.get_value("audio", "muted", false))


func save_data() -> Error:
```

```
    var config := ConfigFile.new()
    config.set_value("scores", "high_score", high_score)
    config.set_value("audio", "volume", volume)
    config.set_value("audio", "muted", muted)
    return config.save(path)


# Call this when a run ends. Returns true if it is a new record.
func submit_score(score: int) -> bool:
    if score <= high_score:
        return false
    high_score = score
    save_data()
    return true


func set_muted(value: bool) -> void:
    muted = value
    apply_audio()
    save_data()


func apply_audio() -> void:
    AudioServer.set_bus_mute(0, muted)
    AudioServer.set_bus_volume_db(0, linear_to_db(volume))
```

Read it with the Chapter 17 habits in mind: `load_data` returns quietly when there is no file (first run), converts values with `int(...)` and `float(...)`, and clamps the volume. `submit_score` is the only way the high score changes, and it only saves when there is a real record, so the file is not rewritten after every run.

Game: the state of a run, and the way between screens

The `Game` autoload is the director. It knows whether a run is in progress and owns the score and the lives. It announces changes with signals (Chapter 16), and it is the only place that changes scenes, so all the screen-flow arrows of Figure 18.1 live in one short file.

```
# game.gd  (Game, an autoload that extends Node)
extends Node
## Autoload "Game": the state of the current run, and moving between screens.

signal score_changed(new_score: int)
signal lives_changed(new_lives: int)

const START_LIVES: int = 3
const TITLE_SCENE: String = "res://title.tscn"
const PLAY_SCENE: String = "res://game.tscn"
const OVER_SCENE: String = "res://game_over.tscn"

var score: int = 0
var lives: int = START_LIVES
var running: bool = false  # true while a run is being played
var new_best: bool = false  # did the last run beat the record?


func start_run() -> void:
    score = 0
    lives = START_LIVES
    new_best = false
    running = true
    get_tree().paused = false
    get_tree().change_scene_to_file(PLAY_SCENE)


func add_score(points: int) -> void:
    if not running:
        return
    score += points
```

```
    score_changed.emit(score)


func lose_life() -> void:
    if not running:
        return
    lives -= 1
    lives_changed.emit(lives)
    if lives <= 0:
        end_run()


func end_run() -> void:
    running = false
    new_best = Save.submit_score(score)
    Sfx.play("over")
    get_tree().change_scene_to_file(OVER_SCENE)


func go_to_title() -> void:
    running = false
    get_tree().paused = false  # never carry a pause into another screen
    get_tree().change_scene_to_file(TITLE_SCENE)
```

Points to notice:

- **`start_run()` resets everything**, then changes scene. This is the "new game gives old gold" bug from Chapter 16, prevented by design. Both the title's Play button and the game over screen's Play again button call this one function.
- **Every function that changes the state checks `running`.** After the last life is lost, a coin could still be collected in the same frame. Without the guard, the score would change after the run had ended.
- **`end_run()` asks `Save` whether it is a record** and remembers the answer in `new_best`, so the game over screen can read it later.
- **`go_to_title()` un-pauses first.** The pause menu's "Quit to title" button calls this. If we forgot, the title screen would load frozen: the tree's `paused` flag is global and survives a scene change.
- **Signals carry the new value** (`score_changed(new_score)`), so a listener never has to ask.

Sfx: sounds made from math

The rule of this book is no external art or sound, so Pip's game makes its beeps from sine waves. Chapter 11 introduced `AudioStreamGenerator`; here is the whole machine in one small autoload.

```
# sfx.gd  (Sfx, an autoload that extends Node)
extends Node
## Autoload "Sfx": tiny beeps made from math, so we need no sound files.
## Usage: Sfx.play("coin")

const MIX_RATE: float = 22050.0
const SOUNDS: Dictionary = {
    "coin": [[880.0, 0.06], [1320.0, 0.08]],
    "hurt": [[220.0, 0.12], [150.0, 0.18]],
    "over": [[440.0, 0.15], [330.0, 0.15], [220.0, 0.3]],
    "click": [[600.0, 0.04]],
}

var _playback: AudioStreamGeneratorPlayback
var _queue: Array = []  # each entry: [frequency, frames_left]
var _phase: float = 0.0


func _ready() -> void:
    process_mode = Node.PROCESS_MODE_ALWAYS  # keep working while the game is paused
    var stream := AudioStreamGenerator.new()
```

```
    stream.mix_rate = MIX_RATE
    stream.buffer_length = 0.25
    var player := AudioStreamPlayer.new()
    player.stream = stream
    add_child(player)
    player.play()
    _playback = player.get_stream_playback() as AudioStreamGeneratorPlayback


func play(sound_name: String) -> void:
    if _queue.size() > 12:
        return
    for note in SOUNDS.get(sound_name, []):
        _queue.append([note[0], int(note[1] * MIX_RATE)])


func _process(_delta: float) -> void:
    if _playback == null:
        return
    var room: int = _playback.get_frames_available()
    while room > 0 and not _queue.is_empty():
        var note: Array = _queue[0]
        var frequency: float = note[0]
        var frames_left: int = note[1]
        var sample: float = sin(_phase * TAU) * 0.2
        _playback.push_frame(Vector2(sample, sample))
        _phase = fmod(_phase + frequency / MIX_RATE, 1.0)
        note[1] = frames_left - 1
        room -= 1
        if frames_left <= 1:
            _queue.pop_front()


func _exit_tree() -> void:
    _playback = null  # let go of the playback object so it can be cleaned up
```

How it works, in plain words:

1. `_ready()` builds an `AudioStreamPlayer` whose stream is an `AudioStreamGenerator`, a stream with no file behind it. We fill it ourselves.
2. `play("coin")` looks up a short list of notes `[frequency, seconds]` and puts them in a queue.
3. Every frame, `_process` pushes as many audio **frames** (one left/right sample pair) as the generator has room for, calculating `sin(_phase * TAU) * 0.2` each time. A sine wave is a pure tone; the number `0.2` keeps it quiet; the frequency (880 for a high beep, 150 for a low thud) decides the pitch.
4. `process_mode = Node.PROCESS_MODE_ALWAYS` is the new idea. By default nodes stop when the game is paused. A sound player that fell silent whenever you paused would be wrong for the "click" of the pause menu. We study process modes properly in section 18.6.
5. `_exit_tree()` lets go of the playback object when the game closes, which is tidy housekeeping.

> **Note:** The beeps are simple on purpose. If you later add real sound files (free ones from Kenney.nl, itch.io, or OpenGameArt), only this one autoload needs to change: the rest of the game calls `Sfx.play("coin")` and does not care how the sound is made. That is the benefit of putting a feature behind one small doorway.

18.4 Pip, coins, and slimes

Three scenes, each small and self-contained. All are drawn with code, so there are no art files.

Pip's face

One tiny script that draws a face, used in two places (the player and the big Pip on the title screen). `@export var radius` lets each place choose a size.

```
# pip_face.gd  (a Node2D inside the player scene, and a Node2D on the title screen)
extends Node2D
## Pip's look, drawn with code. Used by the player, the title and the game over screen.

@export var radius: float = 18.0


func _draw() -> void:
    var dark := Color(0.13, 0.15, 0.2)
    draw_circle(Vector2.ZERO, radius, Color(0.49, 0.78, 0.89))
    var s: float = radius / 18.0
    draw_circle(Vector2(-6, -4) * s, 3.0 * s, dark)
    draw_circle(Vector2(6, -4) * s, 3.0 * s, dark)
    draw_arc(Vector2(0, 2) * s, 8.0 * s, 0.4, PI - 0.4, 12, dark, 2.0 * s)
```

The player

```
Player (CharacterBody2D)   player.gd, in group "player", Motion Mode: Floating
├── CollisionShape2D       CircleShape2D, radius 16
└── Face (Node2D)          pip_face.gd
```

Motion Mode: Floating is a `CharacterBody2D` property meant for top-down games with no gravity or floor. The default, Grounded, assumes a platformer. In the scene file it is stored as `motion_mode = 1`.

To add the node to the **group** `player`: select the Player node, open the **Groups** tab (next to the Inspector), type `player`, and click Add. Other scripts then find Pip with `get_tree().get_first_node_in_group("player")`, or test an unknown body with `body.is_in_group("player")`.

```
# player.gd  (Player, attached to the CharacterBody2D)
extends CharacterBody2D
## Pip. Moves with move_left / move_right / move_up / move_down.

const SPEED: float = 280.0
const RADIUS: float = 18.0
const HURT_TIME: float = 1.2  # seconds of "can't be hurt" after a hit

var arena: Rect2 = Rect2(40, 60, 880, 440)  # game.gd sets the real one
var _invulnerable: float = 0.0


func _physics_process(delta: float) -> void:
    var direction: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
    velocity = direction * SPEED
    move_and_slide()
    var margin := Vector2(RADIUS, RADIUS)
    position = position.clamp(arena.position + margin, arena.end - margin)
    _update_blink(delta)


func _update_blink(delta: float) -> void:
    if _invulnerable > 0.0:
        _invulnerable -= delta
        var visible_phase: bool = int(_invulnerable * 10.0) % 2 == 0
        modulate.a = 1.0 if visible_phase else 0.3
    else:
        modulate.a = 1.0


# Returns true if the hit counted.
```

```
func hurt() -> bool:
    if _invulnerable > 0.0:
        return false
    _invulnerable = HURT_TIME
    Sfx.play("hurt")
    Game.lose_life()
    return true
```

What to notice:

- **Input actions, not keys.** The script asks about `move_left`, `move_right`, and so on. The Input Map (section 18.8) decides which keys those are, so both arrow keys and WASD work, and a gamepad could be added without touching the script.
- **`velocity` then `move_and_slide()`.** This is the pattern from Chapters 7 and 8.
- **`clamp`** keeps Pip inside the arena. The arena rectangle is passed in by the play scene (`player.arena = ARENA`), so the player does not hard-code a map size.
- **`hurt()` returns a bool.** It returns `false` if Pip is still invulnerable. The slime uses that answer: it only pops if the hit counted. The blink is a visual cue: `modulate.a` (the alpha, or transparency) alternates between 0.3 and 1.0 about five times a second.

The coin

```
Coin (Area2D)               coin.gd
└── CollisionShape2D        CircleShape2D, radius 12
```

```
# coin.gd  (Coin, attached to the Area2D)
extends Area2D
## A coin. Pip touches it: +10 points. It vanishes by itself after a while.

const LIFETIME: float = 8.0

var _age: float = 0.0


func _ready() -> void:
    body_entered.connect(_on_body_entered)


func _process(delta: float) -> void:
    _age += delta
    scale.x = absf(cos(_age * 4.0)) * 0.8 + 0.2  # a cheap "spinning" look
    if _age > LIFETIME - 2.0:
        visible = int(_age * 8.0) % 2 == 0  # blink: about to vanish
    if _age >= LIFETIME:
        queue_free()


func _draw() -> void:
    draw_circle(Vector2.ZERO, 12.0, Color(1.0, 0.84, 0.2))
    draw_arc(Vector2.ZERO, 8.0, 0.0, TAU, 20, Color(0.85, 0.6, 0.1), 2.0)


func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        Game.add_score(10)
        Sfx.play("coin")
        queue_free()
```

An **Area2D** detects bodies entering it (Chapter 8). The signal `body_entered(body)` fires when any body touches it, so the script asks "is it Pip?" using the group. The `scale.x` trick squashes the coin sideways with `cos`, so it seems to spin without any animation file. `absf(...) * 0.8 + 0.2` keeps the width from reaching zero (a zero-

width coin would vanish for a moment). In its last two seconds the coin blinks by toggling `visible`, and then removes itself with `queue_free()`.

The slime

```
Slime (Area2D)              slime.gd
└── CollisionShape2D        CircleShape2D, radius 14
```

```gdscript
# slime.gd  (Slime, attached to the Area2D)
extends Area2D
## A slime. It crawls toward Pip. Touching Pip costs a life, and the slime pops.

var speed: float = 80.0  # game.gd sets this when it spawns the slime

var _time: float = 0.0


func _ready() -> void:
    body_entered.connect(_on_body_entered)
    _time = randf() * TAU  # so slimes do not all wobble in step


func _physics_process(delta: float) -> void:
    _time += delta
    scale.y = 1.0 + 0.1 * sin(_time * 8.0)
    var pip := get_tree().get_first_node_in_group("player") as Node2D
    if pip == null:
        return
    position += (pip.position - position).normalized() * speed * delta


func _draw() -> void:
    # A dome: points along the top half of a circle, closed by a flat bottom.
    var points := PackedVector2Array()
    for i in range(0, 13):
        var angle: float = PI + PI * i / 12.0
        points.append(Vector2(cos(angle) * 16.0, sin(angle) * 16.0 + 8.0))
    draw_colored_polygon(points, Color(0.35, 0.8, 0.4))
    draw_circle(Vector2(-5, -2), 3.5, Color.WHITE)
    draw_circle(Vector2(5, -2), 3.5, Color.WHITE)
    draw_circle(Vector2(-5, -2), 1.5, Color(0.1, 0.11, 0.16))
    draw_circle(Vector2(5, -2), 1.5, Color(0.1, 0.11, 0.16))


func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player") and body.hurt():
        queue_free()
```

The slime has almost no brain: each physics frame it takes the direction to Pip, `(pip.position - position).normalized()`, and moves that way at `speed`. `normalized()` shortens a vector to length 1, so speed is not affected by distance (Chapter 6). The dome is a polygon built with a loop: `PI + PI * i / 12.0` sweeps the angle through the top half of a circle, and `cos` / `sin` turn each angle into a point.

`pip == null` is checked before using Pip: the player might not exist for a moment (for instance, during a scene change), and a missing check would crash.

> **Note:** Why do coins and slimes use `Area2D` and not bodies? We want to be *told* about the touch, not to bounce off each other. Areas are for detection; bodies are for blocking. Pip, who needs `move_and_slide`, is the body.

18.5 The play screen

The play scene brings the pieces together.

```
Game (Node2D)                  play_scene.gd
├── Coins (Node2D)             container for spawned coins
├── Slimes (Node2D)            container for spawned slimes
├── Player (instance of player.tscn)   position (480, 280)
├── CoinTimer (Timer)          wait 1.0, Autostart on
├── SlimeTimer (Timer)         wait 3.0, Autostart on
├── Hud (instance of hud.tscn)
└── PauseMenu (instance of pause_menu.tscn)
```

The two empty `Node2D` containers are a tidy habit. Instead of mixing 6 coins and 12 slimes with Pip, the spawner adds them to their own folders, so counting them is one line (`coins.get_child_count()`).

```gdscript
# play_scene.gd  (the root Node2D of game.tscn)
extends Node2D
## The play screen: draws the arena and spawns coins and slimes.

const ARENA: Rect2 = Rect2(40, 60, 880, 440)
const MAX_COINS: int = 6
const MAX_SLIMES: int = 12
const CoinScene: PackedScene = preload("res://coin.tscn")
const SlimeScene: PackedScene = preload("res://slime.tscn")

var _elapsed: float = 0.0

@onready var player: CharacterBody2D = $Player
@onready var coins: Node2D = $Coins
@onready var slimes: Node2D = $Slimes
@onready var coin_timer: Timer = $CoinTimer
@onready var slime_timer: Timer = $SlimeTimer


func _ready() -> void:
    player.arena = ARENA
    player.position = ARENA.get_center()
    coin_timer.timeout.connect(_spawn_coin)
    slime_timer.timeout.connect(_spawn_slime)
    _spawn_coin()


func _process(delta: float) -> void:
    _elapsed += delta
    # The longer you play, the more often slimes arrive.
    slime_timer.wait_time = maxf(0.9, 3.0 - _elapsed * 0.03)


func _draw() -> void:
    draw_rect(ARENA, Color(0.16, 0.18, 0.25))
    draw_rect(ARENA, Color(0.35, 0.4, 0.55), false, 4.0)


func _spawn_coin() -> void:
    if coins.get_child_count() >= MAX_COINS:
        return
    var coin := CoinScene.instantiate() as Area2D
    coin.position = _random_point_away_from_pip(120.0)
    coins.add_child(coin)


func _spawn_slime() -> void:
    if slimes.get_child_count() >= MAX_SLIMES:
        return
    var slime := SlimeScene.instantiate() as Area2D
    slime.speed = minf(70.0 + _elapsed * 1.2, 170.0)
```

```
    slime.position = _random_edge_point()
    for attempt in 8:
        if slime.position.distance_to(player.position) > 220.0:
            break
        slime.position = _random_edge_point()
    slimes.add_child(slime)


func _random_point_away_from_pip(min_distance: float) -> Vector2:
    var point := Vector2.ZERO
    for attempt in 10:
        point = Vector2(
            randf_range(ARENA.position.x + 30.0, ARENA.end.x - 30.0),
            randf_range(ARENA.position.y + 30.0, ARENA.end.y - 30.0))
        if point.distance_to(player.position) >= min_distance:
            break
    return point


func _random_edge_point() -> Vector2:
    match randi() % 4:
        0:
            return Vector2(randf_range(ARENA.position.x, ARENA.end.x), ARENA.position.y)
        1:
            return Vector2(randf_range(ARENA.position.x, ARENA.end.x), ARENA.end.y)
        2:
            return Vector2(ARENA.position.x, randf_range(ARENA.position.y, ARENA.end.y))
        _:
            return Vector2(ARENA.end.x, randf_range(ARENA.position.y, ARENA.end.y))
```

Spawning with timers

Each `Timer` node fires `timeout` repeatedly (Chapter 5). `coin_timer.timeout.connect(_spawn_coin)` calls our function every second. In `_spawn_coin` we first check the cap (`MAX_COINS`), then `instantiate()` the scene, set its position, and `add_child` it to the container. Order matters: set the position **before** `add_child`, so the coin never appears at (0, 0) for a frame.

`preload("res://coin.tscn")` at the top loads the scene file once while the script is compiled. Names starting with a capital letter (`CoinScene`) are the convention for constants that hold loaded scenes.

Placing things fairly

Random placement can be unfair: a slime might spawn on top of Pip, or a coin might appear exactly where Pip stands. Two helper functions make it fair:

- `_random_point_away_from_pip(min_distance)` tries up to 10 random points and stops at the first one far enough from Pip. (`for attempt in 10:` repeats ten times; GDScript lets you loop over an integer.)
- `_random_edge_point()` picks one of four sides of the arena with `match randi() % 4` (a random whole number from 0 to 3) and a random spot along it. The slime spawner then retries up to 8 times for a point at least 220 pixels from Pip.

These "try a few times then accept" loops are a common pattern: they always finish, even if no perfect spot exists.

A difficulty curve

Games feel good when they start easy and get harder. We add the simplest possible curve using a number that grows with time, `_elapsed`:

```
# fragment from play_scene.gd
slime_timer.wait_time = maxf(0.9, 3.0 - _elapsed * 0.03)   # a slime every 3 s, down to every 0.9 s
```

```
slime.speed = minf(70.0 + _elapsed * 1.2, 170.0)          # speed 70, up to 170
```

Read the first line like this: start at 3 seconds between slimes, subtract 0.03 for each second played, but never go below 0.9. After 70 seconds the spawn rate stops getting faster. `maxf` and `minf` are "clamps with one side": `maxf(0.9, x)` means "x, but at least 0.9"; `minf(x, 170)` means "x, but at most 170". Without the limits the game would become impossible, or the timer's `wait_time` would reach zero or go negative, which is an error.

Tuning these numbers is game design. Make the slimes too fast and the game is frustrating; too slow and it is boring. Change one number at a time and play.

The arena

`_draw()` on the root node paints a filled rectangle and a border. Drawing in the root `Node2D` puts the arena *behind* its children (children draw on top of their parent), so coins, slimes, and Pip all appear over it.

18.6 The HUD and the pause menu

The HUD

A HUD (heads-up display) shows information on top of the game. It is a `CanvasLayer`, a node that draws on its own layer and does not move with any camera (Chapter 14).

```
Hud (CanvasLayer)              hud.gd
└── Bar (HBoxContainer)        top, full width, margins 40 left/right, 14 top
    ├── ScoreLabel (Label)     unique name, gold, size 28, expand
    ├── LivesLabel (Label)     unique name, size 28, expand, centered
    └── BestLabel (Label)      unique name, size 28, expand, right-aligned
```

Set each label's **Size Flags > Horizontal** to **Fill + Expand** so the three share the width equally. That is why one is left-aligned, one centered, and one right-aligned.

```
# hud.gd  (Hud, attached to the CanvasLayer)
extends CanvasLayer
## Score, lives and best score along the top of the play screen.

@onready var score_label: Label = %ScoreLabel
@onready var lives_label: Label = %LivesLabel
@onready var best_label: Label = %BestLabel


func _ready() -> void:
    Game.score_changed.connect(_on_score_changed)
    Game.lives_changed.connect(_on_lives_changed)
    _on_score_changed(Game.score)
    _on_lives_changed(Game.lives)
    best_label.text = "Best: %d" % Save.high_score


func _on_score_changed(new_score: int) -> void:
    score_label.text = "Score: %d" % new_score


func _on_lives_changed(new_lives: int) -> void:
    lives_label.text = "Lives: %d" % new_lives
```

It does the same two-step as Chapter 16's gold label: show the current value, then connect the signal. Look at what it does *not* contain: no reference to the player, the slimes, or the play scene. It only knows `Game` and `Save`.

Pausing: the tree's `paused` flag and process modes

Pausing in Godot is a property of the scene tree: `get_tree().paused = true`. When it is on, nodes whose **process mode** says they obey the pause stop receiving `_process`, `_physics_process`, and input callbacks. Physics freezes, Timers stop counting, and Tweens and AnimationPlayers pause too.

Every node has a **Process** > **Mode** property in the Inspector:

Mode	Behavior
Inherit (default)	Do whatever the parent does. A scene's root ends up behaving like Pausable.
Pausable	Runs normally; stops when the tree is paused.
When Paused	Runs *only* while the tree is paused.
Always	Runs regardless of pause.
Disabled	Never runs.

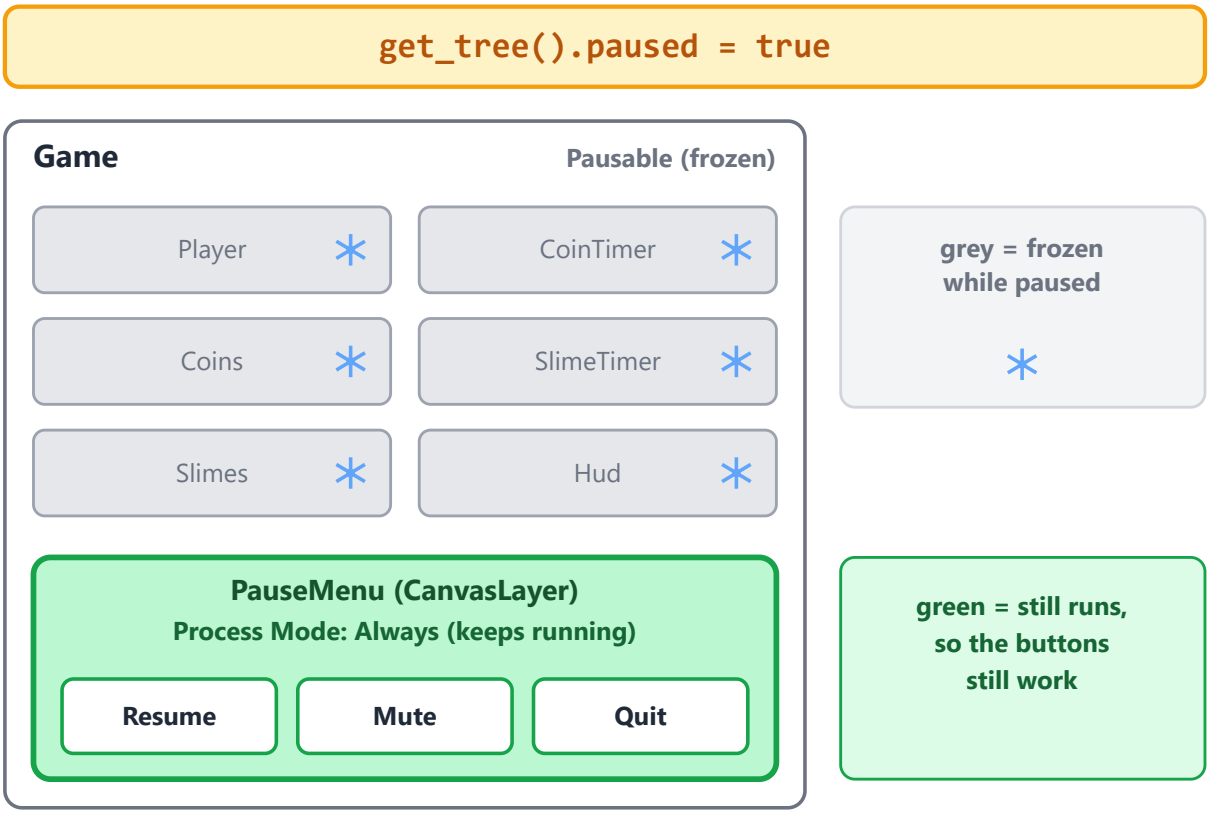


Figure 18.3: When the tree is paused, ordinary nodes freeze. The pause menu is set to Always, so it still hears the Esc key and the button clicks.

The pause menu node is set to **Always** (`process_mode = 3` in the scene file). Because its children inherit, its buttons keep working. Everything else, including Pip, the timers, and the slimes, is ordinary and freezes. We did not have to touch a single other script to make the game pausable.

```
PauseMenu (CanvasLayer)        pause_menu.gd, Layer 10, Process Mode: Always
├── Dim (ColorRect)            Full Rect, color black with alpha 0.6
└── Box (VBoxContainer)        centered, about 260 x 220
    ├── Title (Label)          "Paused", size 44, centered
    ├── ResumeButton (Button)  unique name, "Resume"
```

```
    ├── MuteCheck (CheckButton) unique name, "Mute sound"
    └── TitleButton (Button)   unique name, "Quit to title"
```

```gdscript
# pause_menu.gd  (PauseMenu, attached to the CanvasLayer)
extends CanvasLayer
## The pause menu. This node keeps working while the tree is paused
## (process_mode = Always in the scene), so it can also un-pause.


func _ready() -> void:
    visible = false
    %ResumeButton.pressed.connect(resume)
    %TitleButton.pressed.connect(_on_title_pressed)
    %MuteCheck.button_pressed = Save.muted
    %MuteCheck.toggled.connect(_on_mute_toggled)


func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("pause") and Game.running:
        if get_tree().paused:
            resume()
        else:
            pause()
        get_viewport().set_input_as_handled()


func _notification(what: int) -> void:
    # Pause by itself if the player clicks away to another window.
    if what == NOTIFICATION_APPLICATION_FOCUS_OUT and Game.running and not get_tree().paused:
        pause()


func pause() -> void:
    get_tree().paused = true
    visible = true
    %ResumeButton.grab_focus()
    Sfx.play("click")


func resume() -> void:
    get_tree().paused = false
    visible = false
    Sfx.play("click")


func _on_title_pressed() -> void:
    Game.go_to_title()


func _on_mute_toggled(pressed: bool) -> void:
    Save.set_muted(pressed)
```

Details worth studying:

- `visible = false` in `_ready()` hides the whole menu until needed. A hidden `CanvasLayer` hides its children and they cannot be clicked.
- **`_unhandled_input`** receives key presses that no button or other control has used. `event.is_action_pressed("pause")` is true for the frame the key goes down (not while held). The menu checks `Game.running` so Esc does nothing on the title screen. `set_input_as_handled()` tells Godot "I used this event" so nothing else reacts to it.
- **`_notification(what)`** is a function Godot calls with a number for system events. `NOTIFICATION_APPLICATION_FOCUS_OUT` arrives when the player clicks another window. Pausing automatically then is a kindness: the player is not eaten by slimes while answering a message. You can test it by clicking on another window while playing.

- `grab_focus()` puts keyboard focus on Resume, so the player can press Enter or Space right away, and arrow keys move between buttons.
- The "click" sound is played through `Sfx`, which is also set to Always, so it is heard while paused.

> **Warning:** Do not pause by setting `Engine.time_scale = 0` or by writing your own `is_paused` variable in every script. The tree flag plus process modes is built for this and covers physics, timers, and animation in one switch.

18.7 The title and game over screens

These two screens are almost pure user interface (Chapter 14), and each is a `Control` with a `VBoxContainer` in the middle.

Title

```
Title (Control)                title.gd, Layout: Full Rect
├── Pip (Node2D)               pip_face.gd, radius 54, position (480, 120)
└── Box (VBoxContainer)        centered, about 400 x 300
    ├── GameTitle (Label)      "Pip's Coin Run", gold, size 52
    ├── Hint (Label)           how to play, wraps text (Autowrap Mode: Word)
    ├── BestLabel (Label)      unique name
    ├── PlayButton (Button)    unique name, size 28
    ├── MuteCheck (CheckButton) unique name, "Mute sound"
    └── QuitButton (Button)    unique name
```

```gdscript
# title.gd  (Title, attached to the root Control)
extends Control
## Title screen. Scene tree is in title.tscn.


func _ready() -> void:
    %BestLabel.text = "Best score: %d" % Save.high_score
    %PlayButton.pressed.connect(_on_play_pressed)
    %QuitButton.pressed.connect(_on_quit_pressed)
    %MuteCheck.button_pressed = Save.muted
    %MuteCheck.toggled.connect(_on_mute_toggled)
    %QuitButton.visible = not OS.has_feature("web")  # a web page cannot "quit"
    %PlayButton.grab_focus()


func _on_play_pressed() -> void:
    Sfx.play("click")
    Game.start_run()


func _on_quit_pressed() -> void:
    get_tree().quit()


func _on_mute_toggled(pressed: bool) -> void:
    Save.set_muted(pressed)
```

Details: the Quit button is hidden in web builds, because a web page has no program to quit (`OS.has_feature("web")` is true there). `grab_focus()` on Play means a player can just press Enter. And the "Pip" here is the same `pip_face.gd` script drawn at three times the size.

Game over

```
GameOver (Control)             game_over.gd, Layout: Full Rect
└── Box (VBoxContainer)        centered, about 400 x 320
    ├── Title (Label)          "Game over", red, size 56
    ├── ScoreLabel (Label)     unique name, size 32
    ├── NewBestLabel (Label)   unique name, gold, "NEW BEST SCORE!"
    ├── BestLabel (Label)      unique name
    ├── AgainButton (Button)   unique name, "Play again"
    └── TitleButton (Button)   unique name, "Back to title"
```

```gdscript
# game_over.gd  (GameOver, attached to the root Control)
extends Control
## Game over screen. Scene tree is in game_over.tscn.


func _ready() -> void:
    %ScoreLabel.text = "Your score: %d" % Game.score
    %BestLabel.text = "Best score: %d" % Save.high_score
    %NewBestLabel.visible = Game.new_best
    %AgainButton.pressed.connect(Game.start_run)
    %TitleButton.pressed.connect(Game.go_to_title)
    %AgainButton.grab_focus()
```

Look at how short this is, because `Game` and `Save` do the work. The buttons connect straight to autoload functions: `%AgainButton.pressed.connect(Game.start_run)`. A function reference is passed without parentheses, and the signal calls it later. `%NewBestLabel.visible = Game.new_best` shows the banner only when the last run set a record: assigning a bool to `visible` is a neat way to avoid an `if`.

18.8 Input actions and project settings

Everything above depends on a few settings that are not in any script. In the project files they live in `project.godot`; in the editor you set them as follows.

Input Map (Project > Project Settings > Input Map). Add five actions by typing the name in the box at the top and clicking **Add**. Then click the + at the right of each row, press the key, and confirm:

Action	Keys
`move_left`	A, Left arrow
`move_right`	D, Right arrow
`move_up`	W, Up arrow
`move_down`	S, Down arrow
`pause`	Escape, P

(Choose "Physical Key" when asked, so the controls follow the key's position and work on other keyboard layouts, too.)

Autoloads (Project Settings > Globals > Autoload), in this order: `Save` (`res://save.gd`), `Game` (`res://game.gd`), `Sfx` (`res://sfx.gd`). `Game` uses `Save` and `Sfx` only inside functions called later, so the order is not critical, but `Save` first is a sound habit.

Display. Viewport Width `960`, Height `540`, Stretch Mode `canvas_items`.

Run. Main Scene: `res://title.tscn`.

Background. In **Rendering > Environment > Defaults**, set **Default Clear Color** to a deep blue (the project uses `Color(0.1, 0.11, 0.16)`).

Project 18: Pip's Coin Run

Goal: Build and run the complete game with title screen, play screen, pause menu, game over screen, and a high score that survives restarts.

Time: 75 to 90 minutes if you type everything; much less if you copy from the finished project and study.

> **Try it:** If you want a faster path, open `projects/ch18-pips-coin-run/` first, play the game, then rebuild one scene at a time in your own project, comparing with the original whenever something does not work.

Step 1: Project, settings, and input actions

1. Create a project called `PipsCoinRun`. Set the display and background as in section 18.8.
2. Add the five input actions of section 18.8.
3. Copy `icon.svg` from an earlier project, or keep the default icon.

Checkpoint: Project Settings > Input Map lists `move_left`, `move_right`, `move_up`, `move_down`, `pause`.

Step 2: The three autoloads

Create `save.gd`, `game.gd`, and `sfx.gd` with the code from section 18.3 and register them (section 18.8). `game.gd` mentions scene files that do not exist yet; that is fine, it only loads them when asked.

Checkpoint: Open each script: no red errors in the script editor.

Step 3: Pip

1. Create `pip_face.gd` (Node2D script).
2. Create the **Player** scene (tree in section 18.4), add the group `player`, set Motion Mode to Floating, attach `player.gd`, and save as `player.tscn`.

Checkpoint: Make a temporary test scene containing an instance of `player.tscn` and press F6. Pip shows as a blue face and arrow keys and WASD move Pip. (Pip can walk off screen because there is no arena yet; the play scene sets the arena later.)

Step 4: Coin and slime

1. Build `coin.tscn` and `slime.tscn` (trees in section 18.4), each with a CollisionShape2D whose shape is a new `CircleShape2D`.
2. Attach `coin.gd` and `slime.gd`.

Checkpoint: Put a coin and a slime into the test scene with Pip. Touching the coin makes it vanish with no errors in the Output (the score is ignored by `Game` because no run is active). The slime chases Pip and pops on contact. If Pip is hit, `Game.lose_life()` also does nothing yet, for the same reason.

Step 5: HUD and pause menu

1. Build `hud.tscn` and attach `hud.gd` (section 18.6).

2. Build `pause_menu.tscn`, set **Process > Mode** of its root to **Always**, set **Layer** to `10`, attach `pause_menu.gd`.

Step 6: The play scene

1. Create `game.tscn` as described in section 18.5. Turn **Autostart** on for both timers.
2. Attach `play_scene.gd` to the root node.

Step 7: Title and game over

1. Build `title.tscn` and `game_over.tscn` (trees in section 18.7) and attach their scripts.
2. Set the Main Scene to `res://title.tscn`.

Step 8: Play

Press **F5**.

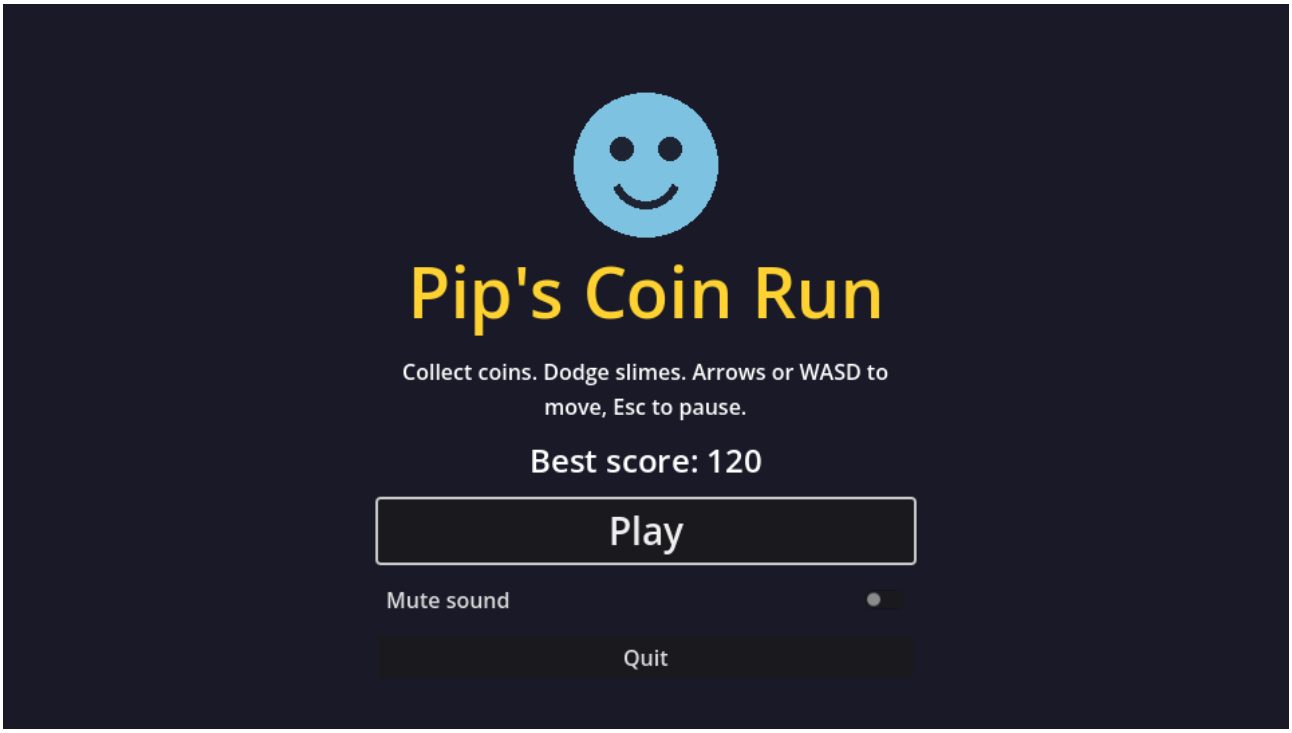


Figure 18.4: The title screen.

1. On the title screen, press Enter (Play already has focus). The play screen starts.
2. Collect coins. The score counts up in gold and a short beep plays.
3. Let a slime touch Pip: a lower beep, a life disappears, Pip blinks, and for a moment slimes pass through harmlessly.
4. Press **Esc**. Everything freezes, the screen dims, and the pause menu shows. Press Esc again, or Resume, and play continues from the exact same moment.
5. Pause, choose **Quit to title**, and press Play again: the new run starts unpaused, with score 0 and 3 lives.
6. Lose all three lives. The game over screen shows your score; if it is more than your best, the banner appears.
7. Close the game and reopen it: the title screen shows the best score from your last session.

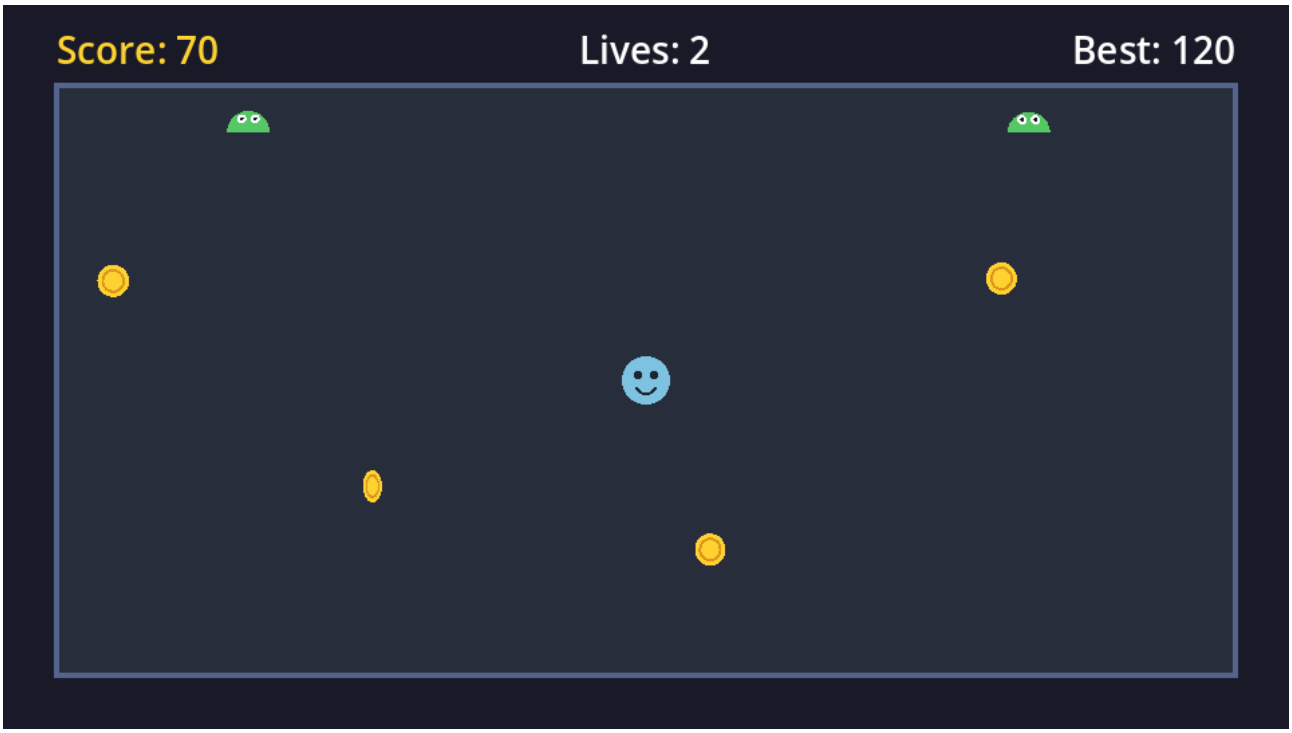


Figure 18.5: The play screen.



Figure 18.6: The pause menu. The play screen is still visible, frozen, behind the dim layer.

Step 9: Verify the logic with a headless test

The project includes `tests/test_run.gd`. It checks the input actions, loads the play scene without a window, lets it run for about 200 frames, injects a hit, ends a run, and checks that the high score is written to disk and read back:

```
# test_run.gd  (a SceneTree script, run from a terminal)
extends SceneTree
## Headless check of the run logic, the saved high score and the input map.
```

```gdscript
## Run: godot --headless --path . -s tests/test_run.gd

var _failures: int = 0
var _frame: int = 0
var _game_scene: Node


func check(condition: bool, label: String) -> void:
    if condition:
        print("  ok    ", label)
    else:
        print("  FAIL  ", label)
        _failures += 1


func _process(_delta: float) -> bool:
    _frame += 1
    var game: Node = root.get_node("Game")
    var save: Node = root.get_node("Save")
    if _frame == 1:
        save.path = "user://test_run.cfg"
        save.high_score = 0
        for action in ["move_left", "move_right", "move_up", "move_down", "pause"]:
            check(InputMap.has_action(action) and InputMap.action_get_events(action).size() == 2, "input
action " + action)
        # play scene runs on its own for a while
        _game_scene = load("res://game.tscn").instantiate()
        root.add_child(_game_scene)
        game.running = true
        game.score = 0
        game.lives = 3
    elif _frame == 2:
        game.add_score(30)
        check(game.score == 30, "score counts")
        check(not save.submit_score(0), "score 0 is not a record")
        var player: Node = _game_scene.get_node("Player")
        check(player.hurt(), "first hit counts")
        check(not player.hurt(), "second hit during blink is ignored")
        check(game.lives == 2, "one life lost")
    elif _frame == 200:
        check(_game_scene.get_node("Coins").get_child_count() > 0, "coins were spawned")
        # End the run: record must be saved to disk.
        game.score = 120
        _game_scene.queue_free()
    elif _frame == 201:
        game.running = true
        game.end_run()
        check(game.new_best, "120 beats 0")
        check(save.high_score == 120, "high score updated")
        save.high_score = 0
        save.load_data()
        check(save.high_score == 120, "high score survives a reload from disk")
        game.running = true
        game.score = 50
        game.end_run()
        check(not game.new_best and save.high_score == 120, "lower score does not replace the record")
        DirAccess.open("user://").remove("test_run.cfg")
        print("TEST ", "PASSED" if _failures == 0 else "FAILED (%d)" % _failures)
        quit(0 if _failures == 0 else 1)
    return false
```

Run it with:

```
godot --headless --path . -s tests/test_run.gd
```

You should see a list of `ok` lines and `TEST PASSED`. A test like this will not tell you whether the game is *fun*, but it catches broken rules (a life lost twice for one hit, a record not saved) in a second, every time you change the code.

> **Note:** A headless run of this project may end with a line such as "1 ObjectDB instance was leaked at exit". It is a harmless warning: the sound generator's playback object was still alive when the program quit.

Expected result

A complete, restartable game with four screens. Pip moves smoothly in the arena; coins add 10 points; slimes cost lives and speed up over time; Esc pauses without breaking timers; the game over screen shows the right message; and the best score is still there after restarting the program. The headless test passes.

Chapter summary

- Plan before you build: a screen-flow map, a rules list, and a table of who owns which data.
- Put data that outlives a scene in autoloads (`Game` for the run, `Save` for the file, `Sfx` for sound) and keep scene-only data in the scene.
- The `Game` autoload is the single place that changes scenes, so the whole screen flow is easy to read.
- Spawn things with `Timer` nodes, `instantiate()`, and `add_child()`, into container nodes. Fair placement uses a few retries.
- Make difficulty a function of elapsed time, with `minf` and `maxf` limits so numbers stay sensible.
- Pausing is `get_tree().paused`. Nodes freeze unless their **Process Mode** is Always (or When Paused). The pause menu is set to Always.
- Always reset run state when a new run starts, and un-pause before leaving the play screen.
- A small headless test of the game's rules catches mistakes quickly; a finished small game beats an unfinished big one.

Review questions

1. Which three lists did the chapter recommend making before writing any code? (Short answer)
2. Why do the score and lives live in the `Game` autoload instead of in the play scene? (Short answer)
3. True or false: the pause menu keeps working while the game is paused because it is a `CanvasLayer`.
4. What does **Process > Mode: Always** do, and which node in this game uses it? (Short answer)
5. `Game.go_to_title()` begins with `get_tree().paused = false`. What would go wrong if that line were missing? (Short answer)
6. In `slime.gd`, why does `_on_body_entered` call `body.is_in_group("player")` before `body.hurt()`? (Short answer)
7. Which expression keeps the time between slimes from dropping below 0.9 seconds? (Multiple choice) a) `minf(0.9, 3.0 - t * 0.03)` b) `maxf(0.9, 3.0 - t * 0.03)` c) `clamp(3.0, 0.9, t)` d) `3.0 - t * 0.03 > 0.9`
8. Why does the game save the high score only when it is beaten, rather than at the end of every run? (Short answer)
9. Why does `hurt()` return a `bool`? (Short answer)
10. True or false: if you set a new coin's position after calling `add_child`, the coin may be drawn for a frame at the wrong place.

Exercises

1. **Practice (easy):** Make coins worth different amounts: add an exported `points` value in `coin.gd`, and spawn one in five coins as a "big coin" worth 50, drawn larger or in orange. (Hint: set `points` and `scale` in `_spawn_coin` before `add_child`.)
2. **Practice (easy):** Change the balance: try three sets of your own numbers for the starting slime interval, speed ramp, and number of lives. Write down which felt best and why.
3. **Apply (medium):** Add a "shield" pickup: a blue circle that appears rarely and, when collected, makes Pip invulnerable for 4 seconds. Add a `shield()` function to `player.gd` for the pickup to call.
4. **Apply (medium):** Show a countdown "3, 2, 1, Go!" at the start of a run. The play scene should start with its two timers stopped (`Autostart` off) and Pip unable to move until the count ends. A `Label` and a `Timer`, or a `Tween`, will do.
5. **Challenge (hard):** Add a difficulty choice on the title screen (Easy / Normal / Hard) that changes the starting lives and the slime speed ramp. Store the choice in `Save` (a ConfigFile value) so it is remembered, and keep a separate high score per difficulty.

Common mistakes

- **Symptom:** pressing Esc does nothing. **Cause:** the `pause` action was not added to the Input Map, or its name differs from the script's `"pause"`. **Fix:** check the spelling in Project Settings > Input Map.
- **Symptom:** the pause menu appears but its buttons do not respond, or Esc cannot un-pause. **Cause:** the PauseMenu's process mode is Inherit, so it is frozen with the rest of the tree. **Fix:** set Process > Mode to **Always** on the PauseMenu root.
- **Symptom:** the new game after "Quit to title" starts frozen. **Cause:** `get_tree().paused` stayed true across the scene change. **Fix:** set it to false in `go_to_title()` and `start_run()`.
- **Symptom:** a coin or slime appears briefly in the top-left corner. **Cause:** the position was set after `add_child`. **Fix:** set the position first.
- **Symptom:** an error such as "Invalid access to property 'arena'" or Pip walks out of the map. **Cause:** the node is not named `Player`, so `$Player` finds nothing, or `player.arena = ARENA` is missing. **Fix:** match the names in the scene tree.
- **Symptom:** the score still goes up after the last life is lost. **Cause:** the `running` guard is missing in `add_score`. **Fix:** return early when the run is not active.

Further reading

- Pausing games and process modes: https://docs.godotengine.org/en/stable/tutorials/scripting/pausing_games.html
- Singletons (Autoload): https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html
- Changing scenes manually: https://docs.godotengine.org/en/stable/tutorials/scripting/change_scenes_manually.html
- Using Area2D: https://docs.godotengine.org/en/stable/tutorials/physics/using_area_2d.html
- Using InputEvent (actions and `_unhandled_input`): https://docs.godotengine.org/en/stable/tutorials/inputs/inputevent.html
- Saving games: https://docs.godotengine.org/en/stable/tutorials/io/saving_games.html

Part IV: Going further

Chapter 19: Your First Steps in 3D

Learning objectives

By the end of this chapter you will be able to:

- Describe the three axes of Godot's 3D space and say which way is up, left, and forward.
- Build a 3D scene from `Node3D`, `MeshInstance3D`, and primitive meshes (box, sphere, capsule, cylinder, plane).
- Place and aim a `Camera3D`, and explain what field of view and clipping distance do.
- Light a scene with directional, omni, and spot lights, turn shadows on, and set up a sky with `WorldEnvironment`.
- Create materials with `StandardMaterial3D` (color, metallic, roughness, emission) and apply them to meshes.
- Move a `CharacterBody3D` with gravity, jumping, and camera-relative walking, and detect pickups with an `Area3D`.

Key terms

- **3D space**: a world described by three numbers (x, y, z) for every position, instead of the two (x, y) you used in 2D.
- **Axis** (plural **axes**): one of the three straight lines, X, Y, and Z, that measure position in 3D. In Godot, Y points up.
- **Node3D**: the basic 3D node. It has a position, rotation, and scale, and it is the parent class of every other 3D node.
- **Transform**: the combined position, rotation, and scale of a 3D node, stored in one value (a `Transform3D`).
- **Mesh**: the shape of a 3D object, built from many tiny triangles. A mesh is only a shape; it has no place in the world until a `MeshInstance3D` shows it.
- **Primitive mesh**: a simple shape that Godot builds for you, such as a box, sphere, or capsule. No modeling program needed.
- **Camera3D**: the node that acts as the player's eye. What the camera sees is what appears in the game window.
- **Field of view (FOV)**: how wide the camera's view is, measured in degrees. A wide FOV shows more of the world.
- **Light**: a node that lights up 3D objects. Without a light, a 3D scene is dark.
- **Material**: a recipe that says how a surface looks: its color, shininess, glow, and so on.
- **Albedo**: the base color of a surface under plain white light.
- **Environment**: the settings for the world around your objects: the sky, the ambient light, and the overall color of the picture. It is held by a `WorldEnvironment` node.
- **CharacterBody3D**: the 3D version of the `CharacterBody2D` you already know. It moves with code and slides along walls and floors.

19.1 From flat to deep: 3D space

Everything you have built so far lives on a flat sheet. A position was two numbers: how far right, and how far down. Moving into 3D adds one more number: how far *forward or back*. That is the whole big idea. The rest of this chapter is about the new things that this third number lets you do, and the new things you must look after because of it.

Think of a 2D game as a painting on a wall, and a 3D game as a small stage with real depth, where you place little models and then walk a camera around them.

The three axes

Every position in 3D is three numbers, written as a `Vector3(x, y, z)`. Godot uses these directions:

- **X** points to the **right**. Negative X is left.
- **Y** points **up**. Negative Y is down.
- **Z** points **toward you** (toward the viewer). Negative Z points **forward**, away from you into the screen.

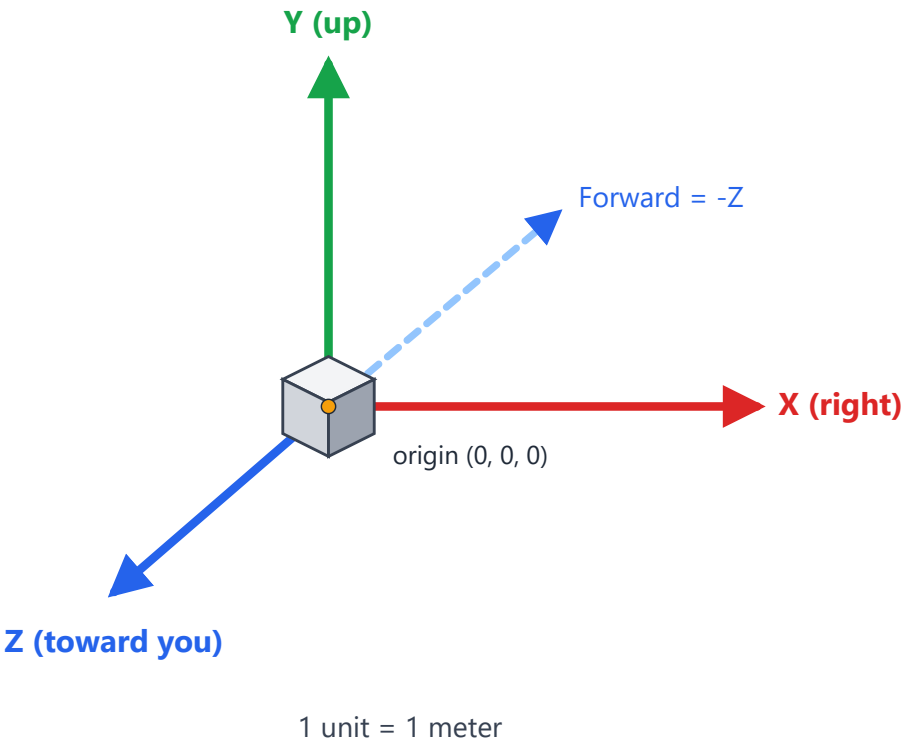


Figure 19.1: Godot's 3D axes. X is red, Y is green, Z is blue. "Forward" is the negative Z direction, which surprises almost everyone the first time.

Two details deserve a moment, because they cause most beginner confusion.

First, **Y is up**, not down. In 2D, Y grows downward on the screen. In 3D, Y grows upward, the way a height should. So `position.y = 3` means "three units above the origin."

Second, **forward is negative Z**. When a character in Godot "walks forward," its Z value *gets smaller*. A handy memory trick: the editor draws the axes in the colors **R**ed, **G**reen, **B**lue for **X**, **Y**, **Z**, in that order, like the letters RGB.

> **Note:** Programmers call this arrangement a *right-handed* coordinate system. If you point the fingers of your right hand along +X and curl them toward +Y, your thumb points along +Z. You do not need to remember the name. Just remember: X right, Y up, Z toward you.

Units

3D space has no pixels. Distances are measured in **units**, and by habit one unit is treated as **one meter**. A `CharacterBody3D` that is two units tall is about the height of a person. Godot's physics works best when your objects have sizes close to real life, so a character 2 units tall is a good default, and a character 200 units tall will feel slow and floaty.

Node3D: the 3D version of Node2D

In Chapter 6 you met `Node2D`, which gave a node a position, rotation, and scale. The 3D twin is `Node3D`. Every 3D node (cameras, lights, meshes, bodies) is a `Node3D` underneath, so every one of them has:

- `position`: a `Vector3`, where the node is, relative to its parent.
- `rotation`: a `Vector3` of three angles in **radians**, one around each axis. (`rotation_degrees` is the same thing in degrees, easier for humans.)
- `scale`: a `Vector3`, how much bigger or smaller the node is along each axis.

Together these three are the node's **transform**. In 2D, rotation was a single number, because there is only one way to spin on a flat sheet. In 3D there are three ways to turn:

- Around **Y**: turning left and right, like shaking your head "no". This is called **yaw**.
- Around **X**: tilting up and down, like nodding "yes". This is **pitch**.
- Around **Z**: tilting your head toward a shoulder. This is **roll**.

The most useful rotation in simple games is the Y rotation: turning a character to face a new direction on the ground.

As in 2D, children follow their parents. If you move a `Node3D` called `Car` that has four wheel nodes as children, the wheels come along. The child's `position` is measured from the parent. To get the position in the world, use `global_position`, exactly as in 2D.

```
# Fragment: a few Node3D basics, inside any script on a Node3D.
position = Vector3(0, 2, -5)      # 2 units up, 5 units forward
rotation_degrees.y = 90.0          # turn a quarter turn to the left
scale = Vector3(2, 1, 1)           # twice as wide, same height and depth
rotate_y(deg_to_rad(45.0))         # spin a further 45 degrees around Y
```

> **Why it works:** `rotate_y` takes an amount in radians, because the math inside the engine uses radians. `deg_to_rad` converts a number from degrees. 180 degrees is about 3.14159 radians (that number is called pi). The built-in constant `TAU` equals a full turn (360 degrees), which is 6.28318 radians.

Moving around the 3D editor

The 3D editor has its own workspace. Click **3D** at the top of the editor. The default mouse controls are:

- **Right mouse button held**: look around (like turning your head). While holding it, press **W A S D** to fly, and **Q / E** to go down and up. This is "freelook."

- **Middle mouse button held**: orbit around what you are looking at.
- **Mouse wheel**: zoom in and out.
- **F**: focus on the selected node.

Selected nodes show colored arrows called the **gizmo**. Drag an arrow to move along that axis, drag a ring to rotate, drag a small square to scale. The arrows use the same red, green, and blue as the axes in Figure 19.1.

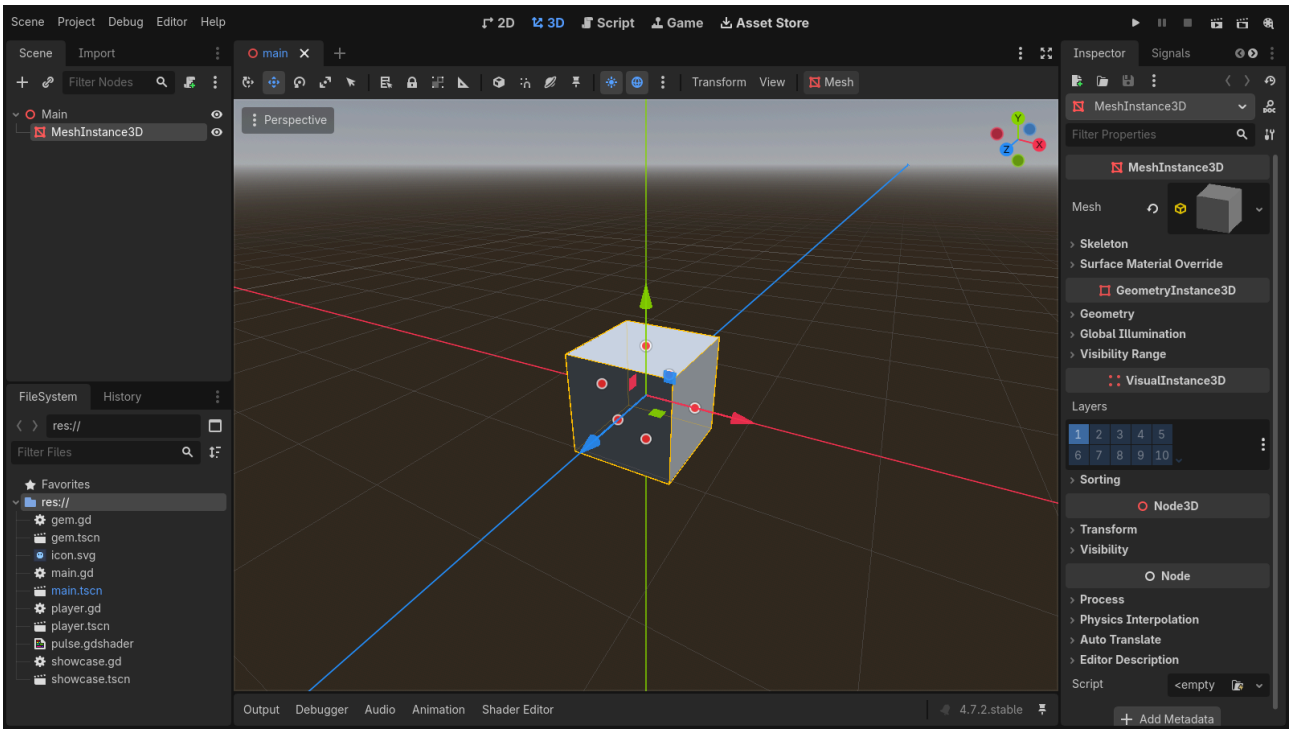


Figure 19.2: The 3D workspace. The red, green, and blue arrows on the selected box are the move gizmo. The Inspector on the right shows the Transform section.

> **Try it:** Create a new project (Compatibility renderer is fine). Make a `Node3D` as the root, rename it `Main`, and save it as `main.tscn`. Click the **3D** workspace. Right-click-drag to look around, then use the mouse wheel to zoom. Find the three colored lines of the world axes crossing at the center.

19.2 Meshes: giving things a shape

A `Node3D` by itself is invisible: it is only a point with a position. To see something, you need a **mesh**, a surface made of many small triangles. Graphics chips are extremely good at drawing triangles, so all 3D art, from a simple cube to a detailed dragon, is built from them.

Two parts work together, and the split is important:

- The **`Mesh`** is a *resource*: the shape data. It does not live in the world.
- The **`MeshInstance3D`** is a *node*: it puts one mesh at a place in the world.

This mirrors what you know from 2D. A `Texture2D` is a picture resource, and a `Sprite2D` is the node that shows it. In 3D, a `Mesh` is the shape and a `MeshInstance3D` is the node that shows it. One mesh can be shown by many instances, which saves memory.

Primitive meshes

You do not need a modeling program to start. Godot can build common shapes for you. They are called **primitive meshes**, and you set them from the `Mesh` property of a `MeshInstance3D`:

Mesh	What it is	Useful properties
`BoxMesh`	A cuboid (a box or brick)	`size` (a Vector3)
`SphereMesh`	A ball	`radius`, `height` (twice the radius for a round ball)
`CapsuleMesh`	A pill shape	`radius`, `height`
`CylinderMesh`	A can or pillar	`top_radius`, `bottom_radius`, `height`
`PlaneMesh`	A flat sheet facing up	`size` (a Vector2)
`PrismMesh`	A triangular wedge or roof	`size`
`TorusMesh`	A donut ring	`inner_radius`, `outer_radius`
`QuadMesh`	A flat sheet facing the camera	`size`

Adding a mesh in the editor takes three clicks:

1. Select `Main` in the Scene dock and press the + button (Add/Create a New Node).
2. Choose `MeshInstance3D`.
3. In the Inspector, click the `Mesh` property and choose **New BoxMesh**. A gray cube appears at the origin.

Click the BoxMesh in the Inspector to open it and change its `Size`. Notice that the mesh and the node are separate. Moving the node moves the box; changing the mesh's size changes the box's shape.

> **Note:** Your box looks dark gray on a gray background, flat, with no shading. That is because there is no light yet. The editor adds a temporary preview light (the sun and moon buttons at the top of the viewport), but your game would show nothing useful. We fix that in section 19.4.

A mesh can also be made in code

Just as you can create a `Label` with `Label.new()`, you can create a mesh instance in a script. This is handy for creating many objects quickly. This code builds a red ball when the scene starts. Attach it to the root `Node3D`:

```
# File: spawn_ball.gd
# Attach to: the root Node3D of a scene
extends Node3D

func _ready() -> void:
    var ball := MeshInstance3D.new()   # the node that shows a mesh
    ball.mesh = SphereMesh.new()       # a sphere with a 1 unit diameter
    ball.position = Vector3(0.0, 1.0, 0.0)
    add_child(ball)                    # without this, the ball never appears
```

The `SphereMesh` default is a ball of radius 0.5, so it is one unit wide. The `add_child` line is easy to forget. A node that is created but never added to the tree is not part of the game.

Pivot points and parenting

A mesh is drawn around the node's origin. A `BoxMesh` of size 2 sits centered on the node's position, so half of it is *below* the node. This matters when you want to rest a box on the floor: if the box is 2 tall, put its node at `y = 1`.

A good habit is to make an empty `Node3D` the parent of a group of meshes. Move or rotate the parent to move the whole object. In this chapter the player is a `CharacterBody3D` with a `Body` node that holds two meshes (a capsule and a small box for a "nose"), so we can turn the body without turning the camera.

19.3 The camera: your eye in the world

A 3D scene can contain many objects, but you only see the world through a **`Camera3D`**. If your scene has no camera, the running game shows a blank screen. In the editor, the viewport has its own freelook camera, which is why you can see things there, but that camera does not exist in the game.

Add a `Camera3D` as a child of `Main`. Notice that the small **Preview** checkbox appears at the top of the viewport when a camera is selected. Turn it on to see exactly what the game will show.

The camera looks along its own **negative Z axis**. That means by default a camera at the origin looks toward -Z, the "forward" direction in Figure 19.1. To aim it, move it and rotate it with the gizmo, or in a script call:

```
# Fragment: aim a camera at a point. Place inside a script that has a camera variable.
camera.position = Vector3(0, 3, 8)
camera.look_at(Vector3.ZERO)   # turn so that the camera looks at the world origin
```

`look_at` is one of the most useful functions in 3D. It rotates a node so its forward direction (-Z) points at the position you give.

Field of view

The most important camera property is **FOV** (field of view), measured in degrees. It controls how much of the world fits in the picture.

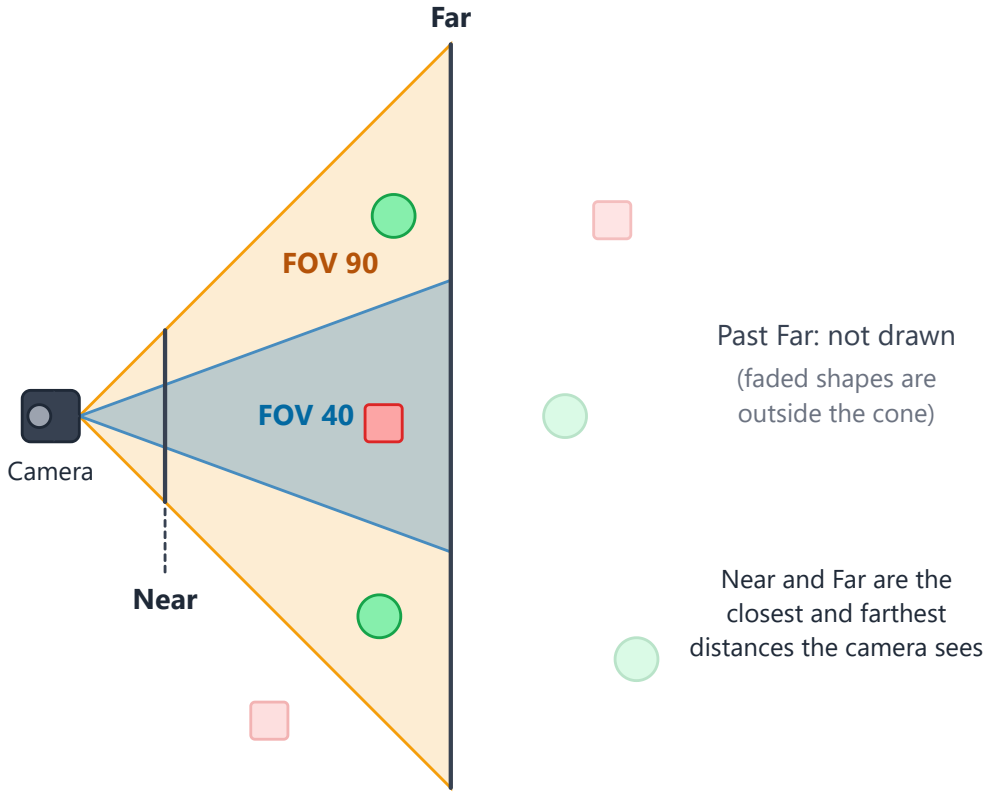


Figure 19.3: The camera's view is a pyramid. A narrow FOV zooms in on the middle. A wide FOV shows more, but stretches things near the edges. Anything closer than Near or farther than Far is not drawn.

- A **small FOV** (30-50) looks zoomed in, like a telephoto lens.
- The default of **75** is a comfortable middle value for most games.
- A **large FOV** (90-110) shows more of the world and feels fast, but bends the edges.

Two more properties set the **clipping planes**: `Near` and `Far`. Anything closer than `Near` or farther than `Far` is not drawn. The defaults (0.05 and 4000) are fine for a small project. If you ever see things "cut off" when you get close, or distant objects disappearing, these are the numbers to check.

The `Projection` property chooses between:

- **Perspective** (the default): distant things look smaller, like real life.
- **Orthogonal**: every object keeps its size regardless of distance. This is the look of classic "isometric" strategy games. Its width is set with the `Size` property.

The camera pivot: a simple third-person camera

Many games place the camera behind and above the player. A simple way is to build a small *rig*:

```
Player (CharacterBody3D)
`-- CameraPivot (Node3D)        # sits at the player's chest, rotates to orbit
    `-- Camera3D                # 3 units up and 7 units back, tilted down 20 degrees
```

The camera is a child of the pivot and sits some distance away. When you rotate the pivot around Y, the camera swings in a circle around the player, like a moon around a planet. We use this exact rig in the project.

> **Warning:** The pivot is a child of the player, so it moves with the player. But if the player's *body* turned, the pivot would turn too, and the camera would spin when the character spins. That is why in the project the player's meshes live under a separate `Body` node: we turn `Body`, never the whole `Player`.

If you need the camera to avoid going through walls, Godot has a `SpringArm3D` node made for this. It is a good next step after this chapter, and the docs link in Further reading covers it.

19.4 Light, shadow, and sky

Without a light, a 3D scene shows only black or a flat color. Lights are what make shapes look solid, because shading (the gradual change from bright to dark across a surface) is how our eyes know that a ball is round and not a flat circle.

Godot has three kinds of light you will use constantly. Compare them in Figure 19.4.

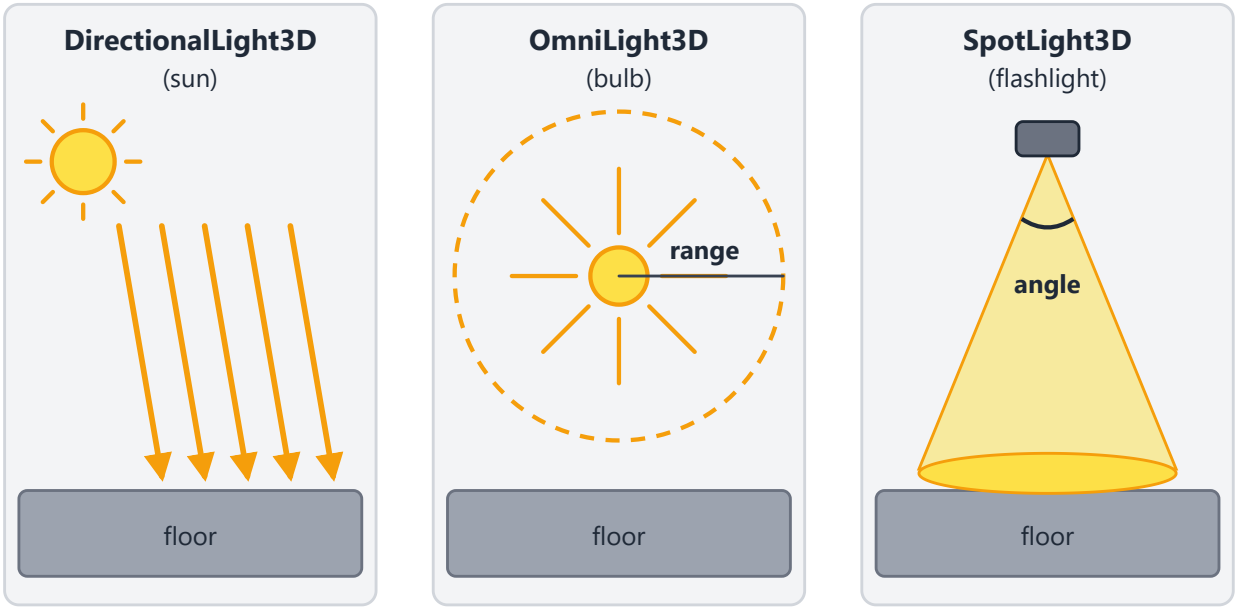


Figure 19.4: The three basic lights. Directional light is like the sun: the same direction everywhere. Omni light is like a bulb: it shines in all directions and fades with distance. Spot light is like a flashlight: a cone.

- **`DirectionalLight3D`** acts like the sun. Only its *rotation* matters; moving it changes nothing, because the sun is so far away that its rays are all parallel. Use one to light a whole outdoor scene.
- **`OmniLight3D`** is a light bulb. It shines in every direction and fades out at a distance set by `Omni Range`. Use it for lamps, torches, and glowing crystals.
- **`SpotLight3D`** is a flashlight. It shines in a cone. `Spot Range` sets how far it reaches and `Spot Angle` sets how wide the cone is.

All three lights share these useful properties:

- `Light Color`: the color of the light.
- `Light Energy`: the brightness. The default is 1.0; raise it for a stronger light.
- `Shadow Enabled`: when turned on, objects block light and cast shadows. Shadows make a huge difference to how real a scene looks, because they tell the eye where objects sit relative to the ground. They also cost speed, so turn them on for the important lights and leave them off for decoration.

> **Try it:** Add a `DirectionalLight3D` to your scene. Rotate it so the sun comes from above and a little to the side (tilt it about -45 degrees on X). Turn on `Shadow > Enabled`. Put a `PlaneMesh` below a floating box, and see the shadow land on the plane.

Environment: the sky and the ambient light

Lights only light what they directly hit. Surfaces that face away from every light would be pitch black, unlike the real world where light bounces everywhere. Godot fakes this with **ambient light**, set in the **Environment** resource.

You attach an `Environment` using a `WorldEnvironment` node. Add one to the scene, and in the Inspector:

1. Click the `Environment` property and choose **New Environment**.
2. Open it. Under **Background**, set `Mode` to **Sky**.
3. Under **Sky**, click the `Sky` property and choose **New Sky**, then set its `Sky Material` to **New ProceduralSkyMaterial**. A blue gradient sky appears.
4. Under **Ambient Light**, set `Source` to **Sky** (the light in the sky also softly lights the objects) or **Color** to pick your own.

The `ProceduralSkyMaterial` is a sky made from a few colors with a bright sun disk. It needs no image file, which is why we use it in the project.

> **Note:** Godot has three renderers. You chose one when you created the project: **Forward+** (best quality, for powerful computers), **Mobile**, and **Compatibility** (runs on almost anything, including web). The examples in this chapter use Compatibility, so they run on any computer you may have. If your computer is newer and you chose Forward+, everything in this chapter also works there, with more advanced effects available.

Showcase: lights in action

The project for this chapter includes a small scene, `showcase.tscn`, that puts all three lights in front of five spheres. You will build it in the exercises and you can run it from the project at any time (F6 with the scene open). Look at how a moving omni light slides its highlight across the surfaces.

19.5 Materials: color, shine, and glow

A mesh is only a shape. What color is it? Is it shiny? Does it glow? That is the job of a **material**. Godot's everyday material is `StandardMaterial3D`.

To add one in the editor, select a `MeshInstance3D`, open its `Mesh`, and click the `Material` property, then choose **New StandardMaterial3D**. (You can also set `Material Override` on the `MeshInstance3D`, which replaces whatever material the mesh has.)

The properties that matter first

`StandardMaterial3D` has dozens of properties. You will use five of them all the time:

- `Albedo > Color`: the base color. If you only change one thing, change this.

- `Metallic`: from 0.0 (plastic, wood, cloth) to 1.0 (metal). Metals reflect their surroundings and have a tinted shine.
- `Roughness`: from 0.0 (a mirror-smooth surface with a tight sharp highlight) to 1.0 (a rough, chalky surface with no highlight).
- `Emission`: turn it on to make the surface glow with its own color, as if it were a lamp. A glowing surface does not light up its neighbors by itself, but it looks bright.
- `Albedo > Texture`: lets you wrap a picture around the shape. We do not use pictures in this chapter, but the property is there.

Figure 19.5 shows how metallic and roughness work together. This kind of material system is called **physically based rendering** (PBR) because it follows how real surfaces behave. You do not need the physics. Remember that metallic says "metal or not" and roughness says "how blurry is the shine."

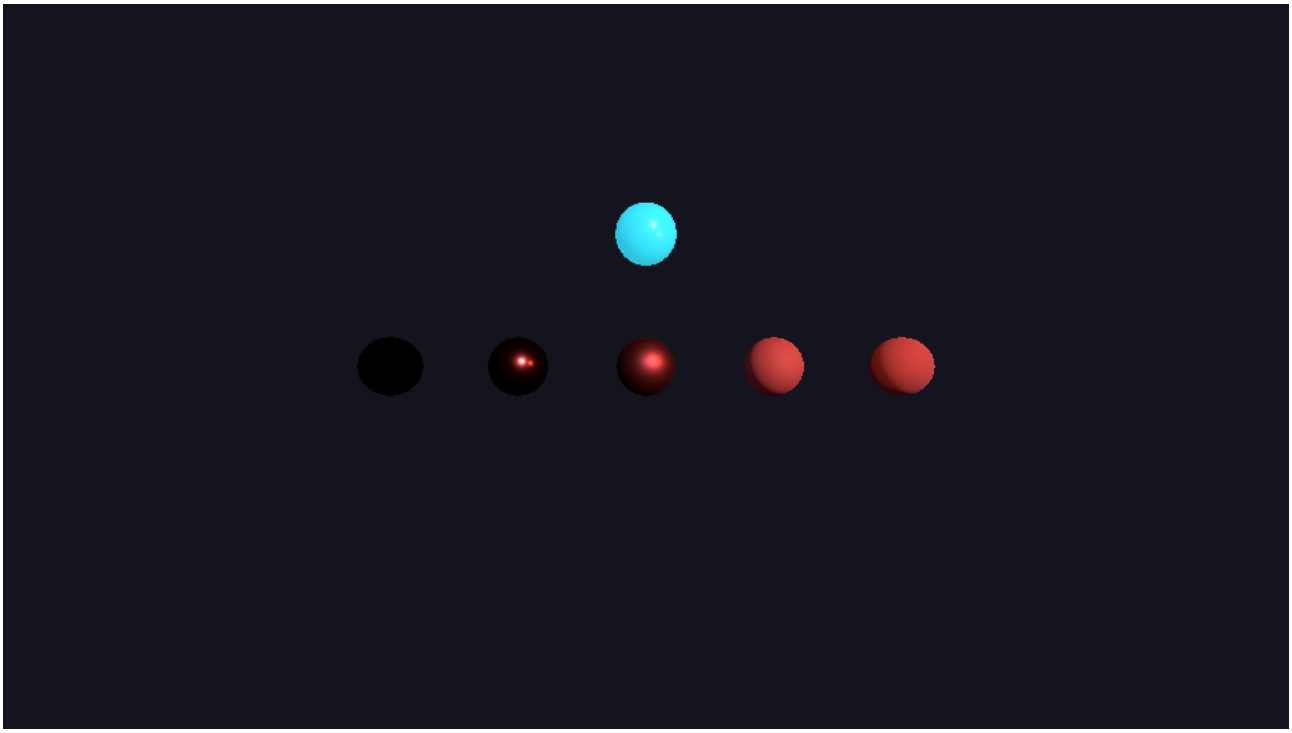


Figure 19.5: The showcase scene. Left to right: smooth metal, then increasingly rough metal, then plastic-like materials. The more rough a surface, the more blurred the highlight. The sixth sphere uses a small shader.

Making materials in code

This is the code that builds the five spheres in `showcase.gd`. It is a good example of creating a mesh, a material, and a node in a loop:

```gdscript
# File: showcase.gd (excerpt)
# Attach to: the root Node3D of showcase.tscn
func _ready() -> void:
    for i in 5:
        var sphere := MeshInstance3D.new()
        sphere.mesh = SphereMesh.new()

        var material := StandardMaterial3D.new()
        material.albedo_color = Color(0.9, 0.3, 0.3)
        material.roughness = i / 4.0       # 0.0, 0.25, 0.5, 0.75, 1.0
        material.metallic = 1.0 if i < 3 else 0.0

        sphere.material_override = material
```

```
        sphere.position = Vector3((i - 2) * 2.2, 0.0, 0.0)
        add_child(sphere)
```

Read it slowly. The loop runs five times, with `i` equal to 0, 1, 2, 3, and 4. Each time, it makes a new sphere and a new material, sets the roughness to a different number between 0 and 1, and places the sphere along the X axis (from -4.4 to +4.4). The expression `1.0 if i < 3 else 0.0` is a one-line way to say "metal for the first three, plain for the rest."

Sharing materials

A material is a resource, so you can share it. If you drag one `StandardMaterial3D` onto ten different meshes, they all use the same one, and editing it changes all ten. This is efficient, and it is also a trap: if you change a *shared* material in code, you change every object using it. To change just one, make it **unique** (in the Inspector, right-click the material and choose **Make Unique**), or create a fresh material in code as the loop above does.

A first look at a 3D shader

Chapter 20 covers shaders in depth, so here is only a small taste. A **spatial shader** is a short program that runs on the graphics chip and decides how every point on a surface looks. This one makes a surface pulse with a glow:

```
// File: pulse.gdshader
shader_type spatial;

uniform vec4 base_color : source_color = vec4(0.2, 0.8, 0.9, 1.0);
uniform float pulse_speed = 2.0;

void fragment() {
    float pulse = 0.5 + 0.5 * sin(TIME * pulse_speed);
    ALBEDO = base_color.rgb;
    EMISSION = base_color.rgb * pulse;
    ROUGHNESS = 0.3;
}
```

Shaders use their own language (GLSL-like, not GDScript), so the syntax looks a little different. The `fragment()` function runs once for every visible dot on the surface. `ALBEDO`, `EMISSION`, and `ROUGHNESS` are built-in outputs that mean the same as the material properties of the same names. `TIME` is a built-in number of seconds since the game started, so the `sin` wave makes the glow rise and fall. The `uniform` lines create settings you can change from outside.

To use it, make a `ShaderMaterial`, load the shader into it, and use that as a material. The project's `showcase.gd` does so for a sixth sphere:

```
# Fragment: use a spatial shader on a sphere (part of showcase.gd's _ready).
var glow := MeshInstance3D.new()
glow.mesh = SphereMesh.new()
var glow_material := ShaderMaterial.new()
glow_material.shader = load("res://pulse.gdshader")
glow.material_override = glow_material
glow.position = Vector3(0.0, 2.2, 0.0)
add_child(glow)
```

19.6 Moving in 3D

Movement code in 3D is very close to the 2D code in Chapter 8. The same ideas apply: a velocity, `move_and_slide()`, and `delta`. The difference is that velocity is now a `Vector3`, and you have gravity to look after on a vertical axis.

Bodies and shapes in 3D

The collision nodes you know from 2D all have 3D twins:

2D	3D
`StaticBody2D`	`StaticBody3D`
`CharacterBody2D`	`CharacterBody3D`
`RigidBody2D`	`RigidBody3D`
`Area2D`	`Area3D`
`CollisionShape2D`	`CollisionShape3D`

A `CollisionShape3D` needs a **shape** resource: `BoxShape3D`, `SphereShape3D`, `CapsuleShape3D`, `CylinderShape3D`, and so on. The rule from Chapter 8 still holds: a body with no collision shape does nothing (the editor shows a yellow warning triangle). A capsule is a popular shape for characters because its rounded ends glide over small bumps and around corners.

> **Warning:** The collision shape and the mesh are two separate things. Changing one never changes the other. If your character walks through a wall, or hovers above the floor, check that the `CollisionShape3D` size matches the `MeshInstance3D` size.

Gravity and jumping

In 2D you may have added gravity to `velocity.y`. In 3D, Y is up, so gravity *subtracts* from `velocity.y`:

```gdscript
# Fragment: gravity and a jump inside _physics_process(delta: float).
if not is_on_floor():
    velocity.y -= gravity * delta        # fall faster and faster

if Input.is_action_just_pressed("jump") and is_on_floor():
    velocity.y = jump_velocity           # shoot upward
```

`is_on_floor()` only works after `move_and_slide()` has been called, so it is correct on the next frame. The ground is anything that makes the body stop when falling, and "floor" is decided by the angle of the surface (flatter than 45 degrees by default).

Walking in the direction the camera faces

In 2D, pressing "right" always moves you toward the right side of the screen. In a 3D game with a rotating camera, "forward" should mean "away from the camera." We calculate it this way:

1. Read the input as a `Vector2` using `Input.get_vector`. The x part is left/right, the y part is back/forward. (Pressing "forward" gives a negative y, so it maps neatly to -Z.)
2. Turn that into a `Vector3(input.x, 0, input.y)`, with a 0 for the up/down part.

3. Rotate the vector by the camera pivot's orientation, with `pivot.global_transform.basis * vector`. A **basis** is the rotation part of a transform; multiplying a vector by it turns the vector the same way the pivot is turned.
4. Set the horizontal part of the velocity to the result times the speed.

```
# Fragment: camera-relative walking. Inside _physics_process.
var input_dir: Vector2 = Input.get_vector("move_left", "move_right", "move_forward", "move_back")
var direction: Vector3 = camera_pivot.global_transform.basis * Vector3(input_dir.x, 0.0, input_dir.y)
direction.y = 0.0
direction = direction.normalized()
velocity.x = direction.x * speed
velocity.z = direction.z * speed
```

The `normalized()` call matters. It scales a vector so its length is exactly 1. Without it, walking diagonally would be faster than walking straight. `Input.get_vector` already keeps its result inside the unit circle for you, but after rotating and flattening it, we normalize again to be safe.

> **Why it works:** If the camera pivot has turned 90 degrees, then the vector for "forward" (0, 0, -1) becomes a vector pointing sideways in the world. Multiplying by the basis does that turn for us, and we never have to write a single line of trigonometry.

Turning the body to face where we walk

If the capsule never turns, it looks wrong, because a capsule is the same from every side. So we add a small "nose" box and rotate the body so the nose leads the way. The needed angle comes from `atan2`, a math function that turns an x and a z into an angle. Then `lerp_angle` turns smoothly toward it instead of snapping:

```
# Fragment: smooth turning. Inside _physics_process, after direction is known.
if direction != Vector3.ZERO:
    var target_angle: float = atan2(-direction.x, -direction.z)
    body.rotation.y = lerp_angle(body.rotation.y, target_angle, turn_speed * delta)
```

`lerp_angle` is like `lerp`, but it understands that angles wrap around: it takes the short way from 350 degrees to 10 degrees, and not the long way round.

Pickups with Area3D

An `Area3D` is a zone that notices when bodies enter it, just like `Area2D`. In the project, each gem is an `Area3D` with a small `SphereShape3D`. When the player's body enters, the gem sends out a `collected` signal and removes itself. Signals work just as you learned in Chapters 5 and 15.

```
# Fragment: the heart of gem.gd.
func _on_body_entered(body: Node3D) -> void:
    if body.name == "Player":
        collected.emit()
        queue_free()
```

Project 19: Pip's Gem Garden

Goal. Build a small 3D game. Pip is a blue capsule with a yellow nose. Eight spinning gems are scattered across a green field with a wall and a step to jump on. Walk around with W A S D, jump with Space, turn the camera with Q and E, and collect all the gems. All shapes are primitive meshes, so you need no outside files.

Time. About 60-90 minutes.

Final project. The complete project is in `projects/ch19-first-3d/`. If you get stuck, open it and compare.

```
projects/ch19-first-3d/
|-- project.godot
|-- icon.svg
|-- main.tscn         # the world (the main scene)
|-- main.gd           # spawns gems, score, restart
|-- player.tscn       # Pip and the camera rig
|-- player.gd         # walking, jumping, camera turning
|-- gem.tscn          # one collectible gem
|-- gem.gd            # spinning and collecting
|-- showcase.tscn     # extra: lights and materials demo
|-- showcase.gd
`-- pulse.gdshader    # extra: a small spatial shader
```

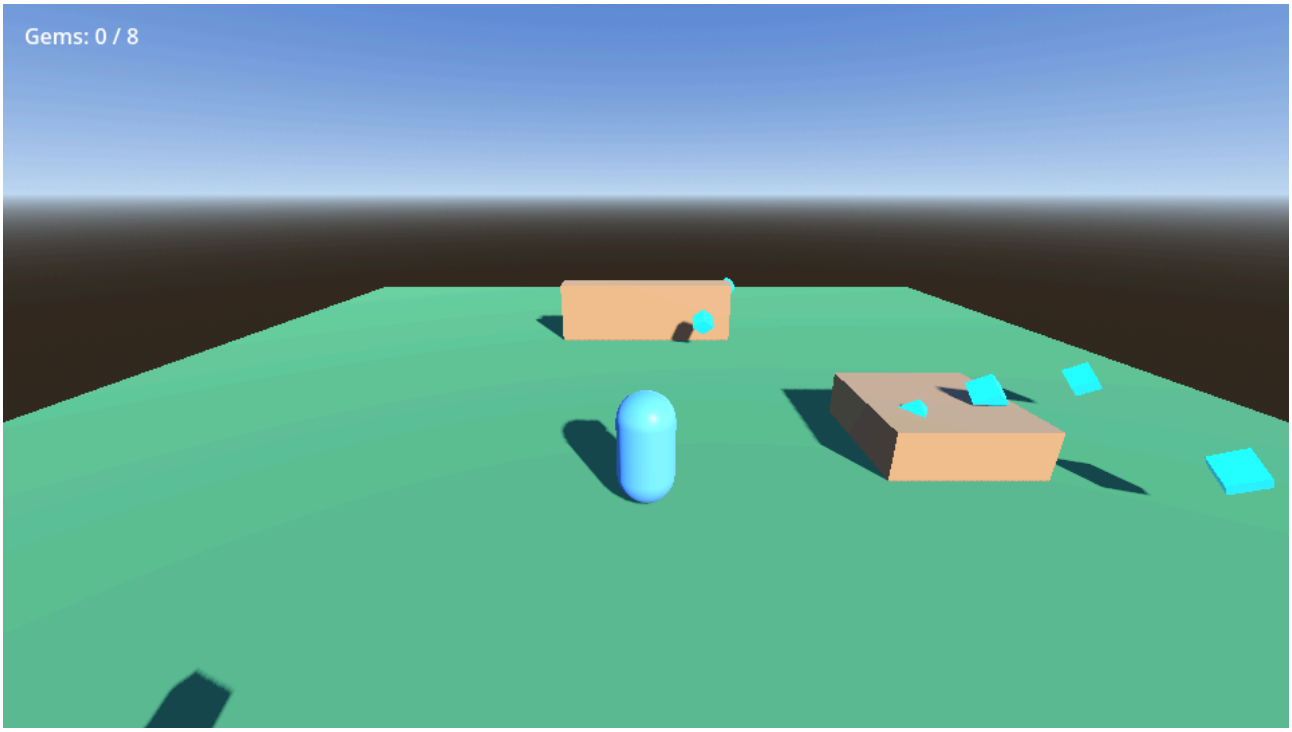


Figure 19.6: The finished project. The shadow under Pip helps you see where he is standing.

Step 1: Create the project and the input actions

1. Create a new project named `gem-garden` with the **Compatibility** renderer.
2. Open **Project > Project Settings > Input Map**. Add these actions. For each, type the name, click **Add**, then click the plus button and press the key:

Action name	Key
`move_left`	A
`move_right`	D
`move_forward`	W
`move_back`	S
`jump`	Space
`camera_left`	Q

Action name	Key
`camera_right`	E

Step 2: The player scene

Create a new scene with a `CharacterBody3D` as the root, rename it `Player`, and save it as `player.tscn`. Build this tree:

```
Player (CharacterBody3D)          # script: player.gd
|-- Body (Node3D)
|   |-- Capsule (MeshInstance3D)   # Mesh: CapsuleMesh
|   `-- Nose (MeshInstance3D)      # Mesh: BoxMesh, size (0.3, 0.3, 0.5)
|-- CollisionShape3D               # Shape: CapsuleShape3D
`-- CameraPivot (Node3D)           # Position: (0, 1, 0)
    `-- Camera3D                   # Position: (0, 3, 7), Rotation X: -20 degrees
```

Details to set:

- `Capsule`: new `CapsuleMesh` (the defaults radius 0.5, height 2 are right). Give it a `StandardMaterial3D` with a light blue albedo.
- `Nose`: new `BoxMesh`, `Size` of (0.3, 0.3, 0.5). Set its node `Position` to (0, 0.4, -0.55), so it sticks out of the front (-Z). Give it a yellow material.
- `CollisionShape3D`: new `CapsuleShape3D` with default values.
- `Camera3D`: position (0, 3, 7), rotation in degrees (-20, 0, 0). In the Inspector, tick **Current** so it is the active camera.

Now attach a script to `Player`:

```gdscript
# File: player.gd
# Attach to: Player (CharacterBody3D) in player.tscn
extends CharacterBody3D

@export var speed: float = 6.0
@export var jump_velocity: float = 7.0
@export var gravity: float = 20.0
@export var turn_speed: float = 10.0
@export var camera_turn_speed: float = 2.0

@onready var camera_pivot: Node3D = $CameraPivot
@onready var body: Node3D = $Body


func _physics_process(delta: float) -> void:
    # 1. Turn the camera pivot with Q and E.
    var camera_turn: float = Input.get_axis("camera_right", "camera_left")
    camera_pivot.rotate_y(camera_turn * camera_turn_speed * delta)

    # 2. Gravity pulls us down while we are in the air.
    if not is_on_floor():
        velocity.y -= gravity * delta

    # 3. Jump.
    if Input.is_action_just_pressed("jump") and is_on_floor():
        velocity.y = jump_velocity

    # 4. Walk relative to where the camera looks.
    var input_dir: Vector2 = Input.get_vector("move_left", "move_right", "move_forward", "move_back")
    var direction: Vector3 = camera_pivot.global_transform.basis * Vector3(input_dir.x, 0.0, input_dir.y)
    direction.y = 0.0
    direction = direction.normalized()

    velocity.x = direction.x * speed
```

```
    velocity.z = direction.z * speed

    # 5. Turn the body so the nose points where we walk.
    if direction != Vector3.ZERO:
        var target_angle: float = atan2(-direction.x, -direction.z)
        body.rotation.y = lerp_angle(body.rotation.y, target_angle, turn_speed * delta)

    move_and_slide()
```

Save the scene. Do not run it yet, because there is no floor for Pip to stand on. Pip would fall forever.

Step 3: The gem scene

Create a new scene with an `Area3D` root named `Gem`, and save it as `gem.tscn`:

```
Gem (Area3D)                # script: gem.gd
|-- Mesh (MeshInstance3D)    # Mesh: BoxMesh, size (0.5, 0.5, 0.5)
`-- CollisionShape3D         # Shape: SphereShape3D, radius 0.6
```

Give the box a `StandardMaterial3D` with a cyan albedo, `Metallic` 0.6, `Roughness` 0.2, and `Emission` enabled with a dark cyan color so it glows a little. Then attach this script to `Gem`:

```
# File: gem.gd
# Attach to: Gem (Area3D) in gem.tscn
extends Area3D

signal collected

@export var spin_speed: float = 2.0
@export var bob_height: float = 0.2

@onready var mesh: MeshInstance3D = $Mesh

var _time: float = 0.0
var _base_y: float = 0.0


func _ready() -> void:
    body_entered.connect(_on_body_entered)
    mesh.rotation_degrees = Vector3(35.0, 0.0, 45.0)  # tilt the cube so it looks like a gem
    _base_y = mesh.position.y
    _time = randf() * TAU  # so gems do not all bob in sync


func _process(delta: float) -> void:
    _time += delta
    mesh.rotate_y(spin_speed * delta)
    mesh.position.y = _base_y + sin(_time * 3.0) * bob_height


func _on_body_entered(body: Node3D) -> void:
    if body.name == "Player":
        collected.emit()
        queue_free()
```

Notice that the spinning happens on the child `Mesh`, not on the `Area3D` itself. The collision shape stays still while the visible gem spins and bobs. This keeps the collision reliable.

Step 4: The world

Create a new scene with a `Node3D` root named `Main`, and save it as `main.tscn`. Build this tree:

```
Main (Node3D)                    # script: main.gd
|-- WorldEnvironment
|-- Sun (DirectionalLight3D)
|-- Floor (StaticBody3D)         # position (0, -0.25, 0)
|   |-- Mesh (MeshInstance3D)    # BoxMesh, size (30, 0.5, 30), green material
|   `-- CollisionShape3D         # BoxShape3D, size (30, 0.5, 30)
|-- Wall (StaticBody3D)          # position (0, 1, -5)
|   |-- Mesh (MeshInstance3D)    # BoxMesh, size (6, 2, 1), brown material
|   `-- CollisionShape3D         # BoxShape3D, size (6, 2, 1)
|-- Step (StaticBody3D)          # position (6, 0.5, 3)
|   |-- Mesh (MeshInstance3D)    # BoxMesh, size (3, 1, 3), brown material
|   `-- CollisionShape3D         # BoxShape3D, size (3, 1, 3)
|-- Player                       # an instance of player.tscn, position (0, 1.1, 5)
|-- Gems (Node3D)                # empty; main.gd fills it
`-- HUD (CanvasLayer)
    |-- ScoreLabel (Label)       # text "Gems: 0 / 8", near the top left
    `-- MessageLabel (Label)     # empty text, below the score
```

To add the `Player` instance, drag `player.tscn` from the FileSystem dock onto `Main` in the Scene dock.

Set up the details:

- `WorldEnvironment`: a new `Environment` with `Background > Mode` set to **Sky**, a new `Sky` with a `ProceduralSkyMaterial`, and `Ambient Light > Source` set to **Sky**.
- `Sun`: rotation (-50, 30, 0) degrees, with `Shadow > Enabled` ticked.
- Each wall's mesh and collision shape must have the same size. The floor's top surface is at y = 0, because its center is at -0.25 and it is 0.5 thick.
- The labels live in a `CanvasLayer` so they stay fixed on the screen. You saw this in Chapter 14.

Now the main script:

```gdscript
# File: main.gd
# Attach to: Main (Node3D) in main.tscn
extends Node3D

const GEM_SCENE: PackedScene = preload("res://gem.tscn")
const GEM_COUNT: int = 8

@onready var player: CharacterBody3D = $Player
@onready var gems: Node3D = $Gems
@onready var score_label: Label = $HUD/ScoreLabel
@onready var message_label: Label = $HUD/MessageLabel

var _start_position: Vector3
var _collected: int = 0
var _finished: bool = false


func _ready() -> void:
    _start_position = player.global_position
    message_label.text = ""
    _spawn_gems()
    _update_score()


func _spawn_gems() -> void:
    var rng := RandomNumberGenerator.new()
    rng.randomize()
    for i in GEM_COUNT:
        var gem: Area3D = GEM_SCENE.instantiate()
        gems.add_child(gem)
        gem.position = Vector3(rng.randf_range(-11.0, 11.0), 1.0, rng.randf_range(-11.0, 11.0))
        gem.collected.connect(_on_gem_collected)


func _on_gem_collected() -> void:
```

```
    _collected += 1
    _update_score()
    if _collected >= GEM_COUNT:
        _finished = true
        message_label.text = "You found them all! Press Enter to play again."


func _update_score() -> void:
    score_label.text = "Gems: %d / %d" % [_collected, GEM_COUNT]


func _physics_process(_delta: float) -> void:
    # If Pip falls off the edge of the world, start again from the beginning.
    if player.global_position.y < -10.0:
        player.global_position = _start_position
        player.velocity = Vector3.ZERO


func _unhandled_input(event: InputEvent) -> void:
    if _finished and event.is_action_pressed("ui_accept"):
        get_tree().reload_current_scene()
```

A few things to notice:

- `gems.add_child(gem)` comes *before* setting `gem.position`. That is not required, but setting the position after the gem is in the tree is a habit that avoids surprises with global positions.
- Gems are created at random spots with `RandomNumberGenerator`. A gem might land inside the wall; that is a fun puzzle and also an invitation to improve the code in the exercises.
- Falling off the 30-by-30 floor puts Pip back at the start.

Step 5: Set the main scene and play

Open **Project > Project Settings > Application > Run** and set **Main Scene** to `main.tscn` (or press F5 and click Select Current). Press **F5** and try:

1. Press **W** to walk forward. Pip turns to face the way he walks.
2. Press **Q** and **E** and watch the camera orbit around Pip.
3. Walk toward the camera-forward direction after turning the camera. "Forward" is still away from the camera.
4. Press **Space** to jump onto the step.
5. Collect all eight gems. The message appears; press **Enter** to restart.

Expected result

A window shows a blue sky and a green field. Pip walks, turns, jumps, and is stopped by the wall. The camera follows and orbits. Each gem you touch disappears and the score at the top left increases. When the eighth is collected, the message appears, and Enter starts over.

Extra: the showcase scene

Create `showcase.tscn` as in the project folder: a `Node3D` root with `WorldEnvironment` (a plain dark `Background > Mode: Custom Color`), a `Camera3D` at (0, 1.5, 8), a `DirectionalLight3D`, an `OmniLight3D` named `Lamp`, and a `SpotLight3D` named `Spot`. Attach `showcase.gd` from the project folder. Press F6 to run it, and press 1, 2, and 3 to switch the three lights off and on.

Chapter summary

- 3D space has three axes. In Godot, X is right, Y is up, and negative Z is forward. One unit is about one meter.
- `Node3D` gives every 3D node a position, a rotation (in radians; yaw is rotation around Y), and a scale.
- A `Mesh` is a shape resource; a `MeshInstance3D` places it in the world. Primitive meshes (box, sphere, capsule, cylinder, plane) need no outside files.
- A `Camera3D` is the player's eye. FOV sets how wide it sees, and `look_at` aims it at a point.
- Lights make 3D readable: `DirectionalLight3D` is the sun, `OmniLight3D` a bulb, `SpotLight3D` a flashlight. A `WorldEnvironment` adds sky and ambient light.
- `StandardMaterial3D` controls color (albedo), shininess (roughness), metal (metallic), and glow (emission). Materials are resources and can be shared.
- `CharacterBody3D` moves like its 2D cousin, with a `Vector3` velocity. Gravity pulls on Y, and `basis * vector` makes movement relative to the camera.
- An `Area3D` plus a signal makes a simple pickup.

Review questions

1. In Godot's 3D space, which axis points up, and which direction is "forward"?
2. True or false: a `MeshInstance3D` and a `Mesh` are the same thing.
3. A box mesh is 2 units tall and its node sits at `position.y = 0`, on a floor whose top surface is at y = 0. How deep does the box sink into the floor, and what `y` would make it rest exactly on the floor?
4. Which light is best for the sun in an outdoor scene, and why does moving it not change the lighting?
5. Which material property should you raise to make a surface look more like a polished mirror? (a) Roughness (b) Metallic and lower Roughness (c) Emission (d) Scale
6. Your scene has meshes and lights, but when you run the game the window is empty. Name the likely missing node.
7. Why does the project store Pip's meshes under a `Body` node and turn `Body` instead of turning `Player`?
8. What does `lerp_angle(a, b, t)` do that plain `lerp` does not?
9. True or false: changing the `Size` of a `BoxMesh` automatically makes its `BoxShape3D` the same size.
10. You press the "forward" key and Pip walks along negative Z when the camera has not turned. After the camera pivot has turned 90 degrees, in which direction is "forward," and which line of code handles this?

Exercises

1. **Practice (easy):** In `showcase.gd`, change the loop so that it creates seven spheres and changes the `albedo_color` for each one using `Color.from_hsv(i / 7.0, 0.8, 0.9)`. Run the scene and describe what you see.
2. **Practice (easy):** In `main.tscn`, change the camera's FOV to 50, then to 100. Run the game with each. Which is easier to play, and why?
3. **Apply (medium):** Add a second wall and a tower (a `CylinderMesh` with a matching `CylinderShape3D`). Add a `SpotLight3D` above the step with a warm color and shadows turned on.
4. **Apply (medium):** Add a "sprint" action on Shift. When it is held, Pip should move at double speed. You will need a new input action and one `if` in `player.gd`.
5. **Challenge (hard):** Make gems avoid the wall and the step: in `_spawn_gems`, pick a random position and, if it lies inside the wall's footprint (x between -3 and 3, z between -5.5 and -4.5), pick again. Hint: use a `while` loop. Then add a timer that counts up in the HUD and shows your best time when you win.

Common mistakes

- **My character falls through the floor.** The floor has no `CollisionShape3D`, or the shape has no shape resource. *Fix:* add the collision shape child and give it a `BoxShape3D`, then check there is no yellow warning triangle.
- **The scene is completely black.** There is no light, or the light's energy is 0, or there is no environment. *Fix:* add a `DirectionalLight3D` and a `WorldEnvironment` with a sky or ambient color.
- **The game window is blank gray.** There is no current camera. *Fix:* add a `Camera3D` and tick its **Current** box.
- **Pip walks sideways when the camera turns.** The movement does not use the camera's basis. *Fix:* multiply the input by `camera_pivot.global_transform.basis`, as in step 4 of the player script.
- **The camera spins when Pip turns.** The camera is under a node that rotates with Pip. *Fix:* turn only the `Body` child, not the whole `Player`.
- **A mesh shows up at the wrong place or twice the size.** The mesh is parented to a node that has a scale or offset. *Fix:* check each parent's Transform in the Inspector and reset it to position 0, scale 1.

Further reading

- Introduction to 3D: https://docs.godotengine.org/en/stable/tutorials/3d/introduction_to_3d.html
- Using Transform: https://docs.godotengine.org/en/stable/tutorials/3d/using_transforms.html
- 3D lights and shadows: https://docs.godotengine.org/en/stable/tutorials/3d/lights_and_shadows.html
- Environment and post-processing: https://docs.godotengine.org/en/stable/tutorials/3d/environment_and_post_processing.html
- StandardMaterial3D: https://docs.godotengine.org/en/stable/tutorials/3d/standard_material_3d.html
- Using CharacterBody2D/3D (the same page covers both): https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html

Chapter 20: Particles, Shaders, and Game Feel

Learning objectives

By the end of this chapter you will be able to:

- Explain what "juice" (game feel) is and add small feedback effects that make actions feel good.
- Create one-shot and continuous particle effects with `GPUParticles2D` and `ParticleProcessMaterial`.
- Shake the screen with a trauma-based `Camera2D` script.
- Write simple `canvas_item` shaders with uniforms, and change their values from GDScript.
- Build a hit flash, a dissolve effect, and an animated background using shaders.
- Use tweens, hit stop, and squash-and-stretch together, and decide when an effect is too much.

Key terms

- **Juice**: small extra feedback (movement, color, sound, particles) that makes a game feel alive and responsive, without changing the rules.
- **Game feel**: how it feels to control the game. Responsive, weighty, satisfying, or sluggish.
- **Particle**: a tiny short-lived sprite, one of hundreds that together make smoke, sparks, dust, or magic.
- **Emitter**: the node that creates particles and sets how many, how fast, and in which direction.
- **Process material**: the resource (`ParticleProcessMaterial`) that holds the rules for how each particle moves and changes.
- **Trauma**: a number from 0 to 1 that says how much the screen should shake right now. It fades back to 0 on its own.
- **Hit stop**: freezing or slowing the game for a split second when something is hit, to make the hit feel heavy.
- **Shader**: a small program that runs on the graphics chip and decides the color of every dot (pixel) that is drawn.
- **Fragment function**: the part of a shader that runs once for every pixel of an object and outputs its color.
- **Uniform**: a setting declared at the top of a shader that you can change from the editor or from GDScript.
- **UV**: coordinates across a picture, from (0, 0) at the top left to (1, 1) at the bottom right, independent of the picture's pixel size.
- **Squash and stretch**: briefly flattening or stretching an object to suggest weight and speed.

20.1 What is juice?

Imagine two versions of the same game. In both, you press a button, a sword hits a slime, and the slime's health goes from 5 to 4. The rules are identical.

In version A, the number changes. Nothing else happens.

In version B, the slime flashes white for an instant, squashes flat and bounces back, a handful of sparks fly out, a "-1" floats up, the screen jolts a little, and for one sixth of a second the whole game seems to freeze on the hit.

Version B feels better to play, even though the *rules* have not changed at all. The difference is called **juice**. Game designers use the word because a juicy game "squirts" feedback every time you touch it, like squeezing an orange.

Juice does not change what your game *is*. It changes how it *feels*. It has three jobs:

1. **Confirm.** "Yes, your button press did something." Players need to know an action worked.
2. **Communicate.** "That was a big hit," or "that was a weak one." Bigger effects mean bigger events.
3. **Delight.** Small surprises make the game a pleasure to touch.

The good news is that most juice is cheap. This chapter uses six simple tools:

Tool	What it adds	Where it is covered
Tweens	Squash and stretch, floating numbers, fades	Section 20.2
Particles	Sparks, dust, smoke, magic	Section 20.3
Screen shake	A sense of impact	Section 20.4
Hit stop	Weight on each hit	Section 20.5
Shader flash	A bright "I was hit" signal	Section 20.6
Shader dissolve and backgrounds	Polished transitions and mood	Section 20.6

We will build all of them in one small "Juice Lab" project, where you can switch each effect on and off with the number keys and feel the difference yourself.

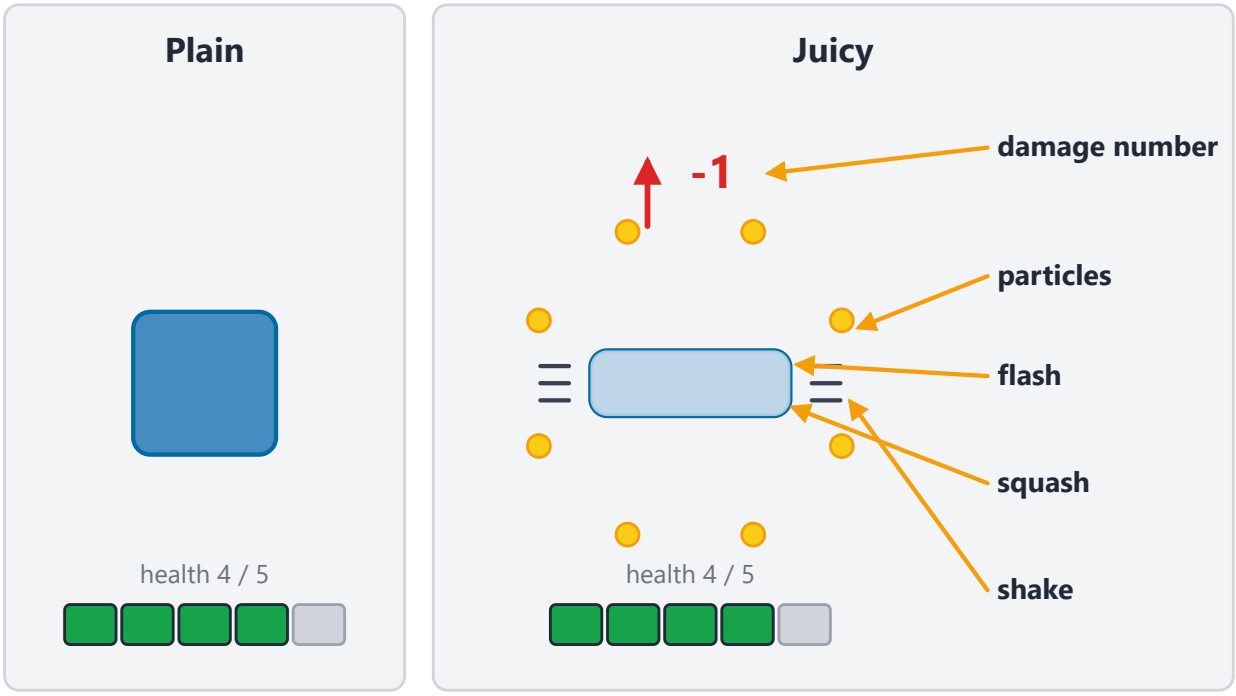


Figure 20.1: Same rules, different feeling. The right side adds a squash, a flash, sparks, a floating number, and a shake.

> **Warning:** Juice can be overdone. If everything shakes, flashes, and explodes all the time, nothing stands out and the player gets tired or even dizzy. Good juice is *proportional*: small events get small effects, big events get big ones. We come back to this in section 20.7.

20.2 Tweens: squash, stretch, and floating numbers

You met the `Tween` in Chapter 10. A tween smoothly changes a property from one value to another over time. It is the most useful tool in your juice kit, so here is a short refresher followed by new uses.

Creating a tween and animating a property looks like this:

```
# Fragment: inside any script on a node with a "scale" property.
var tween: Tween = create_tween()
tween.tween_property(self, "scale", Vector2(1.5, 1.5), 0.3)
```

That means: "Change my `scale` to (1.5, 1.5) over 0.3 seconds." The tween runs by itself in the background; your code continues straight away.

Easing and transitions

By default a tween changes at a steady speed, which looks robotic. Two settings shape the motion:

- `set_trans` chooses the *curve*: `TRANS_LINEAR` (steady), `TRANS_QUAD` and `TRANS_CUBIC` (smooth), `TRANS_BACK` (overshoots a little, then settles), `TRANS_ELASTIC` (wobbles like a spring), `TRANS_BOUNCE` (bounces like a ball).
- `set_ease` chooses *where* the curve is fast: `EASE_IN` (slow start), `EASE_OUT` (slow end), `EASE_IN_OUT` (slow at both ends).

For juice, `EASE_OUT` is the one you use most, because actions in the real world start fast and settle down.

Squash and stretch

Animators have used squash and stretch for a hundred years. When something is hit, it deforms for a moment and springs back. In code, we jump the scale to a squashed shape, then tween back to normal with an elastic curve. This is the `squash()` function of the Juice Lab target:

```
# Fragment from target.gd (the Target is a Sprite2D whose normal scale is 2, 2).
func squash() -> void:
    if _tween:
        _tween.kill()   # stop any older squash that is still running
    scale = Vector2(2.6, 1.5)  # wide and flat, like it was punched
    _tween = create_tween()
    _tween.set_trans(Tween.TRANS_ELASTIC).set_ease(Tween.EASE_OUT)
    _tween.tween_property(self, "scale", Vector2(2.0, 2.0), 0.45)
```

The line `_tween.kill()` is important. If the player hits the target twice quickly, the first tween is still changing the scale. Two tweens fighting over one property cause jumpy results, so we stop the old one first.

Floating damage numbers

Numbers that rise and fade away are a classic. We make a `Label` in code, then run two tweens at once, one for position and one for transparency. `set_parallel(true)` tells a tween to run its steps at the same time instead of one after another:

```
# Fragment from main.gd.
func _spawn_damage_number(where: Vector2) -> void:
    var label := Label.new()
    label.text = "-1"
    label.add_theme_font_size_override("font_size", 36)
    label.add_theme_color_override("font_color", Color(1.0, 0.9, 0.4))
    label.add_theme_color_override("font_outline_color", Color(0.1, 0.0, 0.1))
    label.add_theme_constant_override("outline_size", 8)
    label.position = where
    effects.add_child(label)

    var tween: Tween = create_tween().set_parallel(true)
```

```
    tween.tween_property(label, "position:y", where.y - 70.0, 0.7)
    tween.tween_property(label, "modulate:a", 0.0, 0.7)
    tween.chain().tween_callback(label.queue_free)
```

Look at the last line. `chain()` says "stop running in parallel; wait for everything so far to finish, then do the next step." The next step, `tween_callback`, calls `label.queue_free()` so the label removes itself. Without it, hundreds of invisible labels would pile up in the scene. The property path `"position:y"` means "only the y part of position," and `"modulate:a"` means "only the alpha (transparency) part of the color."

> **Why it works:** Labels, sprites, and all other `CanvasItem` nodes have a `modulate` property: a color that multiplies everything the node draws. Its fourth number, alpha, is how see-through the node is. Tweening `modulate:a` from 1 to 0 is the simplest fade-out in Godot.

20.3 Particles

A **particle** is a tiny, short-lived object. One spark is boring; 30 sparks flying out in all directions is an explosion. Particle systems create these automatically from a few rules, so you do not write code for each one.

In Godot 2D, the node is `GPUParticles2D`. The "GPU" in the name means the particles are calculated on the graphics chip, which can handle thousands without slowing the game. There is also `CPUParticles2D`, which does the same work on the main processor. It has nearly the same properties, works on very old devices, and you can convert one to the other with the Convert option in the editor. Use `GPUParticles2D` by default.

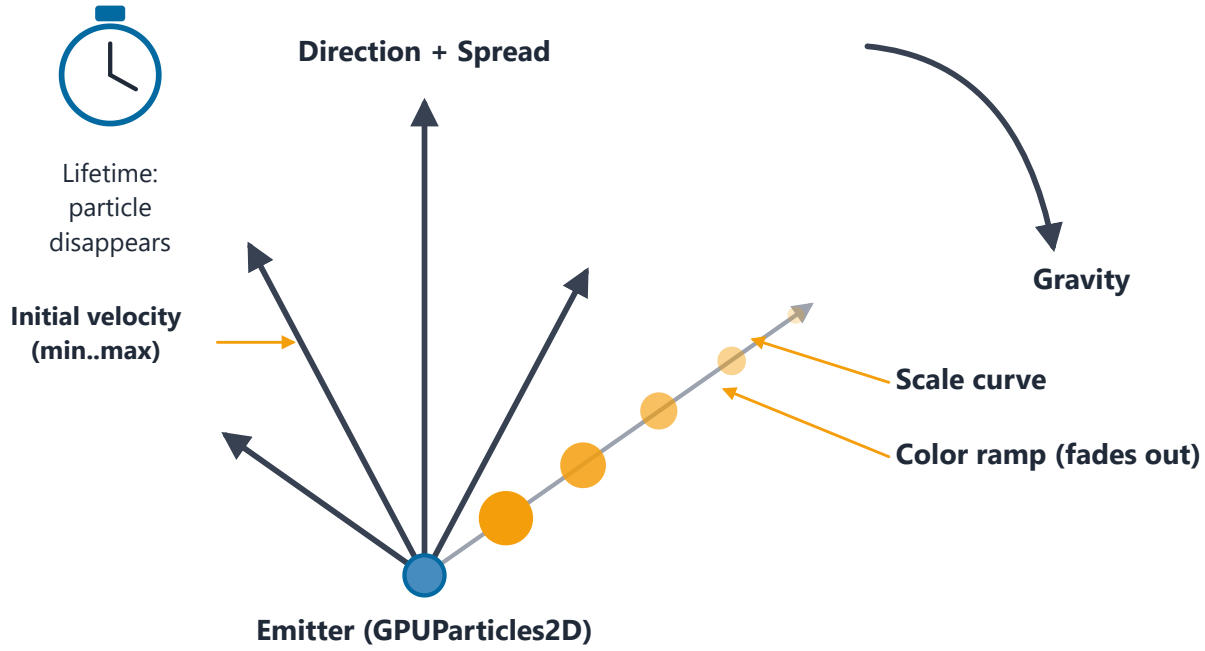


Figure 20.2: How a burst works. The emitter spawns particles, each one flies out with a random speed, is pulled down by gravity, fades, and disappears when its lifetime ends.

Two nodes, two jobs

A particle effect needs two parts, and it is easy to mix them up:

- The `GPUParticles2D` **node** controls *how many* particles, *how long* they live, and *how* they are emitted: `Amount`, `Lifetime`, `One Shot`, `Explosiveness`, and the `Texture` each particle draws.
- The `ParticleProcessMaterial`, in the node's `Process Material` property, controls *how each particle moves and changes*: direction, speed, gravity, size, and color over time.

The properties that matter

On the `GPUParticles2D` node:

- `Emitting`: on means particles are being made right now.
- `Amount`: the number of particles alive at the same time.
- `Lifetime`: how many seconds each particle lives.
- `One Shot`: emit only one batch and stop. Use this for bursts (a hit, a coin pickup). Leave it off for steady effects (smoke, dust).
- `Explosiveness`: 0 spreads the births evenly over the lifetime (a steady stream); 1 makes them all appear at the very same moment (a burst).
- `Texture`: the picture each particle shows. With no texture, each particle is a plain tiny square.
- `Preprocess`: runs the effect forward this many seconds before the game starts, so a steady effect such as dust is already in full swing on the first frame.
- `Local Coords`: when off (the default), particles stay where they were born even if the emitter moves afterward, which is what you want for trails and smoke.

On the `ParticleProcessMaterial`:

- `Direction` and `Spread`: the direction particles are shot in, and how wide the fan is, in degrees. A spread of 180 shoots in every direction around the direction.
- `Initial Velocity` (min and max): the starting speed. Each particle picks a random value between the two, which keeps the burst from looking mechanical.
- `Gravity`: a constant pull. `Vector3(0, 600, 0)` pulls particles down the screen, because 2D Y points down. Set it to zero for floating effects.
- `Scale` (min and max) and `Scale Curve`: how big particles start, and how their size changes over their life.
- `Color` and `Color Ramp`: the starting color, and a gradient the color follows over the particle's life. A ramp from white to see-through white makes a particle fade out.
- `Damping`: slows particles down over time, so they burst out and then drift to a stop.
- `Emission Shape`: where particles are born: a single point (default), a sphere or circle, a box, or a ring.

Building the hit burst

Here is how to build the burst effect used in the Juice Lab, step by step in the editor:

1. Create a new scene with a `GPUParticles2D` root named `Burst`. Save it as `burst.tscn`.
2. Set `Amount` to 28, `Lifetime` to 0.7, tick `One Shot`, and set `Explosiveness` to 1.
3. Untick `Emitting` (the script starts it when it is needed).
4. In the `Process Material` property, choose **New ParticleProcessMaterial**, then open it.
5. Set `Direction` to (0, -1, 0), and `Spread` to 180. In `Initial Velocity`, set Min to 150 and Max to 380.
6. Under `Gravity`, set Y to 600. Under `Damping`, set Min 30 and Max 60.
7. Under `Scale`, set Min 0.6 and Max 1.4. Choose **New CurveTexture** for the `Scale Curve`, and drag the right end of the curve down, so particles shrink as they age.
8. Under `Color`, find `Color Ramp` and choose **New GradientTexture1D**. Edit its gradient so it goes from white to transparent white. This fades each particle out.

9. For the node's `Texture`, choose **New GradientTexture2D**, set its width and height to 16, set `Fill` to **Radial**, and edit its gradient from white to transparent. This gives you soft round dots instead of hard squares.

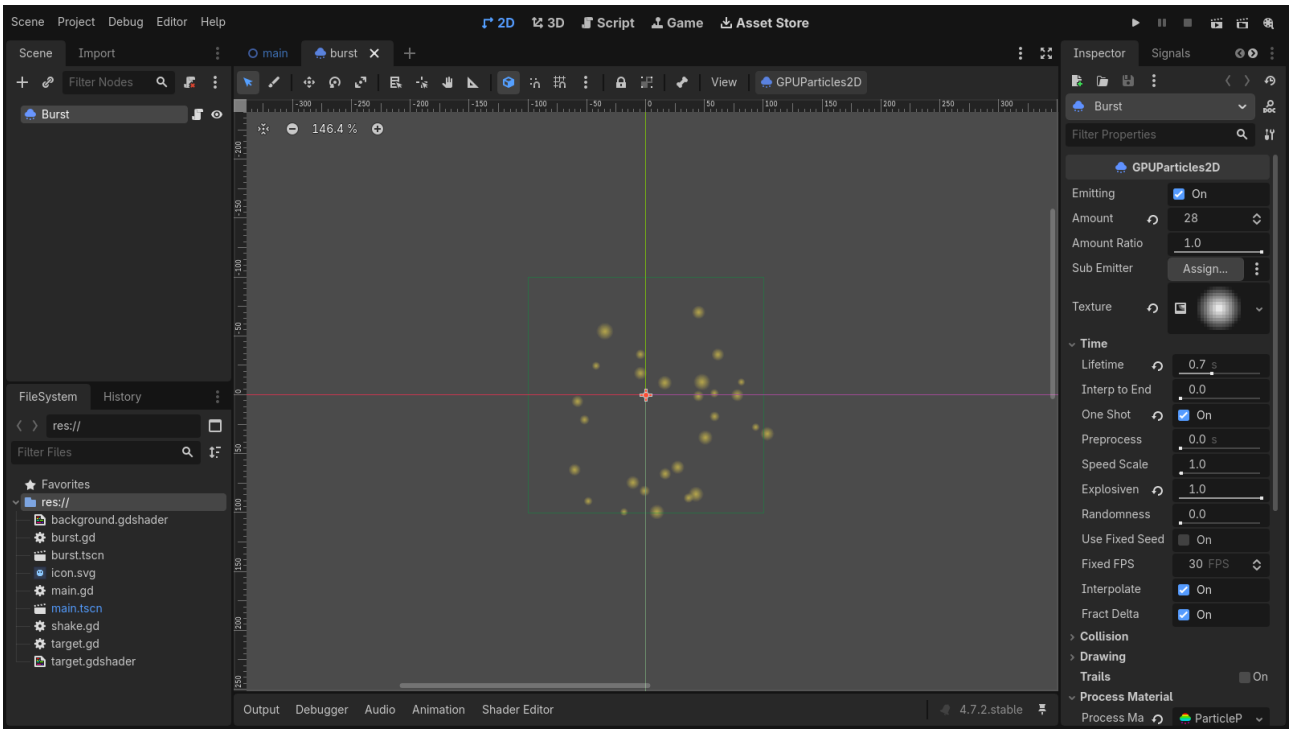


Figure 20.3: Editing the Burst scene. With the node selected, the editor shows a Particles toolbar at the top of the viewport; its Restart button replays the burst.

Now the script. A one-shot particle node should clean itself up when it finishes, or finished bursts would pile up in the scene tree. The node has a `finished` signal that fires when a one-shot effect is done:

```gdscript
# File: burst.gd
# Attach to: Burst (GPUParticles2D) at the root of burst.tscn
extends GPUParticles2D


func _ready() -> void:
    finished.connect(queue_free)  # clean up when the last particle dies
    emitting = true
```

To use the burst, instance it where you need it. Here is how the Juice Lab spawns one, with a tint color:

```gdscript
# Fragment from main.gd.
const BURST_SCENE: PackedScene = preload("res://burst.tscn")

func _spawn_burst(where: Vector2, color: Color) -> void:
    var burst: GPUParticles2D = BURST_SCENE.instantiate()
    burst.position = where
    burst.modulate = color      # tints the white dots
    effects.add_child(burst)
```

Setting `position` *before* `add_child` means the burst is already at the right spot on the first frame it runs. The white dots are tinted with `modulate`, so one scene can make yellow sparks, orange sparks, or green leaves.

> **Try it:** Change the process material in `burst.tscn`: set `Gravity` to (0, 0, 0) and `Spread` to 20. The burst is no longer an explosion; it is a jet. Now try a negative gravity of (0, -200, 0) for rising bubbles.

Continuous effects: floating dust

For steady effects, leave `One Shot` off and use a long lifetime. The Juice Lab has a layer of slow dust floating upward along the bottom of the screen. The settings that make it work are an `Emission Shape` of **Box** (so particles are born along a wide line instead of one point), `Direction` upward with a small spread, slow speed, no gravity, and a `Preprocess` of 9 seconds so the room is already dusty when you start. A `Visibility Rect` (shown as a rectangle in the editor) tells the engine where the particles can be, so it can stop drawing them when they are off-screen. If your particles disappear when the emitter moves off-screen, make that rectangle bigger.

> **Note:** `CPUParticles2D` is the right choice when you need an effect on a very old device, or on the web where GPU particles are limited. In the editor, select a `GPUParticles2D` and use **Convert to CPUParticles2D** from the viewport's Particles menu.

20.4 Screen shake

Few effects add as much impact for as little work as a shake. When the screen jolts, the player feels the hit in their stomach.

The simplest shake moves the camera by a random amount every frame for a while. The problem is how to make it fade nicely and stack when several hits happen quickly. The widely used answer is the **trauma** model:

- Keep one number, `trauma`, between 0 and 1.
- Every hit adds to trauma (but it can never go above 1).
- Every frame, trauma shrinks a little, and the shake strength is computed *from* trauma.
- Use the **square** of trauma for the strength. Small trauma then gives a barely visible tremble, while large trauma gives a violent shake.

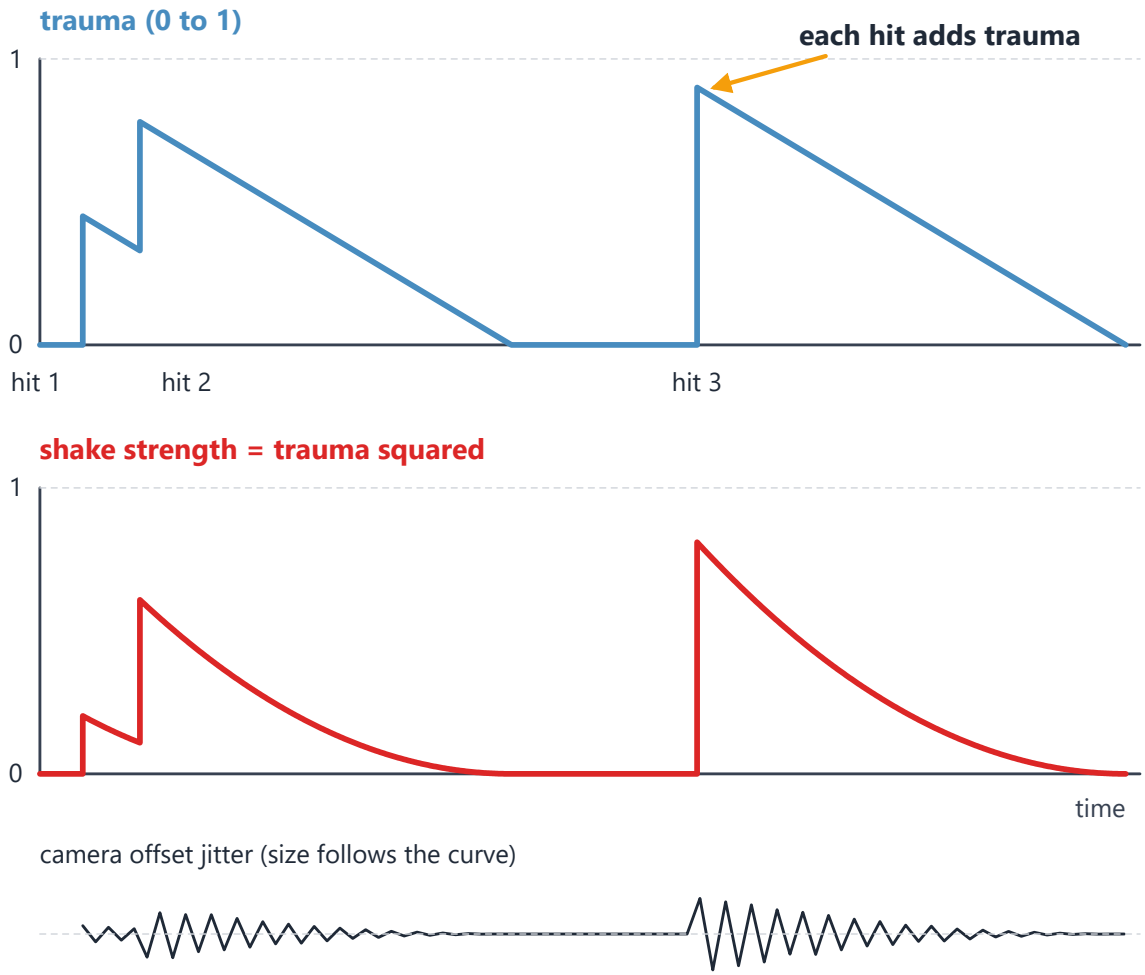


Figure 20.4: Each hit adds a jump of trauma, which then drains away. Squaring it makes the shake drop off faster at the end, so it finishes cleanly rather than buzzing.

Here is the complete script. Attach it to a `Camera2D`:

```
# File: shake.gd
# Attach to: Camera2D
extends Camera2D

@export var max_offset: Vector2 = Vector2(24.0, 16.0)
@export var max_roll: float = 0.06  # radians
@export var decay: float = 1.6  # trauma lost per second
@export var enabled_shake: bool = true

var trauma: float = 0.0


func add_trauma(amount: float) -> void:
    if not enabled_shake:
        return
    trauma = minf(trauma + amount, 1.0)


func _process(delta: float) -> void:
    if trauma > 0.0:
        trauma = maxf(trauma - decay * delta, 0.0)
        var shake: float = trauma * trauma  # squaring makes small shakes subtle
        offset = Vector2(
            randf_range(-1.0, 1.0) * max_offset.x * shake,
            randf_range(-1.0, 1.0) * max_offset.y * shake
        )
```

```
        rotation = randf_range(-1.0, 1.0) * max_roll * shake
    else:
        offset = Vector2.ZERO
        rotation = 0.0
```

How it works, line by line:

- `add_trauma` is the only function other scripts call. A small hit might add 0.3; a boss crash, 1.0. `minf` picks the smaller of two numbers, so trauma is capped at 1.
- In `_process`, trauma drops by `decay * delta` each frame. With `decay = 1.6`, a full trauma of 1 takes about 0.6 seconds to disappear.
- `maxf(..., 0.0)` stops it going below zero.
- The shake offset is a random value between -1 and 1, multiplied by the maximum allowed distance and by the shake strength.
- `offset` is a built-in `Camera2D` property that shifts the view without moving the camera node itself. That is why the camera can return to rest simply by resetting it to zero.
- A little rotation (the "roll") adds to the feeling without being large. The default `max_roll` is about 3.4 degrees.

To use it from another script, call `camera.add_trauma(0.45)` when a normal hit lands, and `camera.add_trauma(0.9)` for a big one.

> **Warning:** Screen shake can make some players feel unwell. Many professional games offer a setting to reduce or disable it. The `enabled_shake` property exists for this reason; in Chapter 17 you learned to save settings, so you could store a "screen shake" option in a `ConfigFile`. Always consider adding the option.

The camera must exist

`Camera2D` shake only works if the camera is the **current** camera. A new `Camera2D` is current automatically when it is the only one. If you place the camera at the center of your screen (for a 960 by 540 window, at position (480, 270)), the picture looks exactly as it did before the camera was added.

20.5 Hit stop, flash, and the clock

Hit stop

Fighting games, action games, and platformers often *freeze* the game for a few frames at the moment of an impact. This is **hit stop**. The brain reads the pause as "that hit was heavy."

Godot has a global speed control for the game clock: `Engine.time_scale`. A value of 1.0 is normal speed, 0.5 is half speed, and 0.0 stops almost everything that depends on delta time. We use a very small value rather than zero so animations and tweens still creep forward:

```
# Fragment from main.gd.
func _hit_stop(seconds: float) -> void:
    Engine.time_scale = 0.05
    await get_tree().create_timer(seconds, true, false, true).timeout
    Engine.time_scale = 1.0
```

There is a trap here. `create_timer` makes a timer that normally also obeys the time scale. If the game runs at 5 percent speed, a timer of 0.06 seconds would take 1.2 seconds in real life! The fourth argument of `create_timer`, `ignore_time_scale`, set to `true`, makes the timer count real time. The arguments, in order, are the wait time, whether it works while the game is paused, whether it counts physics frames, and whether to ignore the time scale. So `(seconds, true, false, true)` means "wait this long, in real time."

> **Warning:** Always restore `Engine.time_scale = 1.0`. If an error stops the function after the first line, your whole game stays in slow motion, even after you restart the scene. If a game feels mysteriously slow, check this first.

Keep hit stops very short: 0.04 to 0.08 seconds for a normal hit, up to 0.15 for a finishing blow.

Flashing on a hit

The oldest hit signal in games is a flash of white. There are two ways to do it:

1. **Quick and easy:** tween the node's `modulate` color. Setting `modulate` to a value greater than 1 (for example `Color(3, 3, 3)`) brightens the node. This works on any `CanvasItem` with no shader.
2. **A shader:** paint all visible pixels one color, so the object becomes a solid white silhouette. This looks sharper and is covered next.

20.6 Shaders

What is a shader?

Until now, you told Godot what to draw by using nodes and properties. A **shader** goes a level deeper. It is a short program that runs on the graphics chip and answers one question for every dot on the object: **"What color is this pixel?"**

Why use a program for that? Because the graphics chip has thousands of tiny cores and runs the same little program for all the pixels at once. It is extremely fast, and it makes possible effects that nodes alone cannot do: flashing a silhouette, dissolving, waving, outlines, water, and so on.

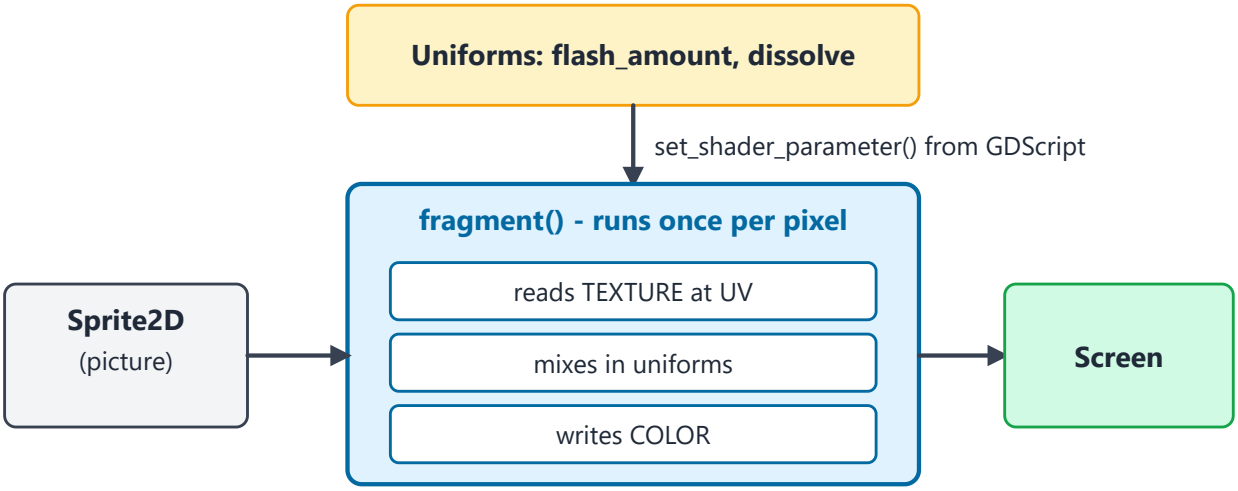


Figure 20.5: A shader is a function applied to every pixel. It reads the original picture (TEXTURE) at a position (UV), may use settings (uniforms), and writes the final color (COLOR).

Your first canvas_item shader

Shaders are written in Godot's shading language, which looks like a cousin of GDScript and C. They live in `.gdshader` files. The first line says what kind of shader it is. For 2D, it is always:

```
shader_type canvas_item;
```

The kinds are `canvas_item` (2D and UI), `spatial` (3D, as in the small taste in Chapter 19), `particles`, `sky`, and `fog`. Here is the smallest useful `canvas_item` shader. It turns a picture into its photo-negative (every color flipped):

```
// File: negative.gdshader
shader_type canvas_item;

void fragment() {
    vec4 tex = texture(TEXTURE, UV);
    COLOR = vec4(1.0 - tex.rgb, tex.a);
}
```

Let's read it:

- `void fragment() { ... }` is the function that runs for every pixel.
- `UV` is the position in the picture, from (0, 0) to (1, 1).
- `TEXTURE` is the node's own picture. `texture(TEXTURE, UV)` reads the color at that position.
- `vec4` is a group of four numbers: red, green, blue, and alpha (see-through-ness). `tex.rgb` is the first three, and `tex.a` is the last.
- `COLOR` is the output. Whatever you put in it is what appears.
- Every line ends with a semicolon `;`, and floating-point numbers need a decimal point: write `1.0`, not `1`.

> **Warning:** The shading language is **not** GDScript. There is no `var`, no `:=`, and no indentation rule. Types come first (`float speed = 2.0;`). Numbers that hold decimals must be written with a decimal point. If you see the error "Invalid arguments for the built-in function", you probably used `1` where the function wanted `1.0`.

To use a shader: select a `Sprite2D` (or a `ColorRect`, or any other `CanvasItem`), open `Material` in the Inspector, choose **New ShaderMaterial**, open it, and under `Shader` choose **Load** to pick your `.gdshader` file (or **New Shader** to write one inline). The shader editor opens at the bottom of the screen, and compile errors appear below the code.

Uniforms: settings you can change

A `uniform` is a variable declared outside any function. It becomes a property you can edit in the Inspector and change from GDScript:

```
uniform float flash_amount : hint_range(0.0, 1.0) = 0.0;
uniform vec4 flash_color : source_color = vec4(1.0, 1.0, 1.0, 1.0);
```

- `hint_range(0.0, 1.0)` makes the Inspector show a slider.
- `source_color` tells the Inspector that this is a color and should show a color picker.
- After the `=` comes the default value.

From GDScript, you change it with `set_shader_parameter`:

```gdscript
# Fragment: change a shader setting from a script on the node that has the material.
var shader_material: ShaderMaterial = material as ShaderMaterial
shader_material.set_shader_parameter("flash_amount", 1.0)
```

A tween can animate it, too, using the special path `shader_parameter/name`:

```gdscript
# Fragment: fade the flash back to 0 over a tenth of a second.
var tween: Tween = create_tween()
tween.tween_property(shader_material, "shader_parameter/flash_amount", 0.0, 0.1)
```

The hit flash and dissolve shader

Here is the complete `target.gdshader`. It contains two effects in one: a **flash** (paint everything one color) and a **dissolve** (the picture crumbles away through random holes, with a glowing edge). Read the whole thing first; then we explain each part.

```
// File: target.gdshader
shader_type canvas_item;

// Flash: 0 shows the normal picture, 1 paints it all one colour.
uniform vec4 flash_color : source_color = vec4(1.0, 1.0, 1.0, 1.0);
uniform float flash_amount : hint_range(0.0, 1.0) = 0.0;

// Dissolve: 0 shows everything, 1 shows nothing.
uniform float dissolve : hint_range(0.0, 1.0) = 0.0;
uniform vec4 edge_color : source_color = vec4(1.0, 0.6, 0.1, 1.0);
uniform float edge_width = 0.08;

float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453);
}

float value_noise(vec2 p) {
    vec2 cell = floor(p);
    vec2 f = fract(p);
    f = f * f * (3.0 - 2.0 * f); // smooth the blending
    float a = hash(cell);
    float b = hash(cell + vec2(1.0, 0.0));
    float c = hash(cell + vec2(0.0, 1.0));
    float d = hash(cell + vec2(1.0, 1.0));
    return mix(mix(a, b, f.x), mix(c, d, f.x), f.y);
}

void fragment() {
    vec4 tex = texture(TEXTURE, UV);
    float n = value_noise(UV * 10.0);

    float threshold = dissolve * (1.0 + edge_width);
    float alive = step(threshold, n);
    float edge = 1.0 - smoothstep(threshold, threshold + edge_width, n);

    vec3 rgb = mix(tex.rgb, flash_color.rgb, flash_amount);
    rgb = mix(rgb, edge_color.rgb, edge * step(0.001, dissolve));

    COLOR = vec4(rgb, tex.a * alive);
}
```

The flash. `mix(a, b, t)` blends from `a` to `b`: `t = 0` gives `a`, `t = 1` gives `b`, and 0.5 gives the half-way blend. So `mix(tex.rgb, flash_color.rgb, flash_amount)` shows the normal picture when `flash_amount` is 0 and a solid white silhouette when it is 1. The alpha is kept as is, so the transparent corners of the picture stay transparent. This is why the shader flash beats the `modulate` trick: the silhouette is crisp.

The noise. For the dissolve, we need a random-looking pattern, with a different number at each place in the picture. `hash` turns a 2D position into a number between 0 and 1 that looks random but is always the same for the same position. It uses `sin`, `dot`, and `fract` (the part after the decimal point) to scramble the number. The `value_noise` function blends the hash values of the four corners of each grid cell, which turns harsh static into smooth blotchy patches. You do not need to understand the details; treat `value_noise` as a ready-made tool that gives a smooth random value between 0 and 1.

The dissolve. For each pixel we compute its noise value `n`. We also have a rising `threshold`. Where `n` is below the threshold the pixel is gone, where it is above the threshold it is still there. As `dissolve` rises from 0 to 1, the threshold sweeps upward, and more and more patches disappear, in a random order. `step(threshold, n)` returns 0 if `n` is smaller than the threshold and 1 otherwise. We multiply the pixel's alpha by it, so gone pixels become transparent.

The glowing edge. `smoothstep(a, b, x)` ramps smoothly from 0 to 1 as `x` goes from `a` to `b`. Pixels where `n` is just slightly above the threshold get an `edge` value close to 1, and we paint them orange. The result is a burning rim that travels across the picture. The final `step(0.001, dissolve)` prevents the orange edge from showing when no dissolve is happening.

The multiplier. `threshold = dissolve * (1.0 + edge_width)` makes sure that at `dissolve = 1` the threshold is above every possible noise value, so nothing is left; and at 0 the threshold is 0, so everything shows.

Here is the matching GDScript for the sprite. It uses all the techniques you just learned:

```gdscript
# File: target.gd
# Attach to: Target (Sprite2D) in main.tscn. The Sprite2D must have a ShaderMaterial using
target.gdshader.
extends Sprite2D

signal died

@export var max_hp: int = 5

var hp: int = 5
var _alive: bool = true
var _tween: Tween
var _material: ShaderMaterial


func _ready() -> void:
    hp = max_hp
    _material = material as ShaderMaterial


func is_alive() -> bool:
    return _alive


## Lose one hit point. Returns true if the target just died.
func take_hit() -> bool:
    hp -= 1
    if hp <= 0:
        _alive = false
        died.emit()
        return true
    return false


func flash(color: Color = Color.WHITE, duration: float = 0.12) -> void:
    _material.set_shader_parameter("flash_color", color)
    _material.set_shader_parameter("flash_amount", 1.0)
    var tween: Tween = create_tween()
    tween.tween_property(_material, "shader_parameter/flash_amount", 0.0, duration)
```

```gdscript
func squash() -> void:
    if _tween:
        _tween.kill()
    scale = Vector2(2.6, 1.5)  # wide and flat, like it was punched
    _tween = create_tween()
    _tween.set_trans(Tween.TRANS_ELASTIC).set_ease(Tween.EASE_OUT)
    _tween.tween_property(self, "scale", Vector2(2.0, 2.0), 0.45)


func dissolve_away(duration: float = 0.8) -> void:
    var tween: Tween = create_tween()
    tween.tween_property(_material, "shader_parameter/dissolve", 1.0, duration)


func respawn() -> void:
    hp = max_hp
    _alive = true
    _material.set_shader_parameter("dissolve", 0.0)
    _material.set_shader_parameter("flash_amount", 0.0)
    scale = Vector2(0.1, 0.1)
    var tween: Tween = create_tween()
    tween.set_trans(Tween.TRANS_BACK).set_ease(Tween.EASE_OUT)
    tween.tween_property(self, "scale", Vector2(2.0, 2.0), 0.4)
```

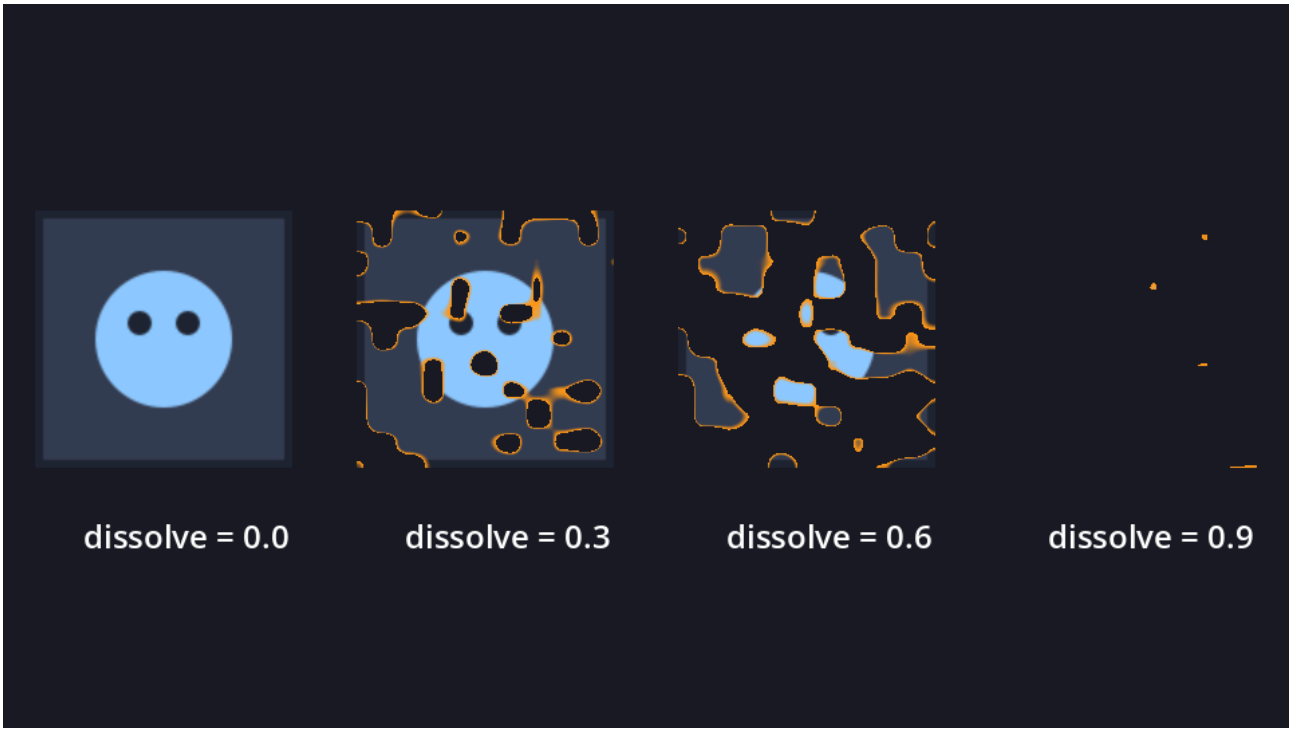


Figure 20.6: The dissolve shader at four settings. The orange edge appears where the picture is just about to vanish.

> **Warning:** If you give the same `ShaderMaterial` to several sprites, they share the uniforms: flashing one flashes all. For per-object effects, tick the material's **Resource > Local to Scene** option, or make the material unique in code (`material = material.duplicate()`).

A shader for the background

The second shader paints a slowly moving wave of dark purple behind everything. It needs no picture at all, so it can go on a plain `ColorRect`:

```
// File: background.gdshader
shader_type canvas_item;
```

```
uniform vec4 color_a : source_color = vec4(0.10, 0.08, 0.22, 1.0);
uniform vec4 color_b : source_color = vec4(0.22, 0.12, 0.38, 1.0);
uniform float speed : hint_range(0.0, 2.0) = 0.4;
uniform float bands = 4.0;

void fragment() {
    // A gentle sideways wobble makes the bands wave.
    float wave = sin(UV.x * 6.0 + TIME * speed * 2.0) * 0.04;
    float t = 0.5 + 0.5 * sin((UV.y + wave) * bands * 3.14159 - TIME * speed * 2.0);
    vec4 base = mix(color_a, color_b, t);

    // Darken the corners (a vignette).
    float d = distance(UV, vec2(0.5, 0.5));
    base.rgb *= 1.0 - smoothstep(0.3, 0.85, d) * 0.6;

    COLOR = base;
}
```

`TIME` is a built-in number of seconds since the game began, so the bands move. A `sin` wave goes between -1 and 1; multiplying by 0.5 and adding 0.5 moves it to the range 0 to 1, which is the right range for `mix`. `distance(UV, vec2(0.5, 0.5))` measures how far each pixel is from the center of the rectangle, and we darken pixels further out. That darkening of the corners is called a **vignette**, a classic touch.

> **Try it:** Change `bands` to 12 and `speed` to 1.5. Then change `color_b` to a dark green. Because they are uniforms, you can edit them right in the Inspector while the game is running (use the **Remote** tab of the Scene dock) and watch the result live.

Where to learn more shaders

The shader language has many more tools: `sin`, `cos`, `mix`, `smoothstep`, `clamp`, and `length` do most of the work. The docs have an excellent beginner series, linked in Further reading. For this book, flash, dissolve, and backgrounds are enough to begin.

20.7 Using juice well

You now have a toolbox. Here are some guidelines for using it.

Match effect size to event size. A footstep gets a tiny dust puff. A hit gets sparks and a short shake. A boss defeat gets everything: long hit stop, huge burst, strong shake, a slow dissolve.

Layer small effects. No single effect makes a hit feel great, but five tiny ones together do. In the Juice Lab, a hit uses a squash, a flash, a burst, a damage number, a shake, and a hit stop. Each is subtle alone.

Keep the player in control. Never let an effect block input or cover the action. Hit stop should be a blink, not a pause. Particles should not hide enemies.

Think about performance. Particles are cheap, but hundreds of big, overlapping, see-through particles can slow a weak computer. Use a modest `Amount`, short lifetimes, and free one-shot nodes when finished (as `burst.gd` does).

Offer options. Screen shake, flashing lights, and strong motion can cause discomfort or even medical problems for some people. Flashing at rates near three or more times a second can trigger seizures in a small number of players. Offer a setting to reduce effects, and keep flashes short.

Do it last. Juice is polish. Get your game working and fun first, then add feel. The effects above take minutes each because the rules are already in place.

Project 20: The Juice Lab

Goal. Build a small lab where you can hit a target with the Space key and turn each juice effect on and off with the keys 1 to 4. You will feel for yourself how much each effect adds.

Time. About 60-90 minutes.

Final project. `projects/ch20-juice/` has everything. Compare your work to it if you get stuck.

```
projects/ch20-juice/
|-- project.godot
|-- icon.svg              # the picture the target shows
|-- main.tscn             # the main scene
|-- main.gd               # input, effects, HUD
|-- target.gd             # hit points, flash, squash, dissolve
|-- target.gdshader       # flash + dissolve shader
|-- background.gdshader   # moving purple background
|-- shake.gd              # camera shake
|-- burst.tscn            # one-shot particle burst
`-- burst.gd
```

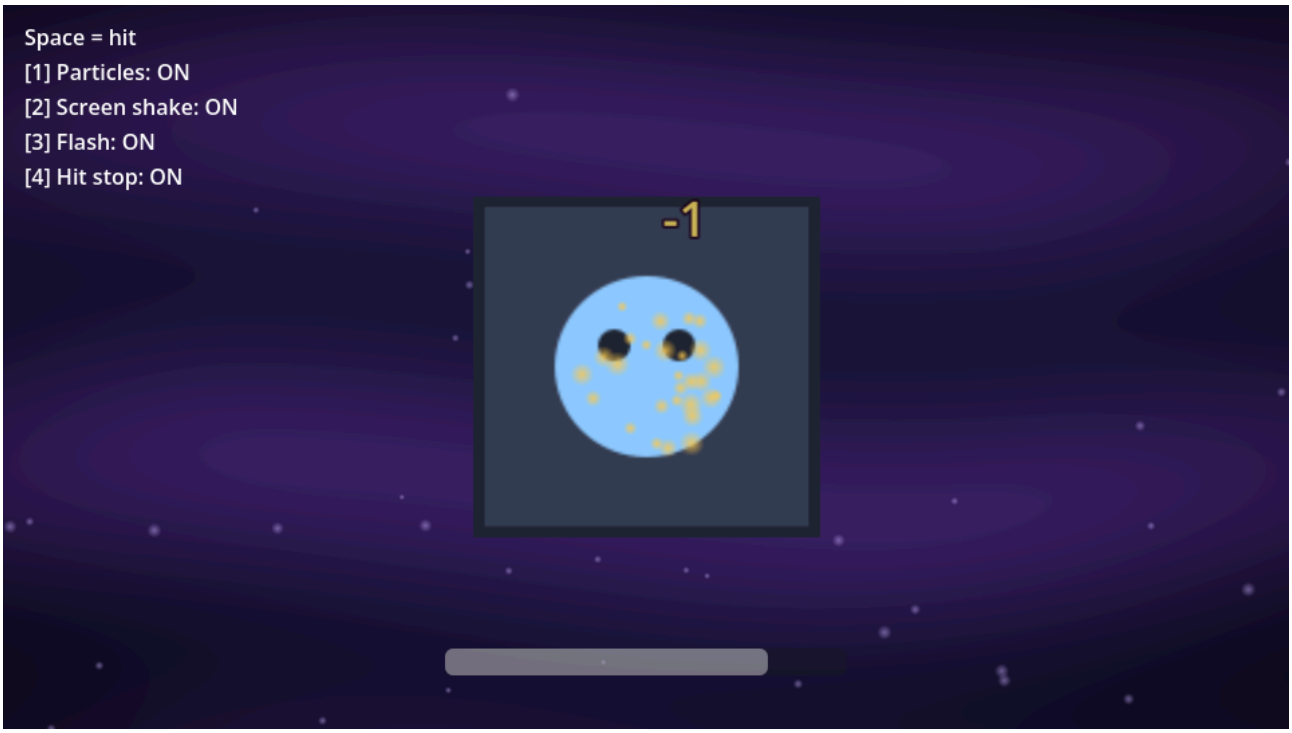


Figure 20.7: The finished Juice Lab in the middle of a hit.

Step 1: Project setup

1. Create a new project `juice-lab` (Compatibility renderer). In **Project Settings > Display > Window**, set the viewport width to 960 and the height to 540.
2. In the Input Map, add an action named `hit` and bind it to **Space**.

Step 2: Write the shaders

1. Create `target.gdshader` and `background.gdshader` in the FileSystem dock (right-click > Create New > Script... works, or use **Create New > Shader** and name them). Paste in the code from section 20.6. Save.
2. If the shader editor reports an error, read the line number it gives. The most common cause is a missing semicolon or an integer where a decimal was needed.

Step 3: Build the burst

Make `burst.tscn` and `burst.gd` exactly as described in section 20.3.

Step 4: Build the main scene

Create a new scene with a `Node2D` root named `Main`. Save it as `main.tscn`. Build this tree:

```
Main (Node2D)                        # script: main.gd
|-- BackgroundLayer (CanvasLayer)    # Layer: -1, so it sits behind everything and does not shake
|   `-- Background (ColorRect)       # Size 960 x 540, Material: ShaderMaterial with background.gdshader
|-- Camera2D                         # Position (480, 270), script: shake.gd
|-- Dust (GPUParticles2D)            # Position (480, 560), continuous dust (see below)
|-- Target (Sprite2D)                # Position (480, 270), Scale (2, 2), Texture: icon.svg,
ShaderMaterial with target.gdshader, script: target.gd
|-- Effects (Node2D)                 # empty; bursts and damage numbers are added here
`-- HUD (CanvasLayer)
    |-- InfoLabel (Label)            # top left, shows the key help
    `-- HpBar (ProgressBar)          # bottom center, Max Value 5, Step 1, Show Percentage off
```

Settings for the `Dust` node: `Amount` 40, `Lifetime` 9, `Preprocess` 9, a soft dot `Texture` (a 16 by 16 radial `GradientTexture2D`), and a process material with `Emission Shape` **Box**, `Box Extents` (480, 5, 1), `Direction` (0, -1, 0), `Spread` 12, `Velocity` 20 to 60, `Gravity` (0, 0, 0), `Scale` 0.3 to 0.8, and a light purple `Color` with alpha 0.5. Set the `Visibility Rect` to position (-500, -600) and size (1000, 700).

For the `Target`, check that its `Material` is a ShaderMaterial whose shader is `target.gdshader`. Attach `target.gd` (section 20.6). For the camera, attach `shake.gd` (section 20.4).

Step 5: The main script

```gdscript
# File: main.gd
# Attach to: Main (Node2D) in main.tscn
extends Node2D

const BURST_SCENE: PackedScene = preload("res://burst.tscn")

@onready var target: Sprite2D = $Target
@onready var camera: Camera2D = $Camera2D
@onready var effects: Node2D = $Effects
@onready var hp_bar: ProgressBar = $HUD/HpBar
@onready var info_label: Label = $HUD/InfoLabel

var use_particles: bool = true
var use_shake: bool = true
var use_flash: bool = true
var use_hitstop: bool = true


func _ready() -> void:
    target.died.connect(_on_target_died)
    hp_bar.max_value = target.max_hp
    _refresh_hud()
```

```
func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("hit"):
        if target.is_alive():
            _hit()
    elif event is InputEventKey and event.pressed and not event.echo:
        match event.keycode:
            KEY_1:
                use_particles = not use_particles
            KEY_2:
                use_shake = not use_shake
            KEY_3:
                use_flash = not use_flash
            KEY_4:
                use_hitstop = not use_hitstop
        camera.enabled_shake = use_shake
        _refresh_hud()


func _hit() -> void:
    var died: bool = target.take_hit()
    hp_bar.value = target.hp

    target.squash()  # squash and stretch is always on, it is cheap and great
    _spawn_damage_number(target.global_position + Vector2(randf_range(-40.0, 40.0), -120.0))

    if use_flash:
        target.flash()
    if use_particles:
        _spawn_burst(target.global_position, Color(1.0, 0.8, 0.3))
    if use_shake:
        camera.add_trauma(0.9 if died else 0.45)
    if use_hitstop:
        _hit_stop(0.12 if died else 0.05)


func _spawn_burst(where: Vector2, color: Color) -> void:
    var burst: GPUParticles2D = BURST_SCENE.instantiate()
    burst.position = where
    burst.modulate = color
    effects.add_child(burst)


func _spawn_damage_number(where: Vector2) -> void:
    var label := Label.new()
    label.text = "-1"
    label.add_theme_font_size_override("font_size", 36)
    label.add_theme_color_override("font_color", Color(1.0, 0.9, 0.4))
    label.add_theme_color_override("font_outline_color", Color(0.1, 0.0, 0.1))
    label.add_theme_constant_override("outline_size", 8)
    label.position = where
    effects.add_child(label)

    var tween: Tween = create_tween().set_parallel(true)
    tween.tween_property(label, "position:y", where.y - 70.0, 0.7)
    tween.tween_property(label, "modulate:a", 0.0, 0.7)
    tween.chain().tween_callback(label.queue_free)


## Freeze the game for a split second. The timer must ignore the time scale,
## otherwise the pause would last 20 times too long.
func _hit_stop(seconds: float) -> void:
    Engine.time_scale = 0.05
    await get_tree().create_timer(seconds, true, false, true).timeout
    Engine.time_scale = 1.0


func _on_target_died() -> void:
    target.dissolve_away(0.8)
    if use_particles:
        _spawn_burst(target.global_position, Color(1.0, 0.5, 0.2))
    await get_tree().create_timer(1.8).timeout
    target.respawn()
```

```
    hp_bar.value = target.max_hp


func _refresh_hud() -> void:
    hp_bar.value = target.hp
    info_label.text = "Space = hit\n[1] Particles: %s\n[2] Screen shake: %s\n[3] Flash: %s\n[4] Hit stop:
%s" % [
        _on_off(use_particles), _on_off(use_shake), _on_off(use_flash), _on_off(use_hitstop)
    ]


func _on_off(value: bool) -> String:
    return "ON" if value else "off"
```

Some notes:

- The effects are kept as separate functions, and each is guarded by a simple `if`. This makes it easy to turn effects on and off to compare.
- `_on_target_died` waits 1.8 seconds with a regular timer and then calls `respawn`. While it waits, the target is dissolved and not alive, so Space does nothing.
- `_unhandled_input` receives the key press only if no UI node used it first. The match on `event.keycode` handles the number keys.

Step 6: Play and compare

Press **F5**. Tap **Space** five times.

1. Notice the squash, the flash, the burst, the "-1", the shake, and the tiny freeze.
2. Press **4** to turn off hit stop. Hit again. The hit feels lighter.
3. Press **2** to turn off shake. Then **3** to turn off the flash. Then **1** for particles. Each time, hit again and feel what is missing.
4. Turn everything back on. On the fifth hit, the target dissolves with a glowing edge, and after about two seconds it pops back with a bouncy scale-in.

Expected result

A moving purple background with gently floating dust. The icon in the middle reacts to every Space press with a squash, a white flash, a spray of golden dots, and a floating "-1". The screen jolts and the game briefly freezes. The health bar goes down. After five hits the icon burns away into holes with an orange edge, then reappears. Keys 1 to 4 switch effects off and on, and the HUD text shows their state.

Chapter summary

- **Juice** is extra feedback that confirms, communicates, and delights, without changing the rules. Scale each effect to the size of the event.
- Tweens with `TRANS_ELASTIC` or `TRANS_BACK` and `EASE_OUT` make squash and stretch, floating numbers, and fades. Kill an old tween before starting a new one on the same property.
- `GPUParticles2D` holds the amount, lifetime, and texture; `ParticleProcessMaterial` holds direction, speed, gravity, size, and color over time. One-shot bursts should `queue_free` on `finished`.
- A trauma-based `Camera2D` script gives screen shake that stacks and fades. The shake strength is the square of the trauma.
- Hit stop slows `Engine.time_scale` briefly. Use a timer that ignores the time scale, and always put the scale back to 1.0.

- A `canvas_item` shader's `fragment()` function runs for every pixel. It reads `TEXTURE` and `UV` and writes `COLOR`. Uniforms are its settings, changed with `set_shader_parameter`.
- Flash, dissolve, and animated backgrounds are all built from `mix`, `step`, `smoothstep`, `sin`, and noise.
- Use effects with restraint, offer a way to reduce them, and add them after the game works.

Review questions

1. In your own words, what is "juice"? Give two examples.
2. True or false: adding screen shake changes the rules of the game.
3. What is the difference between the `GPUParticles2D` node and its `ParticleProcessMaterial`?
4. For a coin pickup sparkle you want all particles to appear at the same instant, only once. Which two properties of `GPUParticles2D` do you set? (a) Preprocess and Amount (b) One Shot and Explosiveness (c) Local Coords and Lifetime (d) Texture and Emitting
5. Why does the shake script use `trauma * trauma` rather than `trauma` itself?
6. A hit stop sets `Engine.time_scale = 0.05` and then waits with `create_timer(0.1)`. The game freezes for far longer than expected. Why, and what is the fix?
7. In a `canvas_item` shader, which built-in variable do you write to produce the final color of a pixel?
8. What does `mix(a, b, 0.25)` return?
9. How do you change a shader's `flash_amount` uniform from GDScript?
10. You flash one enemy and all twenty enemies flash. What is the likely cause, and how do you fix it?

Exercises

1. **Practice (easy):** In `main.gd`, change the hit sparks to a blue color, and make the damage number text say "Ouch!" instead of "-1". Run and check.
2. **Practice (easy):** Open `background.gdshader`, change `color_a` and `color_b` to a dark green and a dark teal, and set `bands` to 8. Describe how the mood changes.
3. **Apply (medium):** Add a fifth toggle on the key 5 for "squash". Hold the value in a variable `use_squash`, show it in the HUD, and guard the call to `target.squash()` with it.
4. **Apply (medium):** Make the burst feel bigger on the final blow. In `_on_target_died`, spawn two extra bursts with different colors. Hint: add a `Color` argument and call `_spawn_burst` several times.
5. **Challenge (hard):** Write a new `outline.gdshader` that draws a colored outline around the sprite. Hint: for each pixel, read the texture alpha at four neighbours (`UV + vec2(pixel, 0.0)` and so on) using `TEXTURE_PIXEL_SIZE`; if the pixel itself is transparent but a neighbour is not, output the outline color.

Common mistakes

- **My particles do nothing.** `Emitting` is off, there is no `Process Material`, or the `Amount` is 0. *Fix:* check all three. For one-shot bursts, make sure the script sets `emitting = true`.
- **Particles are tiny, hard squares.** No `Texture` is set, so each particle is a plain pixel. *Fix:* set a texture, or raise the `Scale` of the process material.
- **The shader does nothing / the node vanishes.** A shader that never writes `COLOR`, or writes zero alpha, hides the node. *Fix:* start from `COLOR = texture(TEXTURE, UV);` and add effects a step at a time. Read the red error text in the shader editor.
- **"Invalid arguments for the built-in function" in a shader.** Mixed integers and decimals. *Fix:* write `1.0` instead of `1`.
- **The game stays in slow motion.** `Engine.time_scale` was not set back to 1.0. *Fix:* make sure each hit stop ends by restoring it, and avoid ending the function early.

- **The screen shake never stops, or the picture is off-center after.** The camera `offset` was not reset to zero when the trauma ended. *Fix:* include the `else` branch that resets `offset` and `rotation`.

Further reading

- Particle systems (2D): https://docs.godotengine.org/en/stable/tutorials/2d/particle_systems_2d.html
- GPUParticles2D class reference: https://docs.godotengine.org/en/stable/classes/class_gpuparticles2d.html
- ParticleProcessMaterial class reference: https://docs.godotengine.org/en/stable/classes/class_particleprocessmaterial.html
- Introduction to shaders: https://docs.godotengine.org/en/stable/tutorials/shaders/introduction_to_shaders.html
- Your first 2D shader: https://docs.godotengine.org/en/stable/tutorials/shaders/your_first_shader/your_first_2d_shader.html
- Shading language reference: https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/shading_language.html
- CanvasItem shaders: https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/canvas_item_shader.html
- Tween class reference: https://docs.godotengine.org/en/stable/classes/class_tween.html

Chapter 21: Game AI and Navigation

Learning objectives

By the end of this chapter you will be able to:

- Explain what "game AI" means and describe the sense, think, act loop.
- Use `Timer` nodes and `create_timer` for cooldowns, delays, and repeating actions.
- Build a finite state machine in GDScript with an `enum` and `match`.
- Set up a `NavigationRegion2D` and `NavigationAgent2D` so a character finds its way around walls.
- Give an enemy senses: a sight range and a line-of-sight check with a physics ray.
- Combine all of this into an enemy that idles, patrols, chases, searches, and recovers.

Key terms

- **Game AI**: the code that makes computer-controlled characters behave in believable, interesting ways. It is not the same as the "artificial intelligence" in the news; most game AI is simple rules.
- **Agent**: any character controlled by AI. In this chapter, the slime.
- **State**: one mode of behavior, such as *idle* or *chase*. An agent is in exactly one state at a time.
- **State machine**: a design where the agent has a list of states and clear rules for moving from one state to another.
- **Transition**: a move from one state to another, caused by an event or condition ("I see the player").
- **Timer**: a node that counts down and sends a signal when it reaches zero.
- **Cooldown**: a short delay before an action can be done again.
- **Navigation mesh** (navmesh): the shape of the area where a character is allowed to walk, made of connected polygons.
- **Pathfinding**: working out a route from here to there that does not go through walls.
- **Waypoint**: a marked position an agent walks to, such as a patrol point.
- **Line of sight**: a clear straight line between two points with nothing solid in the way.
- **Raycast**: asking the physics engine "if I shoot a line from A to B, what is the first thing it hits?"

21.1 What is game AI?

When you hear "AI," you may think of chatbots or robots that learn. Game AI is much humbler and much more practical. It means: *code that decides what a non-player character does next*.

A slime that wanders around, notices you, and chases you does not need to be smart. It needs to be **believable**, **readable**, and **fair**. Players should be able to understand what the enemy is doing ("it saw me!") and have a chance to react. A perfect enemy that never misses is not fun; an enemy with understandable habits is.

Nearly all game AI, from a slime in a tutorial to a ghost in a classic maze game, follows one loop:

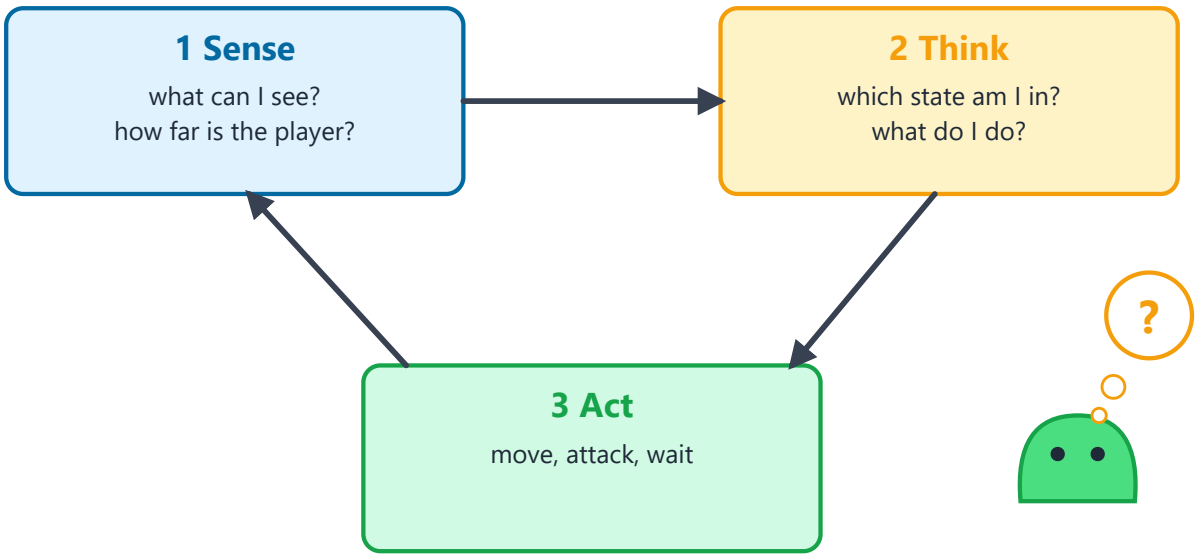


Figure 21.1: The AI loop. The agent senses the world, thinks about what to do, acts, and repeats, many times per second.

1. **Sense.** Gather facts: how far is the player? Can I see them? Did a timer run out?
2. **Think.** Use those facts to choose what to do. Our tool for this is the *state machine* (section 21.3).
3. **Act.** Do it: walk toward a point, attack, or stand still.

This chapter builds each part and then joins them in one project, a slime arena where you collect coins while a slime patrols and hunts you.

21.2 Timers: doing things later

AI is full of waiting. A slime pauses for two seconds, an attack can only be used every second, a monster forgets you after five seconds. All of these need a **timer**.

The Timer node

Godot has a `Timer` node. Add one as a child of a node, and set its properties in the Inspector:

- **`Wait Time`**: how many seconds it counts down.
- **`One Shot`**: when on, the timer stops after it finishes once. When off, it restarts automatically and fires again and again.
- **`Autostart`**: when on, the timer starts as soon as the scene starts.

When the timer reaches zero, it emits the **`timeout`** signal. You connect that signal to a function, just as you connected button presses in Chapter 5.

```gdscript
# File: heartbeat.gd
# Attach to: a Node2D with a child Timer named Timer (Wait Time 1.0, One Shot off, Autostart on)
extends Node2D

@onready var timer: Timer = $Timer


func _ready() -> void:
    timer.timeout.connect(_on_timer_timeout)
```

```
func _on_timer_timeout() -> void:
    print("tick")
```

The output shows "tick" once a second. A timer is *much* easier than counting seconds yourself with `delta` in `_process`, because the engine handles the counting and you just wait for the signal.

Starting and stopping a timer from code

Use these functions:

```
# Fragment: controlling a Timer node named timer.
timer.start()          # start counting with the Wait Time from the Inspector
timer.start(2.5)       # start counting with a different wait time this once
timer.stop()           # cancel it; no timeout signal will arrive
print(timer.time_left) # seconds remaining
print(timer.is_stopped())
```

Calling `start()` on a timer that is already running restarts the count. This is just what you want for a repath timer or a "cooldown resets when hit again" effect.

Quick timers without a node

Sometimes you only need a single wait inside a function. Use `get_tree().create_timer(seconds)` together with `await`:

```
# Fragment: wait half a second inside a function.
func flash_then_hide() -> void:
    show()
    await get_tree().create_timer(0.5).timeout
    hide()
```

The `await` keyword (you saw it in Chapter 20) pauses *this function* until the signal fires, while the rest of the game carries on. It is perfect for short, one-off delays.

> **Note:** Which one should you use? Use a **`Timer` node** when the timer belongs to a thing that can be started, stopped, or restarted (a cooldown, a state's duration). Use **`create_timer` with `await`** for a quick "wait, then do" in a single function. If the node that is waiting is removed from the scene during the wait, the `await` simply never continues, which is usually what you want.

Cooldowns

A **cooldown** is a timer used to stop an action being repeated too fast. A simple pattern:

```
# Fragment: an attack that can only be used once a second.
var can_attack: bool = true

func try_attack() -> void:
    if not can_attack:
        return
    can_attack = false
    print("attack!")
    await get_tree().create_timer(1.0).timeout
    can_attack = true
```

Cooldowns keep AI fair. Our slime uses a similar idea: after it attacks, it enters a "recover" state for 1.2 seconds, when it cannot do anything, giving the player time to get away.

21.3 State machines: giving the AI a mind

The problem with a pile of ifs

Imagine writing the slime's behavior as one long list of `if` checks:

```
if I can see the player and I am not recovering and I am close and ... attack
elif I can see the player and not recovering ... chase
elif I just lost the player and ... search
...
```

Every new behavior makes the list longer, and the conditions get tangled. Soon, adding "flee when hurt" breaks "chase" in a way you cannot find.

The solution: states

A **state machine** organizes behavior into separate **states**. At any moment, the slime is in exactly one of them, and each state has a small job and a few clear exits. Figure 21.2 shows the slime's brain.

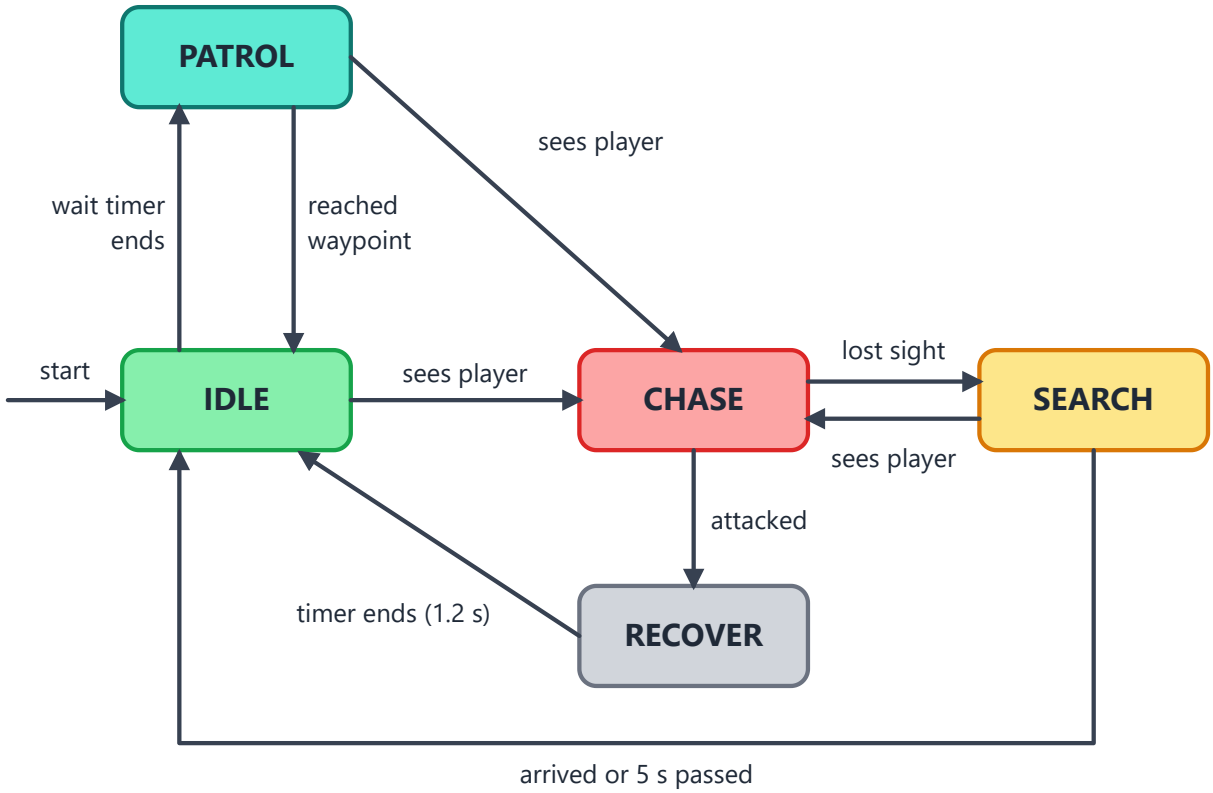


Figure 21.2: The slime's state machine. Each box is something the slime can be doing. Each arrow is a transition with the reason it happens.

Reading the diagram, the slime's story is:

- It starts **IDLE**: standing still for a second or two.
- When the wait ends, it goes **PATROL**: walking to the next waypoint.
- If it sees you while idle, patrolling, or searching, it goes **CHASE**.
- While chasing, if it gets close enough, it attacks and goes to **RECOVER**, a short pause.
- If you break the line of sight while it chases, it goes to **SEARCH** and heads to the last place it saw you. If it finds you again, back to CHASE. If it arrives and finds nothing (or 5 seconds pass), it gives up and goes IDLE.

Notice that a state machine is easy to *draw*. If you can sketch the boxes and arrows on paper first, the code is mostly typing. Do that every time.

Implementing states with enum and match

An `enum` is a named list of options. Use one to list the states, so you never mistype a state as a number or a word:

```
# Fragment: declaring states.
enum State { IDLE, PATROL, CHASE, SEARCH, RECOVER }

var state: State = State.IDLE
```

Behind the scenes, `State.IDLE` is 0, `State.PATROL` is 1, and so on, but you use the names. The `match` statement chooses what to run based on the current value, like a tidy `if / elif` chain:

```
# Fragment: running the code for the current state each physics frame.
func _physics_process(delta: float) -> void:
    match state:
        State.IDLE:
            _think_idle()
        State.PATROL:
            _think_patrol()
        State.CHASE:
            _think_chase()
        State.SEARCH:
            _think_search()
        State.RECOVER:
            pass  # do nothing; the timer will end this state
```

Each state's code stays in its own place. Adding a new state means adding one name to the enum, one branch to `match`, and the new transitions.

A single door between states

All changes of state should go through **one function**. That gives you one place to do "setup" work whenever a state begins, and "cleanup" when it ends:

```
# Fragment: a single function changes state.
func _change_state(new_state: State) -> void:
    state = new_state
    state_timer.stop()      # leaving any state cancels its timers
    repath_timer.stop()
    velocity = Vector2.ZERO

    match state:
        State.IDLE:
            state_timer.start(randf_range(1.0, 2.5))   # wait a random time
        State.RECOVER:
            state_timer.start(1.2)
        # ... other states do their own setup here
```

Why one door? Because otherwise, you will forget to stop a timer in one place, and the slime will do two things at once. By always stopping the timers and clearing the velocity first, every state starts from a clean slate.

> **Why it works:** This idea is called *entering* a state. When you write `_change_state(State.IDLE)`, the slime stops, and a new random wait begins. It works the same however the slime ends up in IDLE: after a patrol, after an attack, or after a search. One door means one behavior.

Showing the state while you work

When you debug AI, the most useful tool is to *see* what state the agent is in. The slime has a small `Label` above its head that shows the current state's name, and it changes color in each state. `State.keys()[state]` turns the enum value back into a text name like "CHASE". We use it in the project. Remove or hide the label in a real game if you do not want players to see it.

21.4 Navigation: finding the way

The problem

Suppose the player stands on one side of a wall and the slime on the other. If the slime just walks in a straight line toward the player, it will press itself against the wall and stay stuck. It needs to know a route around the wall. That is **pathfinding**, and Godot does it for you with the navigation system.

How it works

Three parts work together:

1. A **navigation mesh** describes where walking is allowed. It is a set of connected polygons that cover the floor, with holes where walls and obstacles are.
2. The **navigation server** takes the mesh, and when asked "how do I get from A to B?", it works out a list of points to follow.
3. A `NavigationAgent2D` is a helper node on your character. You give it a target and it tells you which point to walk toward next.

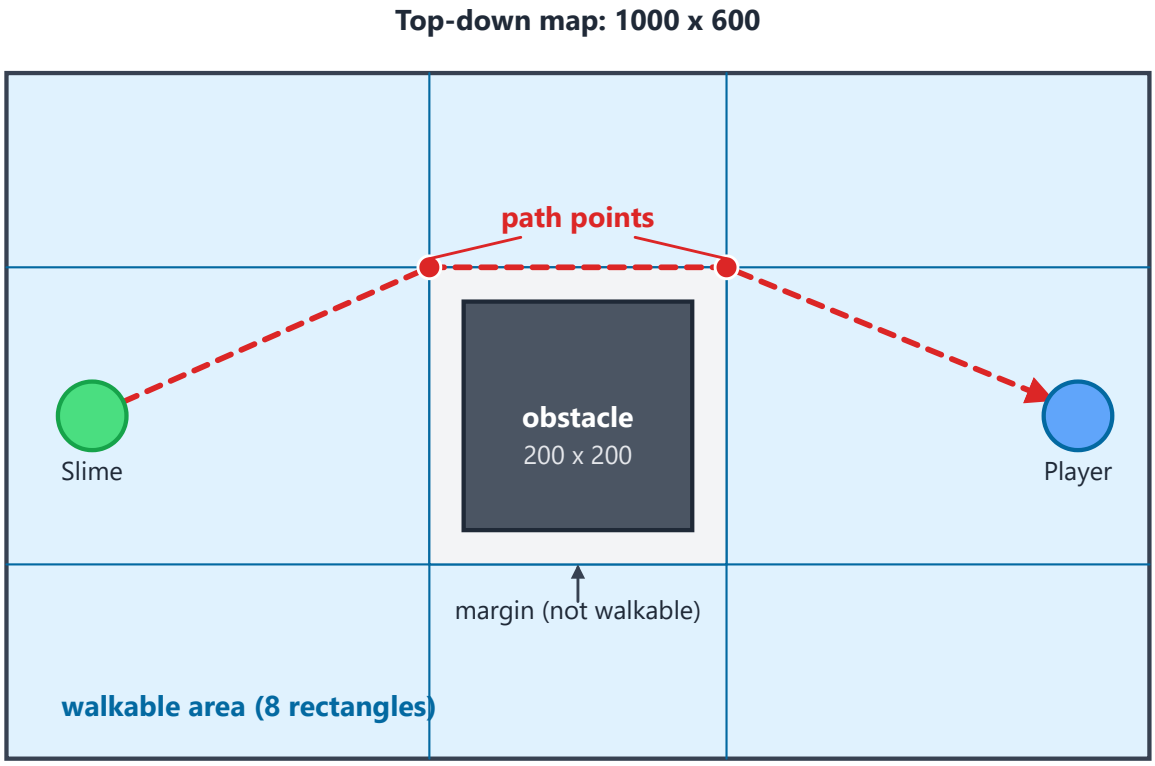


Figure 21.3: The navigation mesh (blue) covers only the floor you can walk on. The path (red) goes around the corners of the hole, because the shortest legal route bends there.

Making the navigation mesh

In 2D, the mesh is a `NavigationPolygon` resource held by a **`NavigationRegion2D`** node. You can build it in the editor:

1. Add a `NavigationRegion2D` to your scene.
2. In the Inspector, choose **New NavigationPolygon** for its `Navigation Polygon` property.
3. With the node selected, the viewport toolbar shows editing buttons. Click the pen button and click points to draw an **outline** around the whole walkable area. Close it by clicking the first point again.
4. Draw a second outline *inside* the first, around each obstacle. An outline inside another outline becomes a **hole**.
5. Press **Bake NavigationPolygon**. Godot fills the outlines with polygons.

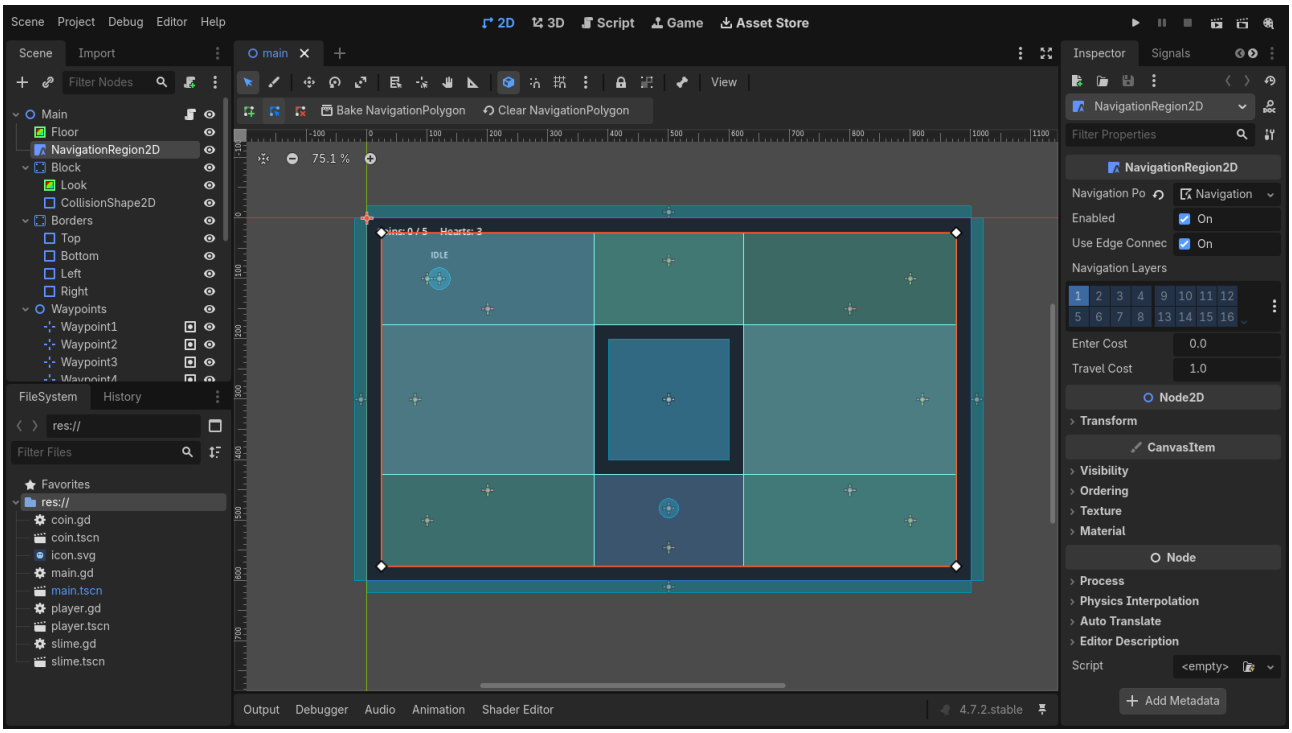


Figure 21.4: The baked navigation mesh in the editor. The blue area is walkable, and the hole in the middle matches the block, with a little margin.

> **Important: leave a margin.** The navigation system plans a path for a single *point*, the center of your character, but your slime has a body with a radius of 18 pixels. If the mesh hugs the wall exactly, the path touches the wall's corner and the slime's body gets stuck on it. So draw the hole bigger than the wall (by at least the character's radius) and the outer outline smaller than the room. In our arena the block is 200 by 200 pixels and the hole is 248 by 248, a 24 pixel margin on each side. Godot can also add this margin for you with the `agent_radius` property of the navigation polygon when baking.

The project's `main.tscn` stores the mesh as eight rectangles that cover the floor around the hole, written by hand; the result is the same as baking. Baking from the editor with two outlines gives you equivalent coverage with triangles.

The NavigationAgent2D

Add a `NavigationAgent2D` as a child of your `CharacterBody2D`. It has a few properties worth knowing:

- `Path Desired Distance`: how close the character must get to a point on the path before it counts as reached, and the agent moves on to the next one. We use 6 pixels.
- `Target Desired Distance`: how close to the final target counts as "arrived". We use 10 pixels.
- `Avoidance Enabled`: lets several agents steer around each other. We leave it off here. It is a good topic for further study.
- `Debug Enabled`: draws the path in the running game. Turn it on while learning; you can see exactly where the agent plans to go.

Using the agent takes three steps:

1. **Set the target.** `agent.target_position = some_position`. The agent works out the path.
2. **Ask for the next point.** `agent.get_next_path_position()` returns the next corner on the route.
3. **Walk toward it.** Aim the velocity at that point and call `move_and_slide()`.

```gdscript
# Fragment from slime.gd: follow the path.
func _move_along_path(speed: float) -> void:
    var next_position: Vector2 = agent.get_next_path_position()
    velocity = global_position.direction_to(next_position) * speed
    move_and_slide()
```

`direction_to` gives a vector of length 1 that points from here to there. Multiplying by `speed` gives velocity in pixels per second.

To know when the journey is over, ask `agent.is_navigation_finished()`. It becomes true when the agent is within `Target Desired Distance` of the target (or when there is no path at all).

> **Warning:** The navigation map is not ready on the very first frame. If you set `target_position` in `_ready`, the agent finds no path and the character does not move. The standard fix is to wait one physics frame first:
>
> ```gdscript
> func _ready() -> void:
>     await get_tree().physics_frame
>     agent.target_position = some_position
> ```
>
> The slime does exactly this, then switches on its brain with a flag.

Chasing a moving target

A chase target moves, but a path is a fixed list of points. If you set the target only once, the slime would run to where the player *was*. The fix is to set the target again every so often. Doing it every frame is wasteful (pathfinding costs a little each time), so use a repeating `Timer`, here called `RepathTimer`, with a wait time of 0.25 seconds:

```gdscript
# Fragment from slime.gd.
func _on_repath_timer_timeout() -> void:
    if state == State.CHASE:
        agent.target_position = _player.global_position
```

A quarter second is quick enough to feel smart and slow enough to be cheap.

21.5 Senses: seeing the player

A believable enemy does not know where you are all the time. It must *notice* you. We give the slime sight with two simple checks.

Check 1: distance

Is the player closer than the sight range? `global_position.distance_to(other.global_position)` gives the distance in pixels. If it is more than `sight_range`, the slime does not see you, no matter what.

Check 2: line of sight

Close is not enough. A player right behind a wall is close, but the slime should not see through the wall. To check, we cast a **ray**: an invisible line from the slime to the player. If the line hits a wall first, the view is blocked.

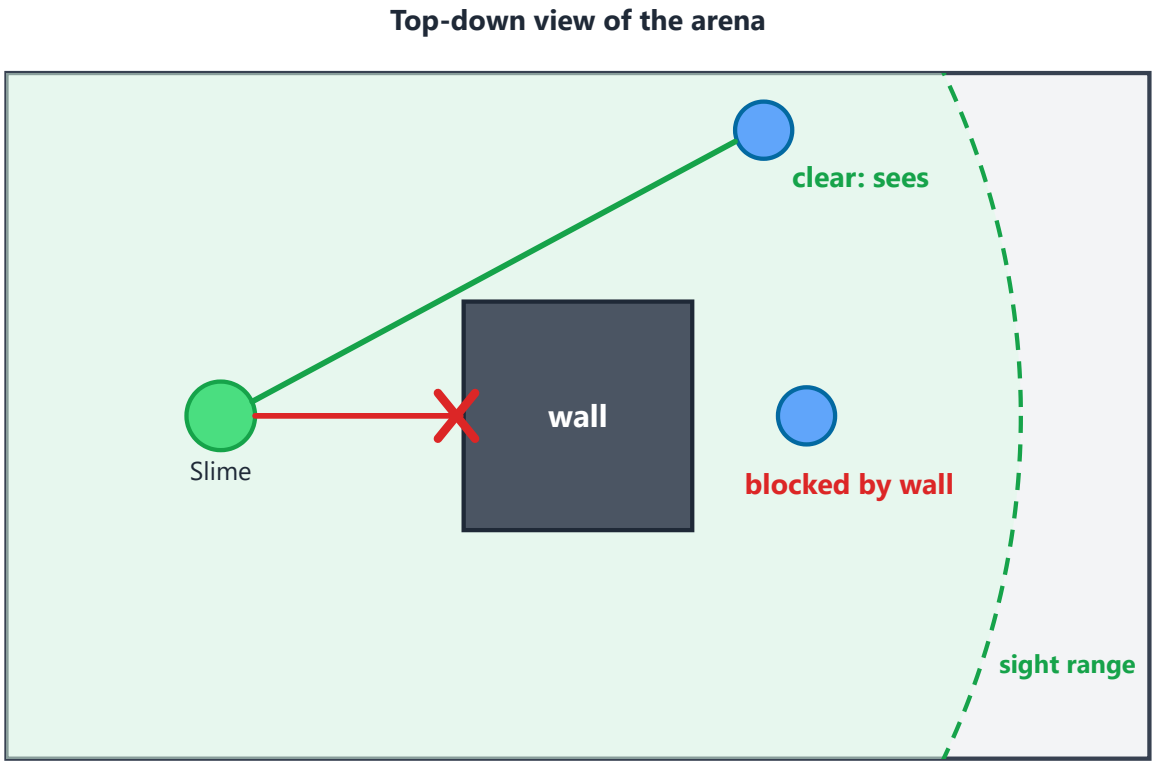


Figure 21.5: Sight needs both range and a clear line. The dashed circle is the sight range; the red ray is blocked by the wall.

Raycasts are done through the physics world:

```gdscript
# Fragment from slime.gd.
func _can_see_player() -> bool:
    if global_position.distance_to(_player.global_position) > sight_range:
        return false
    # Ask the physics world: is any wall (layer 1) between us and Pip?
    var query: PhysicsRayQueryParameters2D = PhysicsRayQueryParameters2D.create(
        global_position, _player.global_position, 1
    )
    var hit: Dictionary = get_world_2d().direct_space_state.intersect_ray(query)
    return hit.is_empty()
```

How it works:

- `PhysicsRayQueryParameters2D.create(from, to, mask)` describes the ray. The third argument is a **collision mask** number: it says which physics layers the ray can hit. Layer 1 is the first layer, so the value is 1. In the project, walls are on layer 1, the player is on layer 2, and the slime is on layer 4 (the third layer, whose value is 4). The ray only looks at walls, so Pip himself does not block it.
- `get_world_2d().direct_space_state` is the physics world, and `intersect_ray` shoots the ray. It returns a dictionary. If the ray hit something, the dictionary has information (such as `position` and `collider`). If it hit nothing, the dictionary is **empty**.
- So `hit.is_empty()` means "nothing blocks the view," which means the slime can see the player.

> **Note:** Collision layers use numbers that double: layer 1 is the value 1, layer 2 is 2, layer 3 is 4, layer 4 is 8, and so on. In the Inspector you tick numbered boxes, but in code a mask is a sum of these values. You learned the layer boxes in Chapter 8; this is the code version.

Groups as a phone book

How does the slime find the player? We put the player scene's root node in the group `player`, and put the four patrol markers in a group `waypoint`. Then, in `_ready`:

```
# Fragment from slime.gd.
_player = get_tree().get_first_node_in_group("player")
_waypoints = get_tree().get_nodes_in_group("waypoint")
```

This means the slime scene does not need a fixed path to the player; it works in any level that has them. (Groups were introduced in Chapter 5.)

21.6 The complete slime brain

Everything is now ready. Here is the full `slime.gd`. Read it from top to bottom; each function matches something from this chapter.

```
# File: slime.gd
# Attach to: Slime (CharacterBody2D) at the root of slime.tscn
# Needs these children: NavigationAgent2D, StateTimer (Timer), RepathTimer (Timer), StateLabel (Label).
extends CharacterBody2D

signal attacked_player

enum State { IDLE, PATROL, CHASE, SEARCH, RECOVER }

const STATE_COLORS: Dictionary = {
    State.IDLE: Color(0.5, 0.8, 0.5),
    State.PATROL: Color(0.4, 0.9, 0.7),
    State.CHASE: Color(1.0, 0.35, 0.3),
    State.SEARCH: Color(1.0, 0.8, 0.3),
    State.RECOVER: Color(0.6, 0.6, 0.7),
}

@export var patrol_speed: float = 70.0
@export var chase_speed: float = 135.0
@export var sight_range: float = 280.0
@export var attack_range: float = 36.0
@export var radius: float = 18.0

var state: State = State.IDLE

var _player: Node2D
var _waypoints: Array[Node] = []
var _waypoint_index: int = -1
var _last_seen_position: Vector2
var _ready_to_think: bool = false

@onready var agent: NavigationAgent2D = $NavigationAgent2D
@onready var state_timer: Timer = $StateTimer
@onready var repath_timer: Timer = $RepathTimer
@onready var state_label: Label = $StateLabel


func _ready() -> void:
    _player = get_tree().get_first_node_in_group("player")
    _waypoints = get_tree().get_nodes_in_group("waypoint")
    state_timer.timeout.connect(_on_state_timer_timeout)
    repath_timer.timeout.connect(_on_repath_timer_timeout)

    # The navigation map is not ready until the first physics frame has passed.
    await get_tree().physics_frame
    _ready_to_think = true
    _change_state(State.IDLE)
```

```
func _physics_process(_delta: float) -> void:
    if not _ready_to_think or _player == null:
        return

    match state:
        State.IDLE:
            if _can_see_player():
                _change_state(State.CHASE)
        State.PATROL:
            if _can_see_player():
                _change_state(State.CHASE)
            elif agent.is_navigation_finished():
                _change_state(State.IDLE)
            else:
                _move_along_path(patrol_speed)
        State.CHASE:
            _chase()
        State.SEARCH:
            if _can_see_player():
                _change_state(State.CHASE)
            elif agent.is_navigation_finished():
                _change_state(State.IDLE)
            else:
                _move_along_path(patrol_speed * 1.3)
        State.RECOVER:
            pass  # stand still and catch breath until the timer ends

    queue_redraw()


func _chase() -> void:
    if _can_see_player():
        _last_seen_position = _player.global_position
        if global_position.distance_to(_player.global_position) <= attack_range:
            attacked_player.emit()
            _change_state(State.RECOVER)
            return
        _move_along_path(chase_speed)
    else:
        _change_state(State.SEARCH)


func _change_state(new_state: State) -> void:
    state = new_state
    state_label.text = State.keys()[state]
    state_timer.stop()
    repath_timer.stop()
    velocity = Vector2.ZERO

    match state:
        State.IDLE:
            state_timer.start(randf_range(1.0, 2.5))
        State.PATROL:
            if _waypoints.is_empty():
                _change_state(State.IDLE)
                return
            _waypoint_index = (_waypoint_index + 1) % _waypoints.size()
            agent.target_position = _waypoints[_waypoint_index].global_position
        State.CHASE:
            _last_seen_position = _player.global_position
            agent.target_position = _last_seen_position
            repath_timer.start()
        State.SEARCH:
            agent.target_position = _last_seen_position
            state_timer.start(5.0)
        State.RECOVER:
            state_timer.start(1.2)


func _move_along_path(speed: float) -> void:
```

```
    var next_position: Vector2 = agent.get_next_path_position()
    velocity = global_position.direction_to(next_position) * speed
    move_and_slide()


func _can_see_player() -> bool:
    if global_position.distance_to(_player.global_position) > sight_range:
        return false
    # Ask the physics world: is any wall (layer 1) between us and Pip?
    var query: PhysicsRayQueryParameters2D = PhysicsRayQueryParameters2D.create(
        global_position, _player.global_position, 1
    )
    var hit: Dictionary = get_world_2d().direct_space_state.intersect_ray(query)
    return hit.is_empty()


func _on_state_timer_timeout() -> void:
    match state:
        State.IDLE:
            _change_state(State.PATROL)
        State.SEARCH:
            _change_state(State.IDLE)
        State.RECOVER:
            _change_state(State.IDLE)


func _on_repath_timer_timeout() -> void:
    if state == State.CHASE:
        agent.target_position = _player.global_position


func _draw() -> void:
    draw_circle(Vector2.ZERO, radius, STATE_COLORS[state])
    # Eyes look the way the slime is heading.
    var look: Vector2 = velocity.normalized() * 4.0
    for side: float in [-6.0, 6.0]:
        draw_circle(Vector2(side, -4.0) + look, 4.5, Color.WHITE)
        draw_circle(Vector2(side, -4.0) + look * 1.5, 2.0, Color(0.1, 0.1, 0.2))
```

Let's trace a few important details.

The `_ready_to_think` flag. Because `_ready` waits for a physics frame with `await`, the first few calls of `_physics_process` might arrive before the brain is ready. The flag makes `_physics_process` return early until the slime is set up. Without it, the match would run with a state of IDLE and an unready navigation map.

One door, many exits. Every transition is a call to `_change_state`. Search the code: it is called from `_ready`, from the three states in `_physics_process`, from `_chase`, from the state timer, and once from inside itself (the empty-waypoint check). Because everything goes through the same function, each state begins cleanly.

`_waypoint_index` starts at -1. The first time the slime patrols, `(-1 + 1) % 4` is 0, so it walks to the first waypoint. The `%` (modulo) operator gives the remainder after division, so the index wraps back to 0 after the last waypoint. This is how you loop through a list forever.

Different speeds tell a story. The slime patrols slowly (70), chases faster (135), but Pip moves at 220. That means a chase is a threat but not a death sentence: the player can always outrun the slime in open space. A slime that is faster than the player feels unfair.

Recover is mercy. After an attack, the slime stands still for 1.2 seconds. Pip is also invulnerable for about a second after being hurt. Together they give the player a moment to escape. This is the kind of kindness that makes an enemy fun rather than frustrating.

The drawn eyes. `_draw()` makes the slime without any image files. Each frame, `queue_redraw()` asks Godot to call `_draw()` again, so the color follows the state and the eyes follow the velocity. `for side: float in [-6.0, 6.0]` loops twice, once for each eye.

> **Try it:** In the Inspector, change `sight_range` to 600 on the Slime node (it is `@export`). The slime now spots you from across the arena. Then set it to 120. Which feels more fun to play against? Different enemies can share this script with different numbers.

Project 21: Slime Arena

Goal. Build a top-down arena where Pip collects five coins while a slime patrols, chases, searches, and attacks. Pip has three hearts. Collect five coins to win.

Time. About 75-90 minutes.

Final project. The full project is in `projects/ch21-slime-ai/`.

```
projects/ch21-slime-ai/
|-- project.godot
|-- icon.svg
|-- main.tscn      # the arena: floor, block, borders, navigation, waypoints, coin spots
|-- main.gd        # coins, hearts, win and lose
|-- player.tscn    # Pip (CharacterBody2D)
|-- player.gd
|-- slime.tscn     # the slime with agent and timers
|-- slime.gd       # the AI brain (listed in 21.6)
|-- coin.tscn      # a coin (Area2D)
`-- coin.gd
```

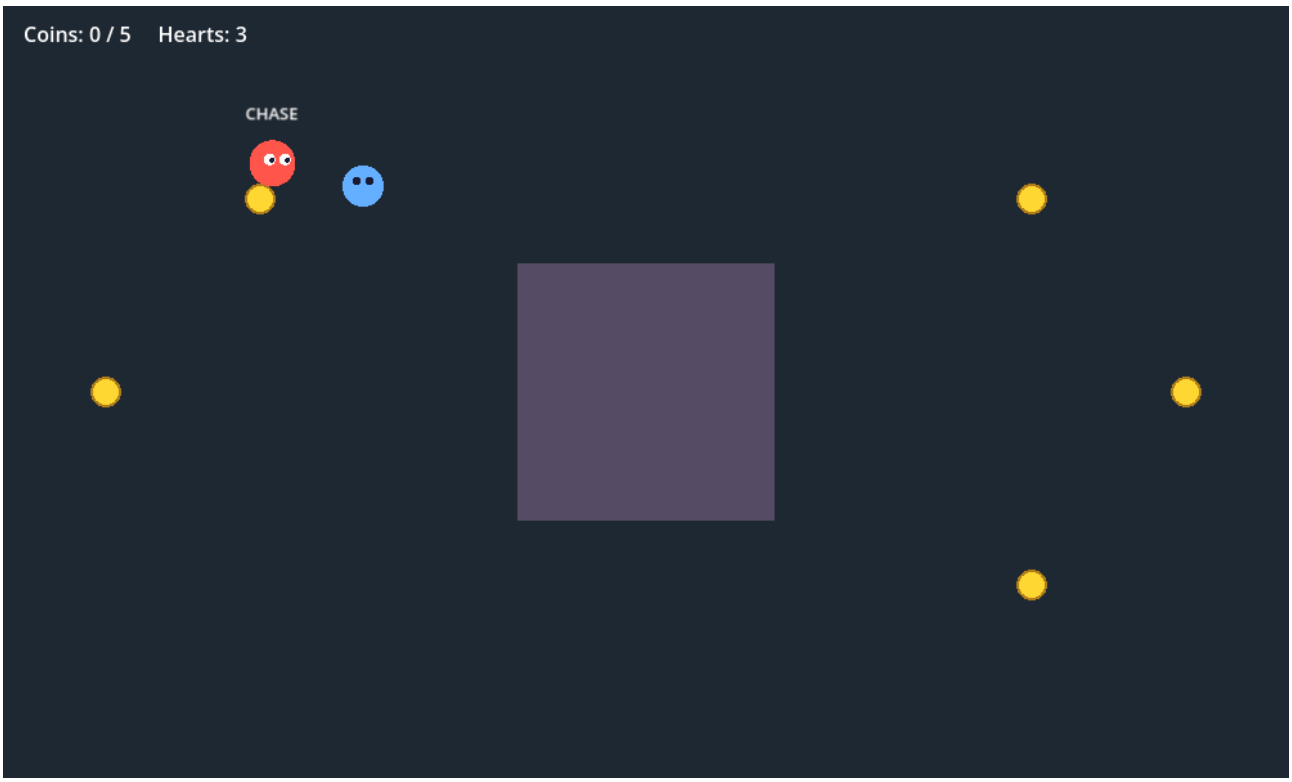


Figure 21.6: The finished game. The label above the slime shows its current state, and its color changes with the state.

Step 1: Project setup

1. Create a new project `slime-arena` (Compatibility renderer). In **Project Settings > Display > Window**, set the viewport to 1000 by 600.
2. In the Input Map, add `move_left` (A), `move_right` (D), `move_up` (W), and `move_down` (S).
3. In **Project Settings > Layer Names > 2D Physics**, you can optionally name layer 1 `walls`, layer 2 `player`, and layer 3 `enemies`.

Step 2: The player

Create a scene with a `CharacterBody2D` root named `Player`, saved as `player.tscn`. Add a `CollisionShape2D` child with a `CircleShape2D` of radius 16. In the Inspector:

- **Groups** (the Groups tab next to the Inspector): add the group `player`.
- **Collision > Layer**: tick only layer **2**. **Mask**: tick only layer **1**. (Pip is on layer 2 and collides with walls.)
- **Motion Mode**: Floating. This tells `move_and_slide` there is no gravity or floor, which is right for a top-down game.

```gdscript
# File: player.gd
# Attach to: Player (CharacterBody2D) in player.tscn
extends CharacterBody2D

@export var speed: float = 220.0
@export var radius: float = 16.0

var _invulnerable: bool = false


func _physics_process(_delta: float) -> void:
    var direction: Vector2 = Input.get_vector("move_left", "move_right", "move_up", "move_down")
    velocity = direction * speed
    move_and_slide()


## Called when an enemy touches Pip. Returns true if Pip really lost a heart.
func hurt() -> bool:
    if _invulnerable:
        return false
    _invulnerable = true
    var tween: Tween = create_tween()
    tween.set_loops(5)
    tween.tween_property(self, "modulate:a", 0.25, 0.1)
    tween.tween_property(self, "modulate:a", 1.0, 0.1)
    tween.finished.connect(func() -> void: _invulnerable = false)
    return true


func _draw() -> void:
    draw_circle(Vector2.ZERO, radius, Color(0.4, 0.7, 1.0))
    draw_circle(Vector2(-5, -4), 3.0, Color(0.1, 0.1, 0.2))
    draw_circle(Vector2(5, -4), 3.0, Color(0.1, 0.1, 0.2))
```

The `hurt()` function does two jobs. It tells the caller whether damage really happened (so the caller only removes a heart when it did), and it makes Pip blink for one second (5 loops of 0.2 seconds) during which he cannot be hurt again. The small function after `finished.connect(` is an unnamed function (a *lambda*) that clears the flag when the blinking ends.

Step 3: The coin

Create `coin.tscn`: an `Area2D` named `Coin` with a `CollisionShape2D` (a `CircleShape2D`, radius 12). Set its **Collision > Layer** to nothing (untick everything) and its **Mask** to layer **2** (it only notices Pip).

```
# File: coin.gd
# Attach to: Coin (Area2D) in coin.tscn
extends Area2D

signal collected


func _ready() -> void:
    body_entered.connect(_on_body_entered)


func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("player"):
        collected.emit()
        queue_free()


func _draw() -> void:
    draw_circle(Vector2.ZERO, 11.0, Color(1.0, 0.85, 0.2))
    draw_arc(Vector2.ZERO, 11.0, 0.0, TAU, 24, Color(0.7, 0.5, 0.1), 2.0)
```

Step 4: The slime scene

Create `slime.tscn`:

```
Slime (CharacterBody2D)        # script: slime.gd; Layer: 3 only; Mask: 1 only; Motion Mode: Floating
|-- CollisionShape2D           # CircleShape2D, radius 18
|-- NavigationAgent2D          # Path Desired Distance 6, Target Desired Distance 10
|-- StateTimer (Timer)         # One Shot: on
|-- RepathTimer (Timer)        # Wait Time 0.25, One Shot: off, Autostart: off
`-- StateLabel (Label)         # text "IDLE", centered, small font, placed above the slime
```

Position `StateLabel` so that it floats above the circle, for example Offset Left -40, Top -48, Right 40, Bottom -25, with `Horizontal Alignment` set to Center. Attach the full `slime.gd` from section 21.6.

Step 5: The arena

Create `main.tscn` with a `Node2D` root named `Main`:

```
Main (Node2D)                      # script: main.gd
|-- Floor (ColorRect)              # size 1000 x 600, dark blue-gray
|-- NavigationRegion2D             # the walkable area (see below)
|-- Block (StaticBody2D)           # position (500, 300)
|   |-- Look (ColorRect)           # offsets -100,-100 to 100,100, purple
|   `-- CollisionShape2D           # RectangleShape2D 200 x 200
|-- Borders (StaticBody2D)
|   |-- Top (CollisionShape2D)     # rectangle 1000 x 20 at (500, -10)
|   |-- Bottom (CollisionShape2D)  # rectangle 1000 x 20 at (500, 610)
|   |-- Left (CollisionShape2D)    # rectangle 20 x 600 at (-10, 300)
|   `-- Right (CollisionShape2D)   # rectangle 20 x 600 at (1010, 300)
|-- Waypoints (Node2D)
|   |-- Waypoint1 (Marker2D)       # (100, 100), group "waypoint"
|   |-- Waypoint2 (Marker2D)       # (900, 100), group "waypoint"
|   |-- Waypoint3 (Marker2D)       # (900, 500), group "waypoint"
|   `-- Waypoint4 (Marker2D)       # (100, 500), group "waypoint"
|-- CoinSpots (Node2D)             # eight Marker2D children named Spot1 ... Spot8
|-- Coins (Node2D)                 # empty; main.gd fills it
|-- Player                         # instance of player.tscn, position (500, 480)
```

```
|-- Slime                          # instance of slime.tscn, position (120, 100)
`-- HUD (CanvasLayer)
    |-- InfoLabel (Label)          # top left
    `-- MessageLabel (Label)       # below it, empty
```

The eight coin spots can be at: (200, 150), (800, 150), (200, 450), (800, 450), (80, 300), (920, 300), (500, 70), (500, 545).

The navigation region. Select `NavigationRegion2D`, give it a new `NavigationPolygon`, and with the pen tool draw the outer outline through (24, 24), (976, 24), (976, 576), (24, 576). Then draw a second outline, a hole, through (376, 176), (624, 176), (624, 424), (376, 424). Press **Bake NavigationPolygon**. The blue mesh should cover the floor except for a square around the block. (If you prefer, copy the `NavigationPolygon` from the project's `main.tscn`.)

Leave the borders and the block on the default collision layer 1. Layer 1 means "walls": it is the layer that Pip's mask, the slime's mask, and the slime's line-of-sight ray all look at.

Step 6: The game script

```gdscript
# File: main.gd
# Attach to: Main (Node2D) in main.tscn
extends Node2D

const COIN_SCENE: PackedScene = preload("res://coin.tscn")
const COINS_TO_WIN: int = 5
const START_HEARTS: int = 3

@onready var player: CharacterBody2D = $Player
@onready var slime: CharacterBody2D = $Slime
@onready var coin_spots: Node2D = $CoinSpots
@onready var coins: Node2D = $Coins
@onready var info_label: Label = $HUD/InfoLabel
@onready var message_label: Label = $HUD/MessageLabel

var hearts: int = START_HEARTS
var coins_collected: int = 0
var _game_over: bool = false


func _ready() -> void:
    slime.attacked_player.connect(_on_slime_attacked_player)
    _spawn_coins()
    _refresh_hud()


func _spawn_coins() -> void:
    var spots: Array[Node] = coin_spots.get_children()
    spots.shuffle()
    for i in COINS_TO_WIN:
        var coin: Area2D = COIN_SCENE.instantiate()
        coin.position = (spots[i] as Marker2D).position
        coin.collected.connect(_on_coin_collected)
        coins.add_child(coin)


func _on_coin_collected() -> void:
    coins_collected += 1
    _refresh_hud()
    if coins_collected >= COINS_TO_WIN:
        _finish("You win! Press Enter to play again.")


func _on_slime_attacked_player() -> void:
    if _game_over:
        return
    if player.hurt():
```

```
        hearts -= 1
        _refresh_hud()
        if hearts <= 0:
            _finish("The slime got you. Press Enter to try again.")


func _finish(text: String) -> void:
    _game_over = true
    message_label.text = text
    slime.set_physics_process(false)
    player.set_physics_process(false)


func _refresh_hud() -> void:
    info_label.text = "Coins: %d / %d     Hearts: %d" % [coins_collected, COINS_TO_WIN, hearts]


func _unhandled_input(event: InputEvent) -> void:
    if _game_over and event.is_action_pressed("ui_accept"):
        get_tree().reload_current_scene()
```

`spots.shuffle()` mixes the list, so the first five items are a random selection of five different places. When the game finishes, `set_physics_process(false)` freezes both characters.

Step 7: Play and observe

Press **F5**.

1. Stay still near the bottom. Watch the label: the slime goes IDLE, PATROL, IDLE, PATROL, visiting the four corners.
2. Walk into its view. The color turns red and the label says CHASE. It follows you, going around the block.
3. Duck behind the central block. The slime loses sight, turns yellow (SEARCH), and walks to where it last saw you.
4. Let it touch you. The slime goes gray (RECOVER), Pip blinks, and a heart disappears.
5. Collect five coins to win.

Expected result

A slime that wanders around the arena along a loop of waypoints, chases Pip as soon as it sees him, goes around the central block along sensible routes, searches at the last place it saw Pip, and rests after an attack. The HUD tracks coins and hearts, and Enter restarts after a win or a loss.

Chapter summary

- Game AI is code that decides what a character does. It follows the loop sense, think, act.
- A `Timer` node counts down and emits `timeout`. Use `start`, `stop`, `One Shot`, and `create_timer` with `await` for delays and cooldowns.
- A state machine keeps behavior tidy. Use an `enum` for the states, a `match` for the per-state code, and a single `_change_state` function for every transition.
- A `NavigationRegion2D` holds the walkable area. Leave a margin for the character's radius.
- A `NavigationAgent2D` plans a path to `target_position`. Call `get_next_path_position()` and move toward it; wait a physics frame before the first query.
- When chasing a moving target, update `target_position` on a repeating timer, not every frame.
- Senses are simple checks: a distance check plus a raycast with `intersect_ray` for line of sight.
- Tune for fairness: slower enemies than the player, a pause after attacks, and invulnerability after being hit.

Review questions

1. In one sentence, what is game AI? Is it always complicated?
2. Name the three steps of the AI loop.
3. True or false: a `Timer` with `One Shot` turned off emits `timeout` repeatedly.
4. When would you pick `await get_tree().create_timer(1.0).timeout` over a `Timer` node?
5. Why does the slime code change state only through `_change_state()`?
6. A state machine has the states IDLE, CHASE, and ATTACK. A character can be in IDLE and CHASE at the same time. True or false?
7. Which statement about the navigation mesh is correct? (a) It must exactly hug the walls. (b) It should leave a margin around obstacles about as large as the character's radius. (c) It is only needed for 3D. (d) It replaces collision shapes.
8. Your enemy never moves when you call `agent.target_position = ...` inside `_ready()`. What is the likely cause and the fix?
9. What does `intersect_ray` return when the ray hits nothing, and how does the slime use that?
10. Why does the slime use a repeating `RepathTimer` while chasing instead of setting the target once?

Exercises

1. **Practice (easy):** In the Inspector, change the Slime's `chase_speed` to 250 (faster than Pip). Play. Why does the game become unfair, and what number feels right to you?
2. **Practice (easy):** Add a fifth waypoint in the Waypoints node (a `Marker2D` in group `waypoint`) at (500, 70). Watch the slime include it in its patrol loop. Does the code need to change?
3. **Apply (medium):** Add a new state `ALERT`: when the slime first sees Pip (from IDLE or PATROL), it stops for 0.5 seconds with a yellow-orange color (a "huh?" moment) before it starts chasing. You will add the name to the enum, a color to `STATE_COLORS`, a branch in `match`, a start of a timer in `_change_state`, and a timeout case.
4. **Apply (medium):** Add a second slime by instancing `slime.tscn` again at (880, 500). Does it work? Connect its `attacked_player` signal as well. Hint: connect it in `main.gd`.
5. **Challenge (hard):** Make the slime flee when Pip collects the third coin: add a `FLEE` state in which the slime walks to the waypoint farthest from Pip for 3 seconds, then goes back to IDLE. You will need a loop over the waypoints that finds the largest `distance_to`.

Common mistakes

- **The enemy does not move at all.** The navigation map is not ready yet, or no `NavigationRegion2D` has a baked polygon. *Fix:* `await get_tree().physics_frame` before the first target, and check the blue mesh is visible in the editor (turn on **Debug > Visible Navigation** in the Debug menu while running).
- **The enemy gets stuck on corners.** The navigation mesh hugs the walls but the body has a radius. *Fix:* shrink the walkable area by the radius of the character.
- **The enemy runs to where I was, not to where I am.** The target was set once. *Fix:* use a repeating repath timer while chasing.
- **The enemy sees me through walls.** The ray's mask does not include the wall layer, or the wall is on a different layer. *Fix:* check that the walls are on layer 1 and the ray's mask is 1.
- **The enemy does two things at once, or a timer fires in the wrong state.** A timer from an earlier state still runs. *Fix:* stop timers in `_change_state` before starting the new state's own.

- **The ray always reports a hit (the enemy never sees).** The ray starts inside the enemy's own collision body, or hits the player's layer. *Fix:* keep the enemy on a layer the mask does not include, or pass the enemy in the query's `exclude` list.

Further reading

- Using navigation regions: https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_using_navigationregions.html
- Using navigation meshes: https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_using_navigationmeshes.html
- Using NavigationAgents: https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_using_navigationagents.html
- Ray-casting: https://docs.godotengine.org/en/stable/tutorials/physics/ray-casting.html
- Timer class reference: https://docs.godotengine.org/en/stable/classes/class_timer.html
- NavigationAgent2D class reference: https://docs.godotengine.org/en/stable/classes/class_navigationagent2d.html

Chapter 22: The Debugger, the Profiler and Good Habits

Learning objectives

By the end of this chapter you will be able to:

- Read an error message in the Debugger panel and use the stack trace to find the line that caused it.
- Set breakpoints, step through code one line at a time, and inspect variable values while the game is paused.
- Use the Remote scene tree and the Inspector to look at and change a running game.
- Use `print`, `push_warning`, `push_error` and `assert` to make a program tell you what it is doing.
- Read the Profiler and the Monitors to decide whether a game is slow, and where.
- Save your work safely with Git (init, add, commit, log, restore) and organize a project so it stays easy to change.

Key terms

- **Bug**: a mistake in a program that makes it behave differently from what you intended.
- **Debugging**: the work of finding why a program misbehaves and fixing it.
- **Error message**: text the engine prints when it cannot do what a line of code asks.
- **Stack trace**: the list of functions that were running, newest first, when an error happened.
- **Breakpoint**: a marker on a line of code that pauses the game just before that line runs.
- **Stepping**: running a paused program one line at a time (step over, step into, continue).
- **Remote tree**: the live scene tree of the running game, shown in the editor.
- **Profiler**: a tool that measures how much time each function takes.
- **Frame time**: how many milliseconds one frame takes to compute and draw.
- **Version control**: a system that records snapshots of your project so you can go back.
- **Commit**: one saved snapshot in Git, with a short message describing it.
- **Repository**: the folder (plus its hidden history) that Git tracks.

22.1 Bugs are normal

Every programmer writes bugs, every day, for their whole career. A bug is not a sign that you are bad at this. It is a sign that you are working on something real. What changes with experience is not how many bugs you write, but how quickly you find them.

Finding a bug is a skill with a method. The method has four steps, and you will use them for the rest of your life, in games or anywhere else.

1. **Reproduce it.** Find the exact steps that make the bug happen. A bug you cannot repeat cannot be fixed.
2. **Narrow it down.** Decide which part of the program is responsible. Is the problem in the input, the calculation, or the display?
3. **Form a guess and test it.** "I think the score is not updating because the label is never refreshed." Then check that guess with a tool, not with hope.
4. **Fix one thing, then test again.** Change one thing at a time. If you change five things and the bug disappears, you do not know which one mattered.

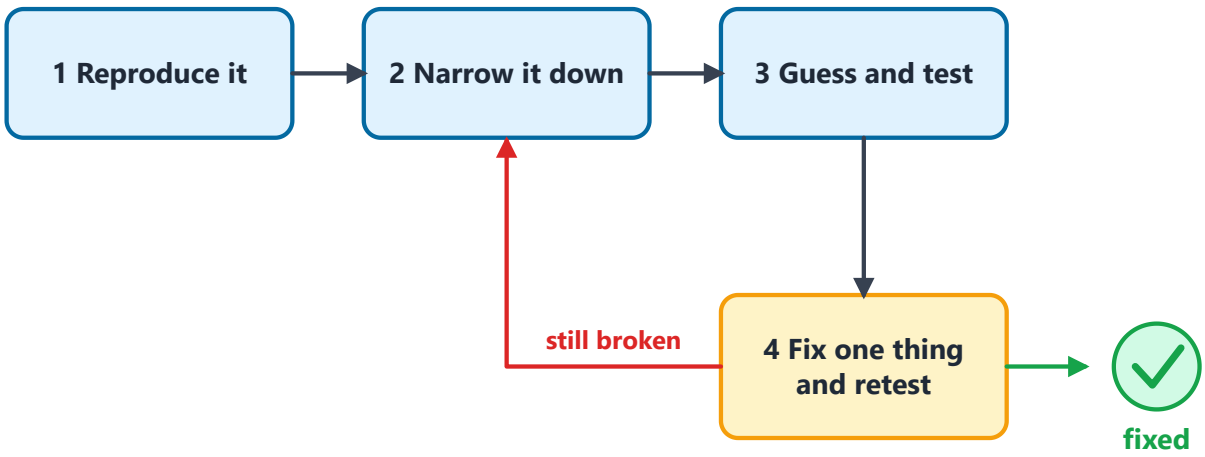


Figure 22.1: The debugging loop. If the test fails, you go around again with a better guess.

There are two broad kinds of bugs, and the tools in this chapter treat them differently.

- **Loud bugs** stop the program or print a red error. "Node not found" is a loud bug. The engine tells you where it happened.
- **Silent bugs** produce no error. The program runs perfectly, but does the wrong thing: the score stays at zero, the character moves too fast. The engine cannot know that you meant something else, so you have to investigate.

Loud bugs are the easy ones. Silent bugs are where breakpoints, printing and the Remote tree earn their keep.

> **Note:** Programmers also use the word **warning** for something the engine thinks is probably a mistake but can still run. Warnings appear in yellow in the Debugger. Treat them as free advice.

22.2 The Debugger panel and reading errors

When you press **Run Project** (F5), Godot starts your game in a separate window. The editor stays open, and the **Debugger** panel appears in the bottom bar. If you cannot see it, click the word **Debugger** in the bottom panel bar to open it.

The Debugger has several tabs. You will meet these:

- **Stack Trace**: only active when the game is paused. Shows where you are in the code.
- **Errors**: every error and warning, with the file and line.
- **Profiler**: time spent per function (section 22.6).
- **Visual Profiler**: time spent by the rendering side of the game (section 22.6).
- **Monitors**: live graphs such as frames per second and memory.
- **Misc**: extra information about the running game.

(Depending on your version and what the game uses, you may also see further tabs, such as a network profiler. You can ignore them for now.)

22.2.1 Anatomy of an error

Here is a typical runtime error. We will see it for real in the project at the end of the chapter.

```
SCRIPT ERROR: Invalid assignment of property or key 'visible' with value of type 'bool' on a base object
of type 'null instance'.
   at: use_shield (res://main.gd:51)
```

Read it in pieces.

- `SCRIPT ERROR`: the kind. A script (your GDScript) caused it.
- `Invalid assignment of property ... 'visible' ... on a base object of type 'null instance'`: what went wrong, in words. You tried to set `visible`, but the object you tried to set it on was `null`, meaning "nothing". There is no object there.
- `use_shield`: the function that was running.
- `res://main.gd:51`: the file and the line number. Double-click the error in the Debugger and the editor jumps to that line.

The most common error in all of Godot is exactly this one: a **null instance**. It almost always means "the node or variable I am using does not hold what I think it holds". In the example, the line was `$Shield.visible = true`, but the node is really named `ShieldFx`. Godot printed an earlier line that said so:

```
ERROR: Node not found: "Shield" (relative to "/root/Main").
```

Read errors from the **top**. The first error is usually the cause; the later ones are often just the fallout.

> **Why it works:** `$Shield` is shorthand for `get_node("Shield")`. When no child has that name, `get_node` prints an error and gives back `null`. Setting a property on `null` is the second error. One typo, two messages.

22.2.2 The stack trace

A **stack** is a pile where the last thing added is the first thing removed. When one function calls another, Godot remembers where it was, like leaving a bookmark. The list of bookmarks is the **call stack**.

Suppose `_unhandled_key_input` calls `use_shield`, which crashes. The stack trace shows:

```
use_shield (res://main.gd:51)
_unhandled_key_input (res://main.gd:46)
```

The top line is where it failed. The lines below tell you *how you got there*. This matters when a function is called from many places and the error only appears from one of them.

22.2.3 Error types you will see often

Message (shortened)	Usual cause	Typical fix
`Node not found: "X"`	Wrong name or path in `$X` or `get_node`	Check the name in the Scene dock; check capital letters
`... on a base object of type 'null instance'`	A variable is `null`	Find where it should have been set; check the node exists

Message (shortened)	Usual cause	Typical fix
`Invalid operands 'int' and 'String' in operator '+'`	Mixing a number with text	Convert with `int()` or `str()`, or fix the data
`Invalid access to property or key ... on a base object of type 'Dictionary'`	Wrong key name	Check spelling; use `has()` or `get()`
`Index 5 out of bounds (size 3)`	Using an array position that does not exist	Check `size()`, check your loop
`Parse Error: ...`	Script has a typing or indentation mistake	Look at the line shown; check tabs and colons
`Identifier "x" not declared`	Misspelled variable, or used outside its scope	Check the spelling and where `var` was written

22.2.4 Parse errors vs. runtime errors

A **parse error** is found *before* the game runs, while Godot reads your script. The script editor underlines the line in red, and the game will not start. A **runtime error** is found *while* the game runs, only when that line is reached. This is why the shield bug in our project hides until you press Space: the faulty line never ran before.

> **Try it:** Open any project of yours, open a script, and delete one colon at the end of an `if` line. Look at the script editor. The line is flagged immediately, and the bottom of the editor shows the error. Put the colon back.

22.2.5 Warnings

GDScript also checks for things that are *probably* mistakes: a variable that is never used, a function parameter that is never used, a value that is thrown away. These show as yellow warnings. For a parameter you intentionally ignore, start its name with an underscore, like `_delta`. That tells both Godot and the next reader "I know it is unused."

You can adjust which warnings appear in **Project > Project Settings > Debug > GDScript** (turn on **Advanced Settings** at the top right of that window if you do not see it).

22.3 Printing: the simplest debugger

Before there were fancy tools, there was `print`. It remains the fastest way to answer "did this line run, and what was the value?"

```
print("score is now ", score)
```

This is a **fragment**; put it inside any function. The text appears in the **Output** panel.

Some useful variations:

```
print("coin collected at ", coin.position)        # several things, separated automatically
print("player=", player.position, " coins=", coins.size())
print_debug("reached the shield code")             # also prints the file, line and function
print_stack()                                      # prints the call stack at this moment
```

`print_debug` and `print_stack` print where they were called from, which turns "I think this ran" into proof.

22.3.1 Warnings and errors from your own code

You can raise your own messages. They show up in the Debugger's Errors tab, just like the engine's:

```
push_warning("Coin spawned outside the screen: ", coin.position)
push_error("Score became negative!")
```

`push_warning` shows a yellow warning and `push_error` shows a red error, but neither stops the game. Use them for "this should never happen, but let me keep playing so I can look around".

22.3.2 assert: a promise the game checks

`assert` states something that must be true. If it is false, the game stops at that line (in a debug run).

```
func bonus_round() -> void:
    var rewards: Dictionary = {"gold": 10, "silver": 5}
    assert(rewards["gold"] is int, "Rewards must be numbers")
    score += rewards["gold"]
```

This is a fragment from the fixed project. The second argument is the message shown when the promise is broken. An assertion is a tripwire: you place it where a wrong value would cause trouble *later*, so you catch it *early*.

> **Warning:** `assert` is for checking your own mistakes while developing. Asserts are skipped in release exports (the engine does not run them there), so never put code you need inside an assert. Do not write `assert(do_something())`; write `var ok = do_something()` and then `assert(ok)`.

22.3.3 When printing is the wrong tool

Printing in `_process` writes 60 lines per second and buries everything. Print only when something *changes*, or use the Debugger's pause instead. Also remember to delete debug prints before sharing the game; they cost time and clutter the log.

22.4 Breakpoints and stepping

Printing shows you the past. A **breakpoint** lets you stop time.

22.4.1 Setting a breakpoint

Open a script. In the narrow strip to the left of the line numbers (the **gutter**), click beside the line you care about. A red dot appears. Clicking again removes it. You can also place the text cursor on the line and press **F9** to toggle a breakpoint (also available in the editor's **Debug** menu while the script editor is open).

You can also write the word `breakpoint` on a line of its own inside a function. It does the same thing, and it is saved in the script, so it is easy to share. Remove it when finished.

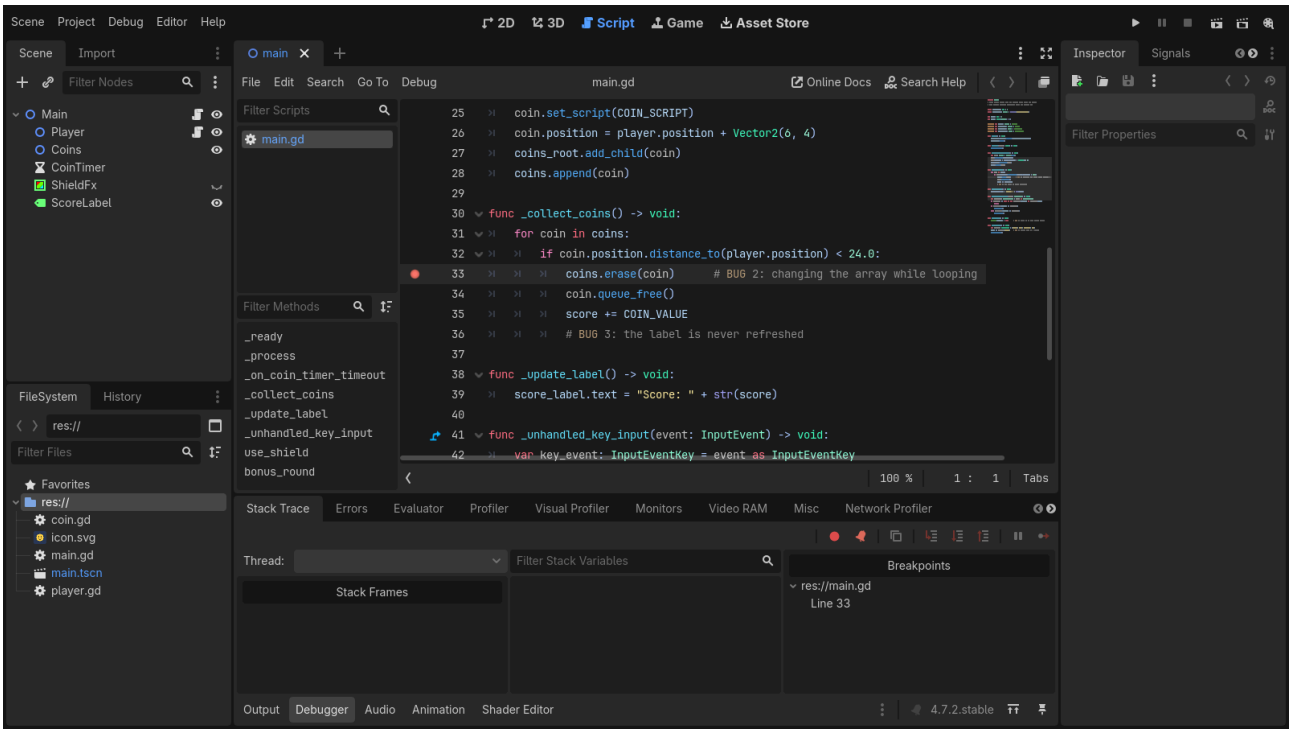


Figure 22.2: A breakpoint is the red dot left of the line number.

Now press **Run Project**. When the game reaches that line, it freezes **before** running it. The editor jumps to the script, the line is highlighted, and the Debugger's Stack Trace tab fills in.

22.4.2 Looking at variables

While the game is paused, the Debugger shows the stack variables (grouped as **Locals** and **Members**, with a **Filter Stack Variables** box) listing the values of the variables that exist right now. This is the real treasure. Instead of guessing what `score` holds, you can read it.

Open the Debugger panel, pick the function in the stack list, and expand **Locals** (variables inside this function) and **Members** (the script's own variables, such as `score`). You can also hover the mouse over a variable name in the script while paused; a small box shows its value.

22.4.3 Stepping through code

When paused, the toolbar at the top of the Debugger shows these buttons. Learn the keyboard keys; you will use them constantly:

Button	Key	What it does
Step Into	**F11**	Run one line. If the line calls a function, go inside that function.
Step Over	**F10**	Run one line. If it calls a function, run the whole function and stop on the next line.
Continue	**F12**	Resume running until the next breakpoint.
Break	(button)	Pause the running game right now, without a breakpoint.

A good rhythm: place a breakpoint just before the suspicious part, run, then press **F10** repeatedly, watching the variables change. The moment a value becomes wrong, the line you just stepped over is the culprit.

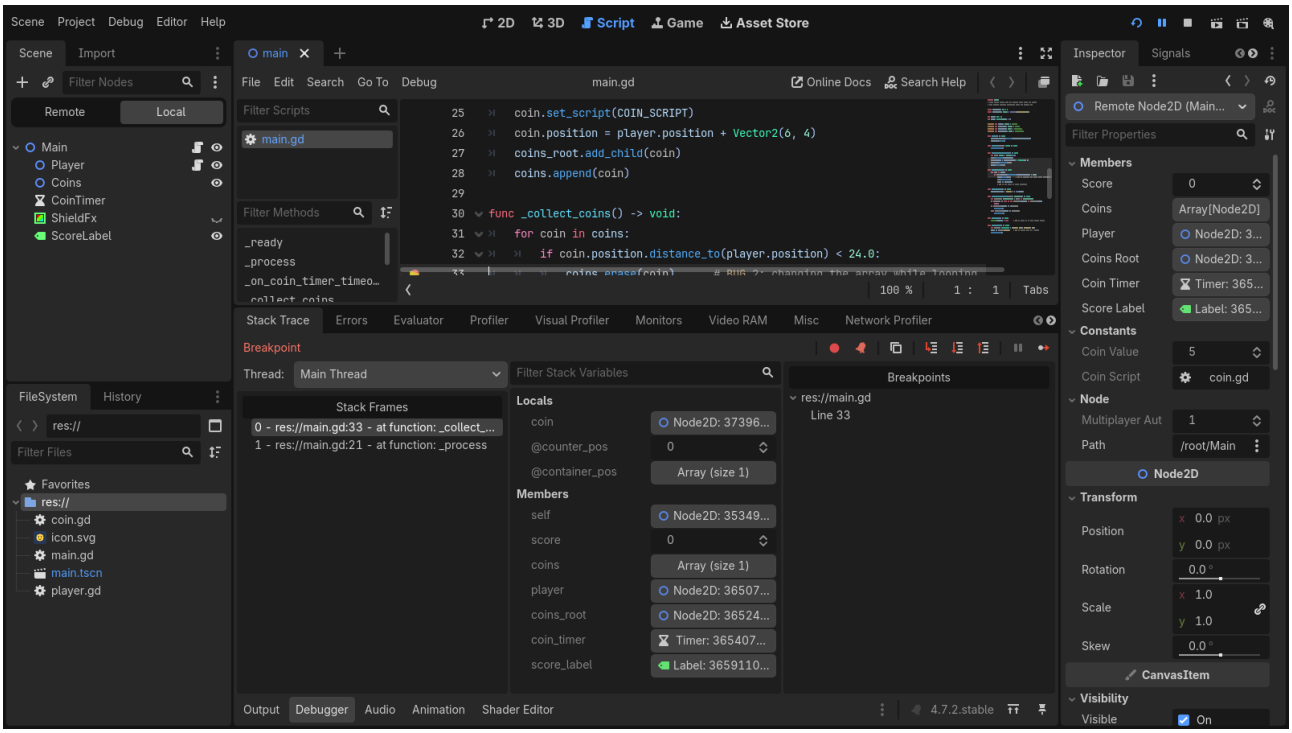


Figure 22.3: Paused at a breakpoint. The stack list is on the left, variable values are on the right.

22.4.4 Worked example: the coins that will not go away

In the buggy project, some coins seem to be ignored: Pip touches two coins at once and only one disappears. Let us hunt it with a breakpoint.

1. In `main.gd`, put a breakpoint on the line `coins.erase(coin)` inside `_collect_coins`.
2. Run the game. Walk Pip onto a coin. The game pauses.
3. Press **F10** once. The line `coins.erase(coin)` runs. Look at `coins` in Members. It is one item shorter.
4. Press **F10** a few more times. Notice that the loop `for coin in coins:` moves on, but it skips the coin that *slid into the place* of the one you removed.

Why? Imagine a row of people numbered 1, 2, 3. The loop is pointing at person 1 and removes them. Now person 2 is in position 1 and person 3 is in position 2. The loop steps forward to position 2: person 3. Person 2 was never looked at. **Never add to or remove from an array while looping over it.** The usual fix is to loop over a copy:

```
for coin in coins.duplicate():
    if coin.position.distance_to(player.position) < 24.0:
        coins.erase(coin)
```

This is a fragment from the fixed project's `_collect_coins`. `duplicate()` makes a second array with the same contents; we loop over the copy while changing the original.

22.4.5 Conditional thinking: breaking only when it matters

If a function runs 60 times a second, a breakpoint on it stops you every frame. A trick: add an `if` that only fires in the case you care about, and put the `breakpoint` keyword inside it.

```
if score < 0:
    breakpoint
```

Now the game only pauses when the score has gone negative. Remember to delete it afterwards.

22.5 The Remote tree: inspecting a running game

While your game runs, the **Scene** dock in the editor shows two buttons at the top: **Local** and **Remote**. Click **Remote**. Now the dock shows the *live* scene tree of the running game, including nodes created by code that do not exist in your saved `.tscn` file.

Click a node in the Remote tree and the **Inspector** shows its current properties. You can change a property (for example, set `visible` on `ShieldFx`) and the running game updates immediately. These changes are temporary: they affect only the running copy, and are lost when the game stops.

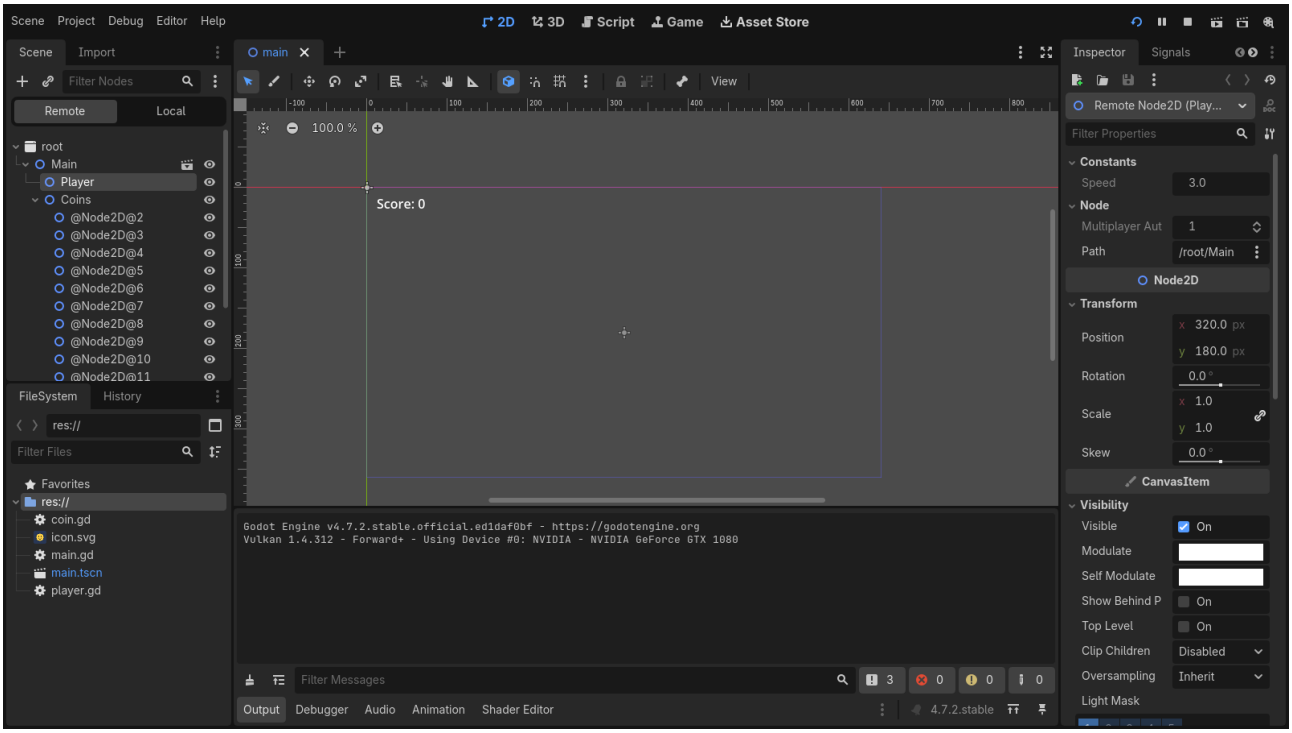


Figure 22.4: The Remote tree. Coins spawned by code appear here even though they are not in the saved scene.

This answers questions that nothing else answers quickly:

- "Is my enemy even in the tree?" Look for it.
- "Why is this coin invisible?" Click it and read `position`, `visible`, `modulate`.
- "Did my `queue_free()` work?" Watch the node vanish.
- "How many bullets exist?" Count the children.

> **Try it:** Run the buggy project. Switch the Scene dock to **Remote**. Click `Player` and watch `position` change as you press the arrow keys. Then click **Coins**: its children list grows once per second as coins spawn. Wait a few seconds, then count them.

Two more helpers are in the editor's **Debug** menu. **Visible Collision Shapes** draws the outlines of collision shapes in the running game, which is perfect for "why does my hit not register?" questions from Chapter 8. **Visible Navigation** does the same for navigation meshes from Chapter 21. Both only affect debug runs.

22.6 The profiler and performance

Sometimes the problem is not a wrong result but a slow one. The game stutters, the fan spins, the phone gets hot. The first rule of speeding up a game is: **measure first.** People are very bad at guessing what is slow. Optimizing the wrong thing costs time and often makes code harder to read for no gain.

22.6.1 Frames and frame time

A game redraws the screen many times a second. Each redraw is a **frame**. A target of 60 frames per second (FPS) gives each frame about 16.7 milliseconds (1000 ms divided by 60) to do all its work: run scripts, move physics, and draw. If the work takes 25 ms, the game drops to about 40 FPS and feels uneven.

The number to watch is **frame time**, not FPS alone, because it shows how much room you have left.

22.6.2 Monitors

Open the Debugger's **Monitors** tab while the game runs. You see live graphs, grouped by area. Useful ones:

- **FPS** and **Process** time: how fast frames are produced and how long scripts take.
- **Physics Process** time: how long physics takes.
- **Objects** and **Nodes** counts: how many exist. A number that climbs forever, never falling, is a **leak** (you create things and never free them).
- **Memory** (static and video memory).

A very common beginner leak is spawning nodes (bullets, coins) without ever calling `queue_free()`. The Nodes counter climbs steadily. That is easy to spot here.

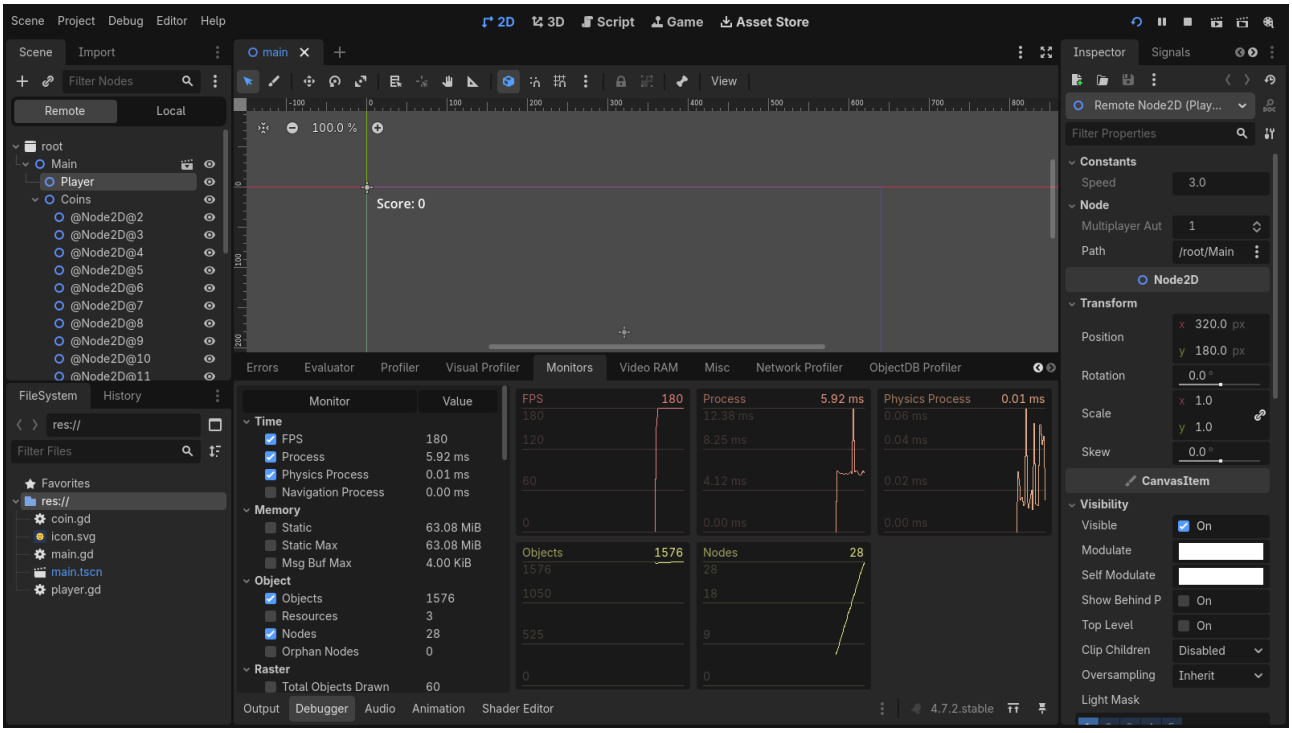


Figure 22.5: Monitors. A steadily climbing node count means something is created and never freed.

22.6.3 The Profiler

Switch to the **Profiler** tab and press **Start**. Play for several seconds, then press **Stop**. The Profiler lists functions with the time they took. Two columns matter:

- **Total**: time including every function that function called.
- **Self**: time inside that function itself.

Sort by **Self** to find the real hogs. If one function of yours eats 10 ms per frame, that is where to look. Often the fix is simple: do not search the whole tree every frame (cache the result in `_ready`), do not recompute what has not changed, and do not run expensive code in `_process` that could run once per second on a `Timer`.

The **Visual Profiler** tab (for the rendering side) shows how long the CPU and GPU spent on drawing, per frame, so you can tell *which side* is the bottleneck. For beginner projects, the main lesson is simply that these tools exist and that you can ask them.

22.6.4 Typical beginner slowdowns

- Calling `get_tree().get_nodes_in_group("x")` or `get_node` every frame instead of once.
- Thousands of nodes with `_process` functions that do nothing. Remove the function, or call `set_process(false)` when a node is idle.
- Printing in `_process`.
- Huge textures for small things; many separate lights or shader effects at once.
- Forgetting to free nodes that left the screen.

> **Note:** Do not worry about speed in small games. A game of coins and circles will run fast on almost anything. Learn the tools now so they are familiar on the day you need them.

22.7 Good habits: version control with Git

Imagine spending a weekend on a feature, and in the end, the game no longer starts, and you cannot remember what you changed. **Version control** prevents this. It records snapshots of your whole project over time, so you can look at the history, compare, and return to any earlier point.

The most widely used tool is **Git**. It is free, works offline, and many websites (GitHub, GitLab, Codeberg) can store a copy of your project online as a backup.

> **Note:** This section teaches the few commands you need. It does not cover everything Git can do, and that is fine. You can use Git for years with about eight commands.

22.7.1 Install and set up

Download Git from https://git-scm.com/ and install it with the default choices. Then open a terminal (on Windows, **Git Bash** or PowerShell; on macOS and Linux, Terminal) and tell Git who you are, once per computer:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

22.7.2 The three places

Git thinks of your work in three places:

1. **Working folder**: your project files as they are right now.
2. **Staging area**: a waiting list of the changes you have chosen to include in the next snapshot.
3. **History**: the commits, the saved snapshots.

You move changes from left to right: edit files, then `git add` (put them on the waiting list), then `git commit` (take the snapshot).

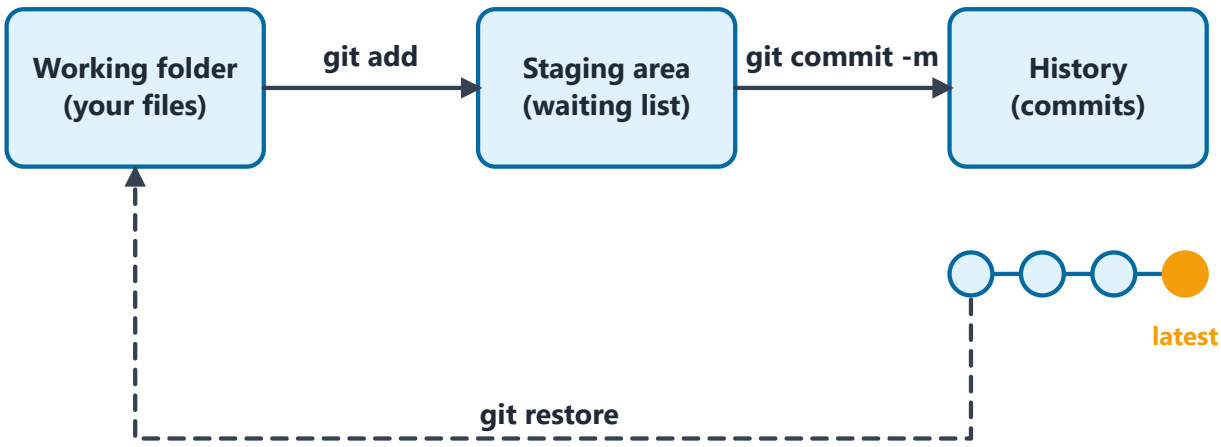


Figure 22.6: Edit, add, commit. Each commit is a snapshot you can return to.

22.7.3 Your first repository

Open a terminal **in your project folder** (the folder containing `project.godot`). Replace the path with your own:

```
cd path/to/ch22-debug-hunt-fixed
git init
```

`git init` turns the folder into a repository by creating a hidden `.git` folder. Do not delete it; it holds the whole history.

22.7.4 The .gitignore file

Godot creates a hidden folder called `.godot/` in every project. It holds imported-file caches that Godot rebuilds by itself. It is large, changes constantly, and has no value in history. Tell Git to ignore it by creating a plain text file named `.gitignore` in the project root:

```
# Godot caches (rebuilt automatically)
.godot/

# Exported games (large, can be rebuilt)
build/
export/
```

```
# Optional: files your editor or operating system creates
.DS_Store
Thumbs.db
```

Commit the `.gitignore` itself. Also **commit the `.uid` files** that Godot (4.4 and later) creates next to scripts, and the `.import` files next to images. They help Godot keep references working when you move or rename files.

> **Warning:** Never commit passwords, API keys or private tokens. If you ever put one in a file and commit it, treat it as leaked even if you delete it later, because it stays in the history. Keep secrets outside the project folder.

22.7.5 Snapshots

```
git status                # what changed?
git add .                 # put every change on the waiting list
git commit -m "First working version"
```

`git status` is the command to run whenever you are unsure what state things are in; it is safe and changes nothing. The message after `-m` should say *what* changed in a few plain words. "Fix coin skipping bug" is useful; "stuff" is not.

After you edit more files:

```
git status
git add .
git commit -m "Add shield toggle"
```

Look at the history:

```
git log --oneline
```

Each line shows a short code (the commit's name) and its message. Press **q** to leave the list.

22.7.6 Looking at and undoing changes

To see exactly what you changed since the last commit:

```
git diff
```

Lines starting with `-` were removed; lines starting with `+` were added. To throw away your changes to **one file** and go back to the last commit:

```
git restore main.gd
```

> **Warning:** `git restore` discards uncommitted changes for good. Git cannot bring back something that was never committed. This is a strong reason to commit often.

To look at an old version without losing anything, use `git switch --detach <code>` with a code from `git log --oneline`, and return with `git switch -`. If this feels like too much for now, simply remember that your history exists; asking for help later is easy because everything is recorded.

22.7.7 Branches in one paragraph

A **branch** is a separate line of history. You can try a risky idea ("rewrite the player movement") on a branch, and if it fails, drop the branch. Create one and move to it with `git switch -c try-new-movement`. Go back with `git switch main` (the default branch is called `main` in recent Git versions; on some setups it is `master`). When the idea works, bring it in with `git merge try-new-movement` while on `main`. Do not worry about merge conflicts until they happen; Git marks them in the file and you choose which side to keep.

22.7.8 A backup online

Create an empty private repository on a site such as GitHub, copy its address, then:

```
git remote add origin https://github.com/YOUR-NAME/YOUR-REPO.git
git push -u origin main
```

From then on `git push` sends new commits to the site. The exact sign-in steps change over time, so follow the site's own instructions for access tokens or its desktop sign-in.

22.7.9 Commit habits

- Commit when something works, however small. "Player moves" is a good commit.
- One idea per commit.
- Commit before a risky change, so you can always return.
- Write messages that finish the sentence "This commit ...".

Godot's editor also has an optional Git integration: **Project > Version Control > Version Control Settings...**. It is optional and the terminal is clearer for learning, so we do not rely on it here.

22.8 Good habits: organizing a project

A project of ten files can live anywhere. A project of three hundred needs a plan. Set up the plan at the start; moving things later is possible but annoying.

22.8.1 Folder layout

A layout that works for small and medium games:

```
my-game/
  project.godot
  icon.svg
  .gitignore
  scenes/
    main.tscn
    player/
      player.tscn
      player.gd
    enemies/
      slime.tscn
      slime.gd
    ui/
      hud.tscn
      hud.gd
  autoload/
    game_state.gd
  assets/
    art/
```

```
  audio/
  fonts/
resources/
  items/
build/            (ignored by Git)
```

The idea is to keep a scene and its script **together** in one folder (the *feature folder* style), and keep shared things (assets, autoloads) in their own folders. The alternative, putting all scenes in `scenes/` and all scripts in `scripts/`, also works. The one real mistake is having **no** rule.

> **Warning:** Move and rename files inside Godot's **FileSystem** dock, not in your operating system's file explorer. Godot then updates all references automatically. If you move files outside Godot, scenes may report missing dependencies.

22.8.2 Naming

- Files and folders: `snake_case` (lowercase with underscores): `slime_enemy.gd`. This avoids case problems, because Windows ignores capital letters in file names but Linux and Web exports do not. A file called `Player.png` loaded as `player.png` works on your Windows computer and breaks on the Web build.
- Node names: `PascalCase` (`ScoreLabel`, `CoinTimer`).
- Variables and functions: `snake_case`. Constants: `UPPER_SNAKE_CASE`. Signals: past tense (`coin_collected`).
- No spaces in file or folder names.

22.8.3 Small functions, clear names

A function that is 100 lines long is hard to debug, because a breakpoint stops you in the middle of a wall of text. Split it by purpose: `_collect_coins`, `_update_label`, `use_shield`. Name a function for what it *does*. A variable called `d` tells you nothing; `distance_to_player` explains itself.

22.8.4 Comments: why, not what

`score += 5 # add five to score` repeats the code. `score += COIN_VALUE` needs no comment at all. Write comments for the *reason* ("duplicate() so erasing is safe while looping"), not the mechanics.

22.8.5 Use static types and constants

Typed variables (`var speed: float = 200.0`) make Godot catch whole families of mistakes before the game runs. Replace "magic numbers" with named constants: `SPEED` instead of a bare `3.0` hidden in an expression. Then the next reader (you, in three months) knows what the number means, and you can change it in one place.

22.8.6 Other habits that pay off

- **Test small and often.** Run the game after every few lines, not after an hour of writing. When something breaks, you know it was the last few lines.
- **Keep a `NOW.md` or to-do file** with what you are doing, what is next, and ideas for later. It stops ideas from being lost and stops the feeling of being behind.
- **Back up** to a second place (the online Git repository counts).
- **Take breaks.** Many bugs are found within a minute after a short walk.
- **Explain the bug aloud** to a friend, a pet or a rubber duck. Putting the problem into words often reveals it. Programmers call this *rubber duck debugging*.

- **Read the whole error message** before changing anything.

Project 22: The Debug Hunt

You will find and fix five bugs in a small game. The buggy project is in `projects/ch22-debug-hunt/`. A fixed version is in `projects/ch22-debug-hunt-fixed/` for you to compare against at the end. Try hard to fix the bugs yourself first.

What the game should do

Pip (a yellow circle) moves with the arrow keys. A gold coin appears every second at a random place. Walking over a coin collects it, adds 5 to the score, and the score label updates. **Space** toggles a blue shield around Pip. **Enter** gives a bonus of 10 points.

Files

```
projects/ch22-debug-hunt/
  project.godot
  icon.svg
  main.tscn
  main.gd
  player.gd
  coin.gd
  .gitignore
```

Scene tree (`main.tscn`)

```
Main (Node2D)            main.gd
  Player (Node2D)        player.gd
  Coins (Node2D)
  CoinTimer (Timer)      wait_time = 1.0
  ShieldFx (ColorRect)   visible = false
  ScoreLabel (Label)
```

Steps

1. Open `projects/ch22-debug-hunt/` from the Project Manager (**Import** and select its `project.godot`) and press **Run Project** (F5). The game starts. There are no errors yet. This is typical: the loud bugs sleep until you trigger them.
2. **Bug 1, silent: Pip is far too fast.** Move around. Pip flies across the screen, and on a fast monitor even faster. Open `player.gd` and read `_process`. The line is `position += direction * SPEED`. It never multiplies by `delta` (Chapter 7). Fix it by changing `SPEED` to `200.0` and the line to `position += direction * SPEED * delta`, and rename the parameter `_delta` to `delta` because you now use it.
3. **Bug 2, silent: coins are skipped.** Use the breakpoint walk-through from section 22.4.4. Fix `main.gd` by looping over `coins.duplicate()`.
4. **Bug 3, silent: the score label never changes.** Collect a coin. The score variable grows (check with a `print`), but the label stays at "Score: 0". The function `_update_label` exists but is never called after collecting. Add `_update_label()` inside the `if` in `_collect_coins`, and once at the end of `_ready`.
5. **Bug 4, loud: Space gives "Node not found".** Press Space. The Debugger shows the error. Double-click it. The line is `$Shield.visible = true`. Open the Scene dock: the node is named `ShieldFx`. Add `@onready var shield_fx: ColorRect = $ShieldFx` near the top, and rewrite `use_shield` as

`shield_fx.visible = not shield_fx.visible`. Also make the shield follow Pip: in `_process`, add `shield_fx.position = player.position - shield_fx.size / 2.0`.

6. **Bug 5, loud: Enter gives "Invalid operands 'int' and 'String'".** The dictionary in `bonus_round` holds `"10"` (text) instead of `10` (a number). Remove the quotation marks, and add an `assert` as in section 22.3.2.
7. Run the game. Collect several coins, press Space twice, and press Enter. Everything should work and the Errors tab stays empty.
8. Open the **Monitors** tab and note the **Nodes** count while coins pile up. Why does it go up? (Answer: coins that are never collected stay in the scene. Not a bug, but think about what a real game would do with them.)
9. Create a Git repository in your fixed copy and make three commits (section 22.7): "Initial buggy version", "Fix movement and coin skipping", "Fix label and shield". Use `git log --oneline` to see them.
10. Compare your result with `projects/ch22-debug-hunt-fixed/`. Differences are fine as long as the behavior is correct.

Expected result

Pip moves at a steady speed on any monitor and cannot leave the window (the fixed version clamps position). Every coin Pip touches disappears, the label counts up in steps of 5, Space shows and hides a blue square around Pip, Enter adds 10, and the Debugger's Errors tab is empty after all of that.

Chapter summary

- Debugging is a method: reproduce, narrow down, guess and test, fix one thing and retest.
- Read errors from the top. The message says what happened; the file and line say where; the stack trace says how you got there.
- A "null instance" error nearly always means a node or variable is not what you thought.
- `print`, `print_debug`, `push_warning`, `push_error` and `assert` let your code report on itself.
- Breakpoints (F9) freeze the game; F10 steps over, F11 steps into, F12 continues, and the variable list shows real values.
- The Remote tree shows and edits the live scene tree of the running game.
- Measure before optimizing: Monitors show trends, the Profiler shows which function takes time.
- Git keeps snapshots: `git init`, `git add .`, `git commit -m`, `git log --oneline`; ignore `.godot/` and `build/`.
- A clear folder layout, consistent names and small functions make future bugs easier to find.

Review questions

1. What are the four steps of the debugging loop, in order?
2. In your own words, what is the difference between a loud bug and a silent bug? Give one example of each from this chapter.
3. An error says `Invalid assignment of property or key 'text' ... on a base object of type 'null instance'`. What does `null instance` tell you, and what is the first thing you check?
4. True or false: When several errors are printed, you should start with the last one.
5. Multiple choice: Which key steps over a line without going inside a function it calls? (a) F9 (b) F10 (c) F11 (d) F12
6. What does the stack trace show, and why is the second line sometimes more useful than the first?
7. True or false: Changes you make to a node in the Remote tree are saved to your scene file.

8. Why is it dangerous to call `coins.erase(coin)` inside `for coin in coins:`, and what is the usual fix?
9. Multiple choice: Which Git command takes a snapshot of the changes you have put on the waiting list? (a) git init (b) git status (c) git commit (d) git diff
10. Why should `.godot/` be listed in `.gitignore`, and why should file names use lowercase `snake_case`?

Exercises

1. **Practice (easy):** In the fixed project, add a `print` to `_collect_coins` that shows the score each time a coin is collected. Watch the Output panel. Then replace it with `print_debug` and notice what extra information appears. Remove it when you are done.
2. **Practice (easy):** Set a breakpoint on the first line of `use_shield`. Run, press Space, then use F10 to step. Write down the value of `shield_fx.visible` before and after the toggling line.
3. **Apply (medium):** Add a `push_warning` to `_on_coin_timer_timeout` that fires when more than 20 coins exist. Let the game run until the warning appears in the Errors tab. Then add a limit so the coin count never grows past 20.
4. **Apply (medium):** Write a `.gitignore` for one of your own earlier projects, create a repository, and make a first commit. Run `git status` and confirm that `.godot/` is not listed.
5. **Challenge (hard):** Break the fixed project on purpose in three different ways (a wrong node name, a wrong type, a wrong dictionary key). For each one, write down the exact error message, the line it points to, and how you found it. Swap with a friend and let them find your bugs using only the Debugger.

Common mistakes

- **Symptom:** You fix an error and a new one appears at the same line. **Cause:** You only read the last message, which was fallout. **Fix:** Scroll to the first error and fix that one.
- **Symptom:** The breakpoint never stops the game. **Cause:** The line never runs (wrong scene, wrong condition) or you ran the game without the debugger (for example, an exported build). **Fix:** Add a `print` above it to check that the line is reached; run from the editor.
- **Symptom:** The Remote tree is empty or the buttons are missing. **Cause:** The game is not running. **Fix:** Run the project first; the Remote tab only works for a live game.
- **Symptom:** Your code works in the editor but a node is "not found" in the exported Web game. **Cause:** A file or node name used a different capital letter than the real name, and the platform cares about case. **Fix:** Match capitalization exactly; use `snake_case` file names.
- **Symptom:** `git add .` adds thousands of files and the repository is huge. **Cause:** No `.gitignore` for `.godot/` or `build/`. **Fix:** Create `.gitignore`, then run `git rm -r --cached .godot` (once) and commit.
- **Symptom:** After moving a script in the file explorer, the scene says "missing dependencies". **Cause:** The move happened outside Godot. **Fix:** Move it back, then move it inside the FileSystem dock.

Further reading

- https://docs.godotengine.org/en/stable/tutorials/scripting/debug/overview_of_debugging_tools.html
- https://docs.godotengine.org/en/stable/tutorials/scripting/debug/debugger_panel.html
- https://docs.godotengine.org/en/stable/tutorials/performance/index.html
- https://docs.godotengine.org/en/stable/tutorials/best_practices/project_organization.html
- https://docs.godotengine.org/en/stable/tutorials/best_practices/version_control_systems.html
- https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_styleguide.html

Part V: Finishing and sharing

Chapter 23: Exporting Your Game

Learning objectives

By the end of this chapter you will be able to:

- Explain the difference between running a game in the editor and exporting it.
- Install export templates and create export presets for Windows, Linux and Web.
- Set a project name, icon and window settings that suit a finished game.
- Export a game from the editor and from the command line, and test the result outside Godot.
- Choose between debug and release exports, and between an embedded and a separate `.pck` file.
- Run a Web export in a browser the correct way, and avoid the most common export mistakes.

Key terms

- **Export**: turning your project into a stand-alone program that runs without the Godot editor.
- **Export template**: a prebuilt, stripped-down copy of the engine that Godot copies into your export together with your game's data.
- **Export preset**: a saved set of export choices for one platform (stored in `export_presets.cfg`).
- **PCK file** (`.pck`): a package holding all your game's scenes, scripts and resources.
- **Debug export**: a build with debugging features switched on, bigger and slightly slower.
- **Release export**: the final build for players.
- **Feature tag**: a word such as `web`, `windows` or `editor` that tells code which kind of build is running.
- **Embedded PCK**: the game data stored inside the executable itself, giving one file.
- **Headless**: running Godot without a window, which is how command-line exports work.
- **HTTP server**: a small program that serves files to a browser, needed to test Web exports.
- **Bundle identifier / product name**: names that identify your game to the operating system.

23.1 From editor to executable

Until now every game you made ran by pressing **Run Project** inside Godot. That works because the editor itself is the engine. Your players do not have the editor, and you do not want to ask them to install it. You want to give them something to double-click.

Exporting produces that. When you export, Godot combines two things:

1. An **export template**: the engine, compiled for a specific platform (Windows, Linux, Web…) and without the editor parts, so it is much smaller.
2. Your **game data**: scenes, scripts, images, sounds and project settings, packed into a single `.pck` file.

For Windows the result is an `.exe`. For Linux it is a program file. For Web it is a small folder of files (`index.html`, `index.js`, `index.wasm`, `index.pck`, and a few more) that a browser can run.

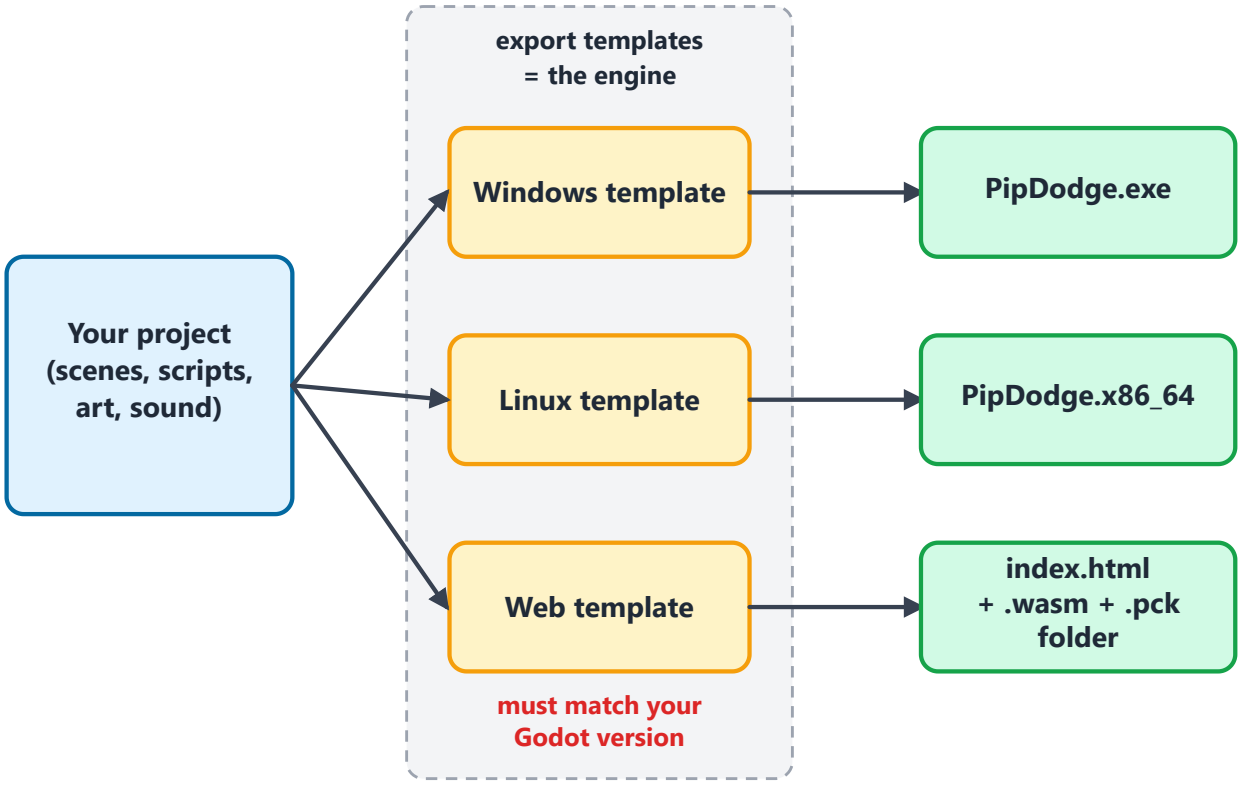


Figure 23.1: Your project plus a platform's export template becomes a build for that platform.

An important consequence: you export **once per platform**. Each build only runs on its own kind of system. A Windows `.exe` will not run on Linux, and neither will play in a browser.

> **Note:** In this chapter we export a tiny game, *Pip Dodge*, to Windows, Linux and Web. The Windows steps were carried out and the finished build was run. The Linux and Web steps follow Godot's official documentation; if a menu on your screen looks different, check the export page of the Godot manual for your version.

23.2 Export templates

Export templates are **not** included in the Godot download. They are a separate, large download (more than a gigabyte for all platforms), matched to your exact Godot version. If the versions differ, Godot refuses to export.

23.2.1 Installing the templates

The easy way, from inside the editor:

1. Open the **Editor** menu and choose **Manage Export Templates...**.
2. The window is titled **Export Template Manager**. Pick your version (for example **4.7.2.stable**) in the **Godot Version** list and press **Install All Templates** (it needs an internet connection). Godot downloads from the mirror shown in **Download from:**.
3. Wait until the version shows up under **Installed Templates**.

If your connection is slow or restricted, the manual way works too. On the official download page (https://godotengine.org/download), get the file called **Export Templates** (a `.tpz` file). Then, in the same **Manage Export Templates** window, click the small file icon at the top right (install from a `.tpz` file) and

select it.

Templates are stored in a folder in your user data, in a subfolder named after the version. On Windows that is `%APPDATA%\Godot\export_templates\4.7.2.stable\` for the version used while writing this book. You never need to touch it, but it is useful to know where to look if Godot says that templates are missing.

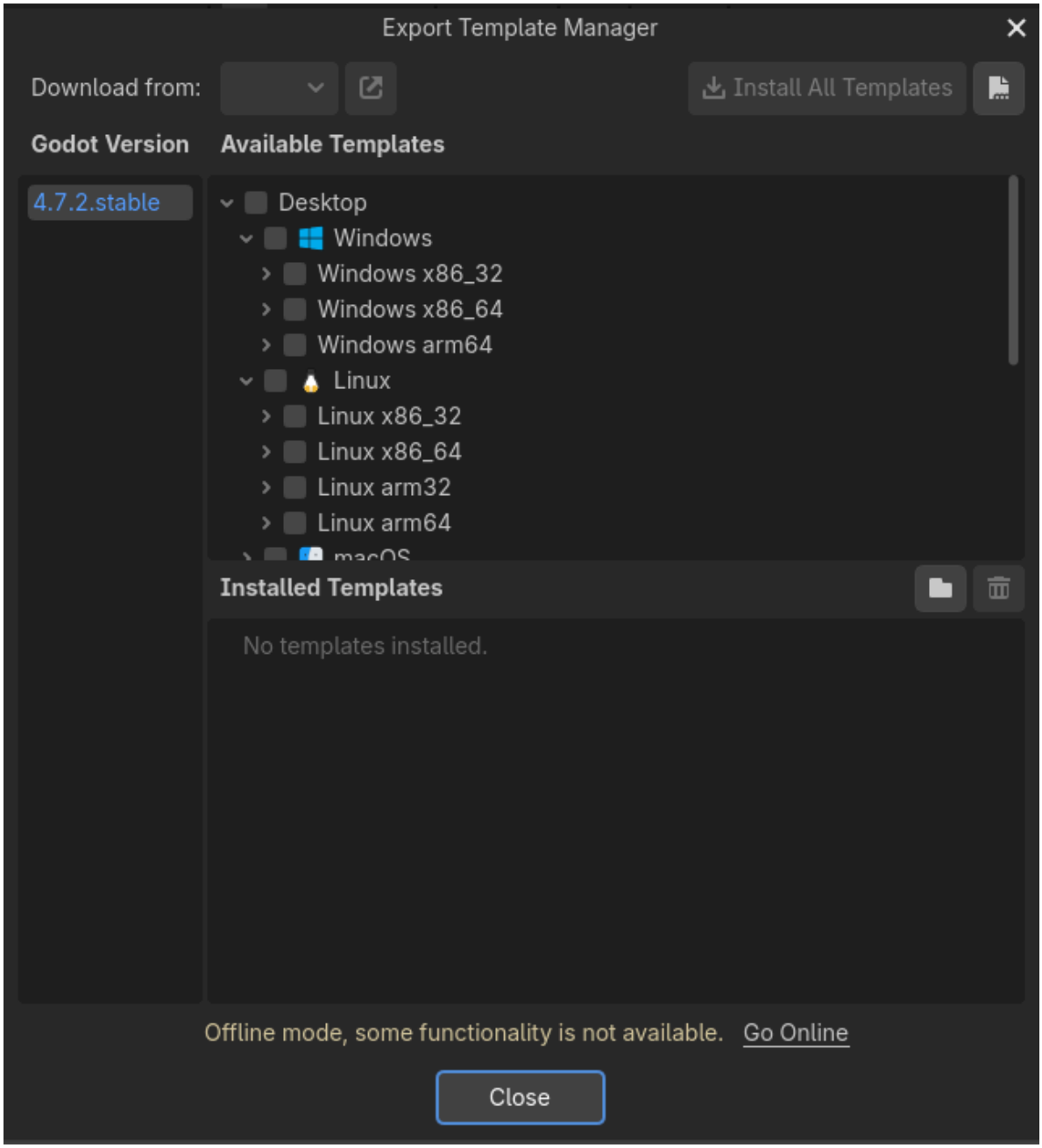


Figure 23.2: Export Template Manager (Editor > Manage Export Templates...). Here no templates are installed yet, so Installed Templates reads "No templates installed."

23.2.2 What is inside

The template file contains one entry per platform and mode. Debug and release variants exist for each. For Windows 64-bit, for example, there are `windows_debug_x86_64.exe`, `windows_release_x86_64.exe` and matching `_console.exe` variants. For Linux there are `linux_debug.x86_64` and `linux_release.x86_64`; for Web, several `web_*.zip` files. Godot picks the right one for you.

> **Warning:** If you update Godot to a new version, download the new templates too. Using templates from another version causes an error, or worse, a game that behaves oddly.

23.3 Preparing the project

An export is a snapshot of your project. Before you take it, get the project ready. A few minutes here makes the game look finished.

23.3.1 Project settings

Open **Project > Project Settings...**. In the **Application > Config** section:

- **Name**: the game's name. This becomes the window title and the default name in some places. Use the real title, for example `Pip Dodge`.
- **Description**: a sentence about the game.
- **Icon**: the picture used for the game. A new project uses `icon.svg`. Replace it with your own square image, ideally at least 256 by 256 pixels, with a simple readable shape.

In **Application > Run**, check the **Main Scene**. The export uses it as the first scene; if it is empty, the exported game starts and shows nothing.

In **Display > Window**:

- **Viewport Width/Height**: the size of the game's own coordinate space. Pip Dodge uses 640 by 360.
- Under **Stretch**: set **Mode** to `canvas_items` and **Aspect** to `keep`. These two lines make the game scale to any window size and keep its proportions, adding black bars if needed. This is a very good default for 2D games.
- **Mode** (the window mode) lets a game start fullscreen if you like. For jam games, a normal window is friendlier.

If you cannot see all of these settings, turn on **Advanced Settings** at the top right of the Project Settings window.

23.3.2 The project file

These settings live in `project.godot`. The relevant lines of Pip Dodge look like this (a fragment):

```
[application]

config/name="Pip Dodge"
config/description="A tiny dodge game used to practise exporting."
run/main_scene="res://main.tscn"
config/icon="res://icon.svg"

[display]

window/size/viewport_width=640
window/size/viewport_height=360
window/stretch/mode="canvas_items"
window/stretch/aspect="keep"
```

You rarely edit this file by hand; the Project Settings window writes it. Reading it once helps you see that "settings" are just text.

23.3.3 A pre-flight checklist

Before exporting, run through this list:

1. The game starts from the main scene and can be played from start to finish.
2. There are no red errors in the Debugger (Chapter 22).
3. Delete debug `print` calls and test hotkeys that you do not want players to find.
4. Every file name uses consistent lowercase letters (case matters on Linux and Web).
5. Anything the game saves goes to `user://`, never to `res://` (the packed game data is read-only once exported). You learned this in Chapter 17.
6. You have a Git commit of the working state (Chapter 22).

23.3.4 Feature tags: code that knows where it runs

Sometimes the same code must behave differently depending on the build. Godot provides **feature tags**, checked with `OS.has_feature("tag")`. Some tags: `editor` (running from the editor), `debug`, `release`, `web`, `windows`, `linux`, `macos`.

Pip Dodge uses them to display what kind of build is running. This function is part of `main.gd` in the project (a fragment):

```
func _build_info_text() -> String:
    var build_kind: String = "editor run"
    if OS.has_feature("editor"):
        build_kind = "editor run"
    elif OS.has_feature("debug"):
        build_kind = "debug export"
    else:
        build_kind = "release export"
    var where: String = OS.get_name()
    if OS.has_feature("web"):
        where = "Web browser"
    return "%s on %s | Godot %s" % [build_kind, where, Engine.get_version_info()["string"]]
```

`OS.get_name()` returns the operating system name as text. `Engine.get_version_info()` returns a dictionary, and its `"string"` entry is the readable version. The `%s` markers are filled in order from the list after the `%` sign.

A classic use: hide a **Quit** button in Web builds, because a browser page cannot "quit":

```
func _ready() -> void:
    $QuitButton.visible = not OS.has_feature("web")
```

(This is a fragment; `QuitButton` stands for a Button node in your own scene.)

23.4 Creating an export preset

An **export preset** records your choices for one platform. You can have several presets for the same platform too (for example a debug build and a release build).

1. Open **Project > Export...**. The Export window appears.
2. Press **Add...** and choose a platform from the list, for example **Windows Desktop**.
3. Godot adds a preset on the left. Select it to edit its settings on the right.

The right side has tabs. The ones you need are:

- **Options**: platform-specific choices (architecture, name, icon, and so on).
- **Resources**: which files go into the export.
- **Features**: lets you see feature tags for this preset and add your own custom tags.

(Other tabs, such as ones for encryption or version information, may also be present depending on the version. Leave them alone for now.)

If the window shows a red message at the bottom such as "No export template found", go back to section 23.2.

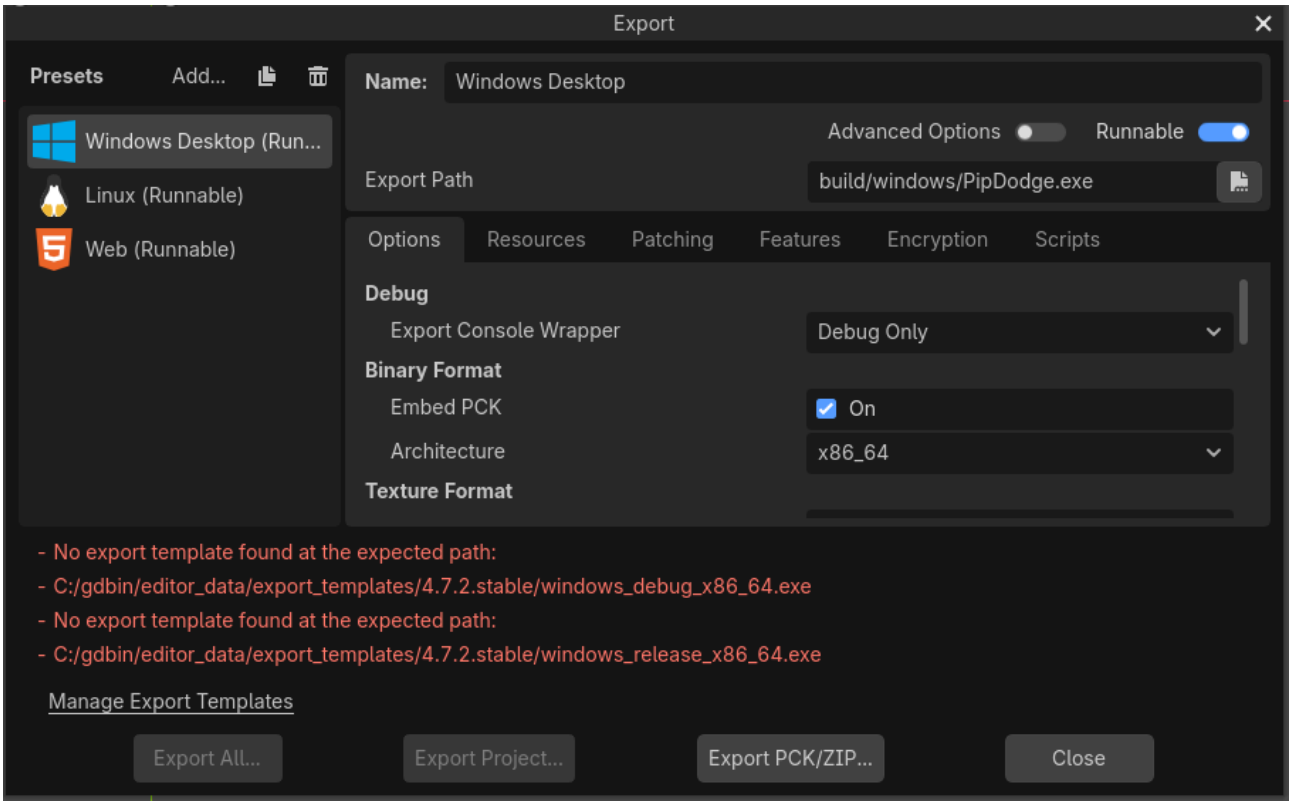


Figure 23.3: The Export window. Presets are on the left, settings on the right, export buttons at the bottom.

23.4.1 The Resources tab

By default the export includes **all resources** in the project that the game uses, which is what you want in nearly every case. The **Export Mode** choices let you export only selected scenes or resources; leave the default unless you have a reason.

The **Filters to exclude files/folders** box takes patterns, such as `*.md` or `docs/*`, for files you want to leave out (design documents, notes). It is good practice to exclude things that only you need.

23.4.2 The preset file

All presets are stored in `export_presets.cfg` in the project folder. It is a plain text file. A trimmed fragment for Pip Dodge:

```
[preset.0]

name="Windows Desktop"
platform="Windows Desktop"
runnable=true
export_filter="all_resources"
export_path="build/windows/PipDodge.exe"
```

```
[preset.0.options]

binary_format/embed_pck=true
binary_format/architecture="x86_64"
application/product_name="Pip Dodge"
```

The full file is in `projects/ch23-export-demo/export_presets.cfg`. Because it is plain text, it can go into Git, and you can copy it to another project.

> **Warning:** Presets can contain passwords or keys for signing and encryption on some platforms. If you ever fill in such fields, keep that file out of a public repository. Godot stores certain secrets elsewhere, but check before you share.

23.5 Exporting to Windows

23.5.1 In the editor

1. In the Export window, select your **Windows Desktop** preset.
2. In the **Options** tab, note **Binary Format**: the **Embed PCK** switch decides whether the game data lives inside the `.exe` (one file) or in a separate `.pck` next to it. For small games, embedding is convenient: one file to share.
3. Check that **Architecture** is `x86_64` (the usual 64-bit PC type).
4. Under **Application**, fill in **Product Name**, **Company Name**, **File Description** and the version fields if you like. These show in the file's properties in Windows.
5. Press **Export Project...** at the bottom. A file dialog asks where to save. Choose a folder (such as `build/windows`) and a name such as `PipDodge.exe`.
6. The **Export With Debug** checkbox in that dialog decides between a debug and release build. For the version you give to players, make sure it is **off**.

If everything is right, Godot packs the files and finishes. Double-click the `.exe` to test it.

23.5.2 The icon

The **Application > Icon** option in the Windows preset is for the icon that Windows shows for the `.exe`. Embedding an icon into an `.exe` file requires changing the file's resources, and Godot historically needed an outside helper program called **rcedit** for this, which you set in **Editor > Editor Settings** under the Windows export options. This book's author did not test icon embedding, so this part is **unverified**. If you leave it empty, the game still runs; the file simply gets a default icon. Check the current documentation (linked at the end of this chapter) for the up-to-date requirements on your Godot version.

The in-game window icon is separate: it comes from **Application > Config > Icon** and does not need rcedit.

23.5.3 Windows SmartScreen

When you send an unsigned `.exe` to someone, Windows may show a blue warning ("Windows protected your PC"). This is normal for programs from small unknown publishers: it does not mean anything is wrong with your game. Players can choose **More info** and then **Run anyway**. Removing the warning completely requires buying a code-signing certificate, which is outside the scope of a beginner's project. For itch.io releases, the Web version avoids this problem altogether, and you can mention the warning on your page.

23.5.4 What the test showed

With the preset above, a command-line export of Pip Dodge on Windows produced a single `PipDodge.exe` of about 109 MB (most of that is the engine). The exported file started and exited cleanly when launched with `--headless --quit-after 60`. A debug export additionally produced a `.console.exe` next to the exe, a console-window variant that prints messages while the game runs, useful when a build misbehaves on a friend's computer.

23.6 Exporting to Linux

The Linux preset is created the same way: **Add… > Linux**. With **Embed PCK** on, you get a single program file, for example `PipDodge.x86_64`.

Details to know:

- The architecture is normally `x86_64` for ordinary PCs.
- On Linux, a downloaded file may not be marked as runnable. The player runs `chmod +x PipDodge.x86_64` in a terminal, or marks it as executable in the file manager's properties, and then starts it. Putting the file in a `.zip` loses the executable mark on some systems, so include a short note in your instructions.
- You do not need a Linux computer to *create* the build; the export on a Windows computer produces a file of roughly 70 MB. But you do need a Linux computer, or a virtual machine, to *test* it. Ask a friend, or use a virtual machine.

> **Note:** Exporting to macOS, Android and iOS needs extra tools and accounts (for example signing certificates, an Android SDK, or an Apple developer account). They are beyond this chapter. The official documentation has full guides for each.

23.7 Exporting to the Web

A Web export lets people play in a browser with nothing to install. It is the best way to reach the most people, and it is what itch.io uses for in-page play.

23.7.1 Create the preset

Project > Export… > Add… > Web. The settings you will use:

- **Thread Support**: off is the simple and safe choice. In recent Godot 4 versions, single-threaded Web export is the default because it works on more hosts. Turning thread support on lets the game use several CPU threads but requires the hosting server to send special security headers (cross-origin isolation); many sites do not by default. Leave it off unless you know you need it.
- **Progressive Web App**: lets visitors "install" the page. Off for a first release.
- **HTML > Canvas Resize Policy**, **Focus Canvas on Start** and similar options: defaults are fine.
- The export path must end with `index.html`, for example `build/web/index.html`. Name it `index.html` exactly: web servers, including itch.io's, look for that name.

23.7.2 The folder must already exist

This catches many people. Godot does **not** create the destination folder for you. If you export to `build/web/index.html` and `build/web/` is missing, the export fails with a message like "Target folder does not exist or is inaccessible". Create the folder first. The same happened for the Windows and Linux presets (the message there was "The given export path doesn't exist").

23.7.3 What you get

The test export produced these files in `build/web/`:

File	What it is
`index.html`	The page that loads the game
`index.js`	Engine loader code
`index.wasm`	The engine itself (WebAssembly), about 39 MB here
`index.pck`	Your game data (only about 8 KB for Pip Dodge)
`index.audio.worklet.js`, `index.audio.position.worklet.js`	Audio helper scripts
`index.icon.png`, `index.apple-touch-icon.png`, `index.png`	Icon and loading image

All files must be kept together in the same folder.

23.7.4 Testing a Web export: do not double-click index.html

If you open `index.html` directly from your disk, the browser blocks the game from loading its own files, and you see an error or a blank page. A Web game must be **served** over HTTP.

Two easy ways:

1. **From the editor.** Godot can start a small local server and open your browser for you. Look for the Web run option in the editor's top-right area (the **Remote Debug** menu, which offers a run-in-browser entry when a Web preset exists). This is based on the documentation and was not clicked through during writing.
2. **From a terminal**, if you have Python installed. Open a terminal *outside* the export folder and give it the folder as an argument:

```
python -m http.server 8060 --directory path/to/build/web
```

Then open `http://localhost:8060/` in your browser. (On some systems the command is `python3`.) Press Ctrl+C in the terminal to stop it.

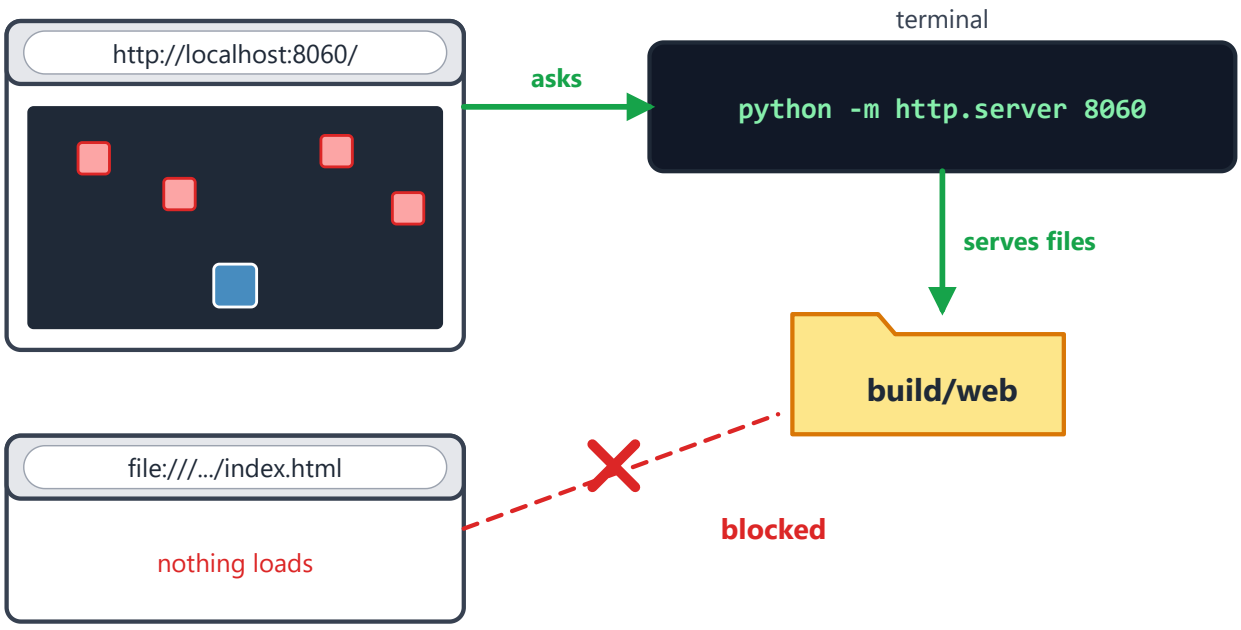


Figure 23.4: Serve the folder; do not open the file directly.

> **Why it works:** Browsers treat local files as unsafe and refuse certain operations on them. An HTTP server, even one on your own computer, makes the page look like a normal website, so the browser trusts it.

23.7.5 Web limits to remember

- **Quit** does nothing useful in a browser; hide the button (section 23.3.4).
- Audio only starts after the player clicks or presses a key once; browsers block sound before a user action. This is a browser rule, not a Godot bug.
- Saving with `user://` works: the browser stores it in its own storage. It can be cleared by the player.
- Web builds are slower to start (download of tens of MB) and may be slower in heavy games.
- File names are case-sensitive on the server (see Chapter 22).
- Starting windowed is best. The page, not your game, decides the size.

23.8 Command-line exports

The editor's Export button is fine for occasional use. When you export often, or want to automate it, use the **command line**. The same Godot program can export without opening the editor.

Run these from a terminal. Replace `godot` with the path to your Godot program (for example the `_console.exe` on Windows, which prints text to the terminal):

```
godot --headless --path path/to/project --export-release "Windows Desktop"
godot --headless --path path/to/project --export-release "Linux"
godot --headless --path path/to/project --export-release "Web"
```

What each part means:

Part	Meaning
`--headless`	Run with no window

Part	Meaning
`--path <folder>`	The project folder (the one with `project.godot`)
`--export-release "Name"`	Export with the preset of that name, as a release build
`--export-debug "Name"`	Same, but a debug build
`--export-pack "Name" file.pck`	Only create the `.pck` data file

The preset name must match what you typed in the Export window, including capital letters. The output path is taken from the preset's **Export Path** field, or you can give a path after the preset name, for example `--export-debug "Windows Desktop" build/windows/PipDodgeDebug.exe`.

Real output from the test of Pip Dodge (shortened):

```
Started Packing (102 steps)
Storing File: res://main.gdc
Storing File: res://main.tscn.remap
Storing File: res://project.binary
```

The word `gdc` shows that scripts are stored in a compiled form in release builds. On a failed export the last lines begin with `ERROR:` and say why, and the program exits with a non-zero exit code, which scripts can detect.

The other two forms were also tried. `--export-debug` created a debug `.exe` plus a `.console.exe`, and `--export-pack` created a 26 KB `.pck` file for Pip Dodge. A `.pck` on its own is useful for patches or for the "separate pck" layout.

23.9 Testing and publishing your build

An export is not finished until it has been tested **outside the editor**, ideally on a different computer. Many problems only show up there.

23.9.1 A test routine

1. Copy only the exported files (not the project) to a new folder.
2. Start the game from there, not from the editor.
3. Play through completely: start, play, lose, restart, quit.
4. Check that saving and loading work.
5. Try a small window size and a large one.
6. Ask somebody else to try it. Watch silently; their confusion is a gift.
7. If it fails, run the debug build with its console window and read the messages (Chapter 22 skills).

23.9.2 Common surprises

Surprise	Why
Works in the editor, black window in the export	No main scene set, or the scene failed to load
Image or sound missing in the export	The file name case differs, or the file was excluded by a filter
Game cannot save	The code writes to `res://`; use `user://`
Web build shows only a blank page	Opened from disk instead of an HTTP server
Web build has no sound	The browser needs a click first

Surprise	Why
Antivirus or SmartScreen warns	Unsigned program from a new publisher (normal)

23.9.3 Distribution

- Compress each build into a `.zip` for sharing. For Web, the `.zip` should contain `index.html` at its top level, not inside another folder.
- Name your files with a version, for example `PipDodge-1.0-windows.zip`.
- Keep the build folders out of Git (the `.gitignore` from Chapter 22 lists `build/`); share them as downloads instead.

Chapter 24 shows how to put your zip on itch.io.

Project 23: Export Pip Dodge

You will open a tiny game, play it, and export it to Windows, Linux and Web. The finished project is in `projects/ch23-export-demo/`.

Files

```
projects/ch23-export-demo/
  project.godot
  export_presets.cfg
  icon.svg
  main.tscn
  main.gd
  .gitignore
```

Scene tree (`main.tscn`)

```
Main (Node2D)            main.gd
  ScoreLabel (Label)
  InfoLabel (Label)
  MessageLabel (Label)   horizontal and vertical alignment centered
```

The game



Figure 23.5: Pip Dodge. This picture was taken from an editor run, so the footer says "editor run on Windows"; your exported build says "release export on Windows".

Pip slides left and right along the ground with the arrow keys. Red blocks fall; touching one ends the game. Space or Enter restarts. The blocks fall faster and spawn more often as the score (seconds survived) grows. Everything is drawn in `_draw()`, so no art files are needed.

The full script is in `projects/ch23-export-demo/main.gd`. Its important ideas, all from earlier chapters:

- `_process(delta)` moves Pip with `Input.get_axis("ui_left", "ui_right")` and `clampf` to keep Pip on screen.
- An array of `Vector2` positions stores the blocks, and a loop that counts **backwards** removes blocks that fell off the screen safely (the lesson from Chapter 22).
- `queue_redraw()` asks Godot to call `_draw()` again.
- `OS.has_feature` fills `InfoLabel` with the kind of build.

Steps

1. Open `projects/ch23-export-demo/` in the Project Manager and run it (F5). Play for a minute. The bottom label should say `editor run on ...`.
2. Open **Project > Project Settings...** and find the name, the main scene and the stretch settings from section 23.3.1. Confirm that each matches the file shown earlier.
3. Open **Editor > Manage Export Templates...** and install the templates for your version (section 23.2.1).
4. Open **Project > Export...**. The project already contains three presets (Windows Desktop, Linux, Web). If your list is empty, press **Add...** and create them.
5. In your file explorer, **create the folders** `build/windows`, `build/linux` and `build/web` inside the project folder.

6. Select **Windows Desktop**, press **Export Project...**, save as `build/windows/PipDodge.exe`, with **Export With Debug** off. Start the exe by double-clicking. The label should say `release export on Windows`.
7. Do the same for **Linux** (`build/linux/PipDodge.x86_64`). If you can reach a Linux computer, copy it over, run `chmod +x PipDodge.x86_64`, and start it.
8. Export **Web** to `build/web/index.html`. Serve it with the Python command in section 23.7.4 and open `http://localhost:8060/`. The label should say `release export on Web browser`. Click the game before expecting keyboard input.
9. Export once more for Windows **with** Export With Debug on, to a different file name, and compare the two files' sizes. Run the debug one and read its label.
10. Repeat the Windows export from the command line (section 23.8) and check that the result is identical in behavior.
11. Zip the Web folder so that `index.html` is at the top level of the zip. You will need this in Chapter 24.

Expected result

You have three builds of the same game. The Windows build runs by double-click, shows `release export on Windows`, and plays like the editor version. The Web build plays in your browser through a local server. The Linux build exists and, if tested, runs after `chmod +x`. No build folders are committed to Git.

Chapter summary

- Exporting combines a platform-specific export template (the engine) with your game's data (a `.pck` file).
- Export templates are a separate download that must match your Godot version exactly (Editor > Manage Export Templates...).
- Set the project name, icon, main scene and stretch settings before exporting.
- Presets live in the Export window (Project > Export...) and are stored in `export_presets.cfg`.
- The destination folder must exist before you export.
- Windows and Linux builds can be a single file with Embed PCK; Linux players may need `chmod +x`.
- Web builds need to be served over HTTP to test; thread support off is the simple default.
- Feature tags (`OS.has_feature`) let one project behave differently in editor, debug, release and Web builds.
- `--headless --export-release "Preset"` exports from the command line.
- Always test the export outside the editor, preferably on another computer.

Review questions

1. What two things does Godot combine to make an exported game?
2. True or false: Export templates are included in the normal Godot download.
3. Why must the version of the export templates match the version of the editor?
4. Multiple choice: In which file are your export presets stored? (a) project.godot (b) export_presets.cfg (c) icon.svg (d) main.tscn
5. You export to `build/web/index.html`, and Godot prints "Target folder does not exist or is inaccessible". What do you do?
6. Why does a Web export show a blank page when you double-click `index.html`, and what is the fix?
7. What does `OS.has_feature("web")` return in a browser build, and give one practical use.
8. True or false: Code that saves a score should write to `res://highscore.cfg` so that it is packed with the game.

9. Multiple choice: Which command-line option runs Godot without opening a window? (a) --quiet (b) --headless (c) --silent (d) --nowindow
10. Name two differences between a debug export and a release export.

Exercises

1. **Practice (easy):** Change `config/name` in the Project Settings to a title of your own and export Windows again. Where does the new name appear?
2. **Practice (easy):** Compare the sizes of the Windows build with Embed PCK on and off. List the files you get in each case.
3. **Apply (medium):** Replace `icon.svg` with a new square image of your own (draw it in any program, or edit the SVG). Re-export the Web build and check the icon shown on the page (`index.icon.png`).
4. **Apply (medium):** Add a Quit button to Pip Dodge's game-over message (a Button node) that calls `get_tree().quit()`, and hide it in the Web build using `OS.has_feature("web")`.
5. **Challenge (hard):** Write a small script (a `.bat` file on Windows or a `.sh` file elsewhere) that creates the build folders if they are missing and runs all three command-line exports in a row. Run it twice to prove it can be repeated.

Common mistakes

- **Symptom:** "No export template found" or "version mismatch". **Cause:** Templates are missing or from another Godot version. **Fix:** Install the templates that match the exact version (Editor > Manage Export Templates...).
- **Symptom:** Export fails immediately with a message about the path or target folder. **Cause:** The output folder does not exist. **Fix:** Create the folder, then export again.
- **Symptom:** The exported game opens and shows an empty gray window. **Cause:** No main scene is set, or the main scene path is wrong. **Fix:** Set **Project Settings > Application > Run > Main Scene**.
- **Symptom:** The Web game does not load or says it cannot fetch files. **Cause:** You opened `index.html` from your disk. **Fix:** Serve the folder over HTTP.
- **Symptom:** Saving works in the editor but fails in the build. **Cause:** The path starts with `res://`, which is read-only in an export. **Fix:** Use `user://`.
- **Symptom:** A picture shows in the editor but not on the Linux or Web build. **Cause:** The code asked for `Player.png`; the file is `player.png`. **Fix:** Match the case exactly and keep all names lowercase.

Further reading

- https://docs.godotengine.org/en/stable/tutorials/export/exporting_basics.html
- https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_windows.html
- https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_linuxbsd.html
- https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_web.html
- https://docs.godotengine.org/en/stable/tutorials/export/feature_tags.html
- https://docs.godotengine.org/en/stable/tutorials/editor/command_line_tutorial.html

Chapter 24: Where to Go Next

Learning objectives

By the end of this chapter you will be able to:

- Publish a browser game on itch.io with a clear page, screenshots and controls.
- Describe what a game jam is and plan a small game for a 48-hour deadline.
- Choose a learning path after this book (2D, 3D, shaders, tools, other languages).
- Use simple design thinking: core loop, scope, playtesting and iteration.
- Judge whether an idea is too big, and cut it down to something you can finish.
- Write a design document and plan a capstone game of your own.

Key terms

- **itch.io**: a website where independent creators publish games, including games that play in a browser.
- **Game jam**: an event where people make a game from scratch within a short time limit, usually around a theme.
- **Theme**: the word or idea a jam gives you to inspire the game.
- **Scope**: how much a project includes; "scope creep" is when it keeps growing.
- **Core loop**: the main thing a player does over and over (jump, collect, dodge).
- **Minimum viable game**: the smallest version of the game that is complete and playable.
- **Playtest**: watching someone else play your game to see what confuses them.
- **Iteration**: improving a design in repeated rounds of build, test, change.
- **Design document**: a short written plan for a game.
- **Game feel** ("juice"): the small touches (sound, motion, effects) that make actions satisfying.
- **Portfolio**: a collection of finished projects that shows what you can do.

24.1 You have finished more than you think

Take a moment to look back. You started this book without knowing what a node was. Since then you have:

- Built scenes out of nodes, and written scripts that control them.
- Made characters that move, collide, animate and make sound.
- Built levels with tilemaps and a platformer with enemies and pickups.
- Created menus, saved data, switched scenes and kept game state in autoloads.
- Made things in 3D, added particles and shaders, written simple enemy AI.
- Found bugs with the debugger, saved your work with Git, and exported games for other people to play.

That is a full foundation. The hardest step in game development is not a technique. It is **finishing**. Many people start thousands of games and finish none. This chapter is about the next step: finishing things, showing them to people, and choosing what to learn next.

24.2 Publishing on itch.io

itch.io is the most popular place for small independent games. It is free to join and free to publish, you control the price (including free), and it can host browser games that play right on the game's page. Because you exported a Web build in Chapter 23, you can have a playable page in under an hour.

> **Note:** Websites change their layout and wording from time to time. The steps below describe the general flow. If a button has moved or been renamed, look for the closest match. The principles stay the same.

24.2.1 Before you begin

Have these ready:

1. A tested **Web export** of your game (Chapter 23), zipped so that `index.html` sits at the **top level** of the zip. A zip whose first item is a folder will not work.
2. A **title** and a one-sentence description.
3. **Controls** written out in plain words.
4. A **cover image** and two to five **screenshots**. A cover image around 630 by 500 pixels is the commonly recommended size. Take screenshots of the game at its most interesting moments, not the title screen only.
5. A short list of the tools used, and credits for any art or sound that is not yours.

24.2.2 Create the page

1. Create an account at https://itch.io and sign in.
2. Open your dashboard and choose **Create new project**.
3. Fill in the **Title**. The **Project URL** is generated from it; you can change it.
4. For **Kind of project**, choose **HTML**. That option is for games played in a browser.
5. Under **Uploads**, upload your zip. Next to the file, tick **This file will be played in the browser**.
6. Set the **Viewport dimensions** to the size of your game window. Pip Dodge uses 640 by 360. A game that uses Godot's `canvas_items` stretch mode (Chapter 23) scales nicely inside this frame.
7. Look for options to enable a **Fullscreen button**, and tick it. Players enjoy it.
8. Write the **description** (see below), add the **cover image** and **screenshots**.
9. Choose a **Pricing** option. "No payments" is a good choice for a first project.
10. Pick **Genre** and **Tags** so people can find the game.
11. Set **Visibility** to **Draft** or **Restricted** while you test. Only after you have checked everything, make it **Public**.

Always open the page in a private window or on another device and play it as a stranger would. If the game will not load, the cause is nearly always the zip structure (a folder inside the zip) or a missing `index.html`.

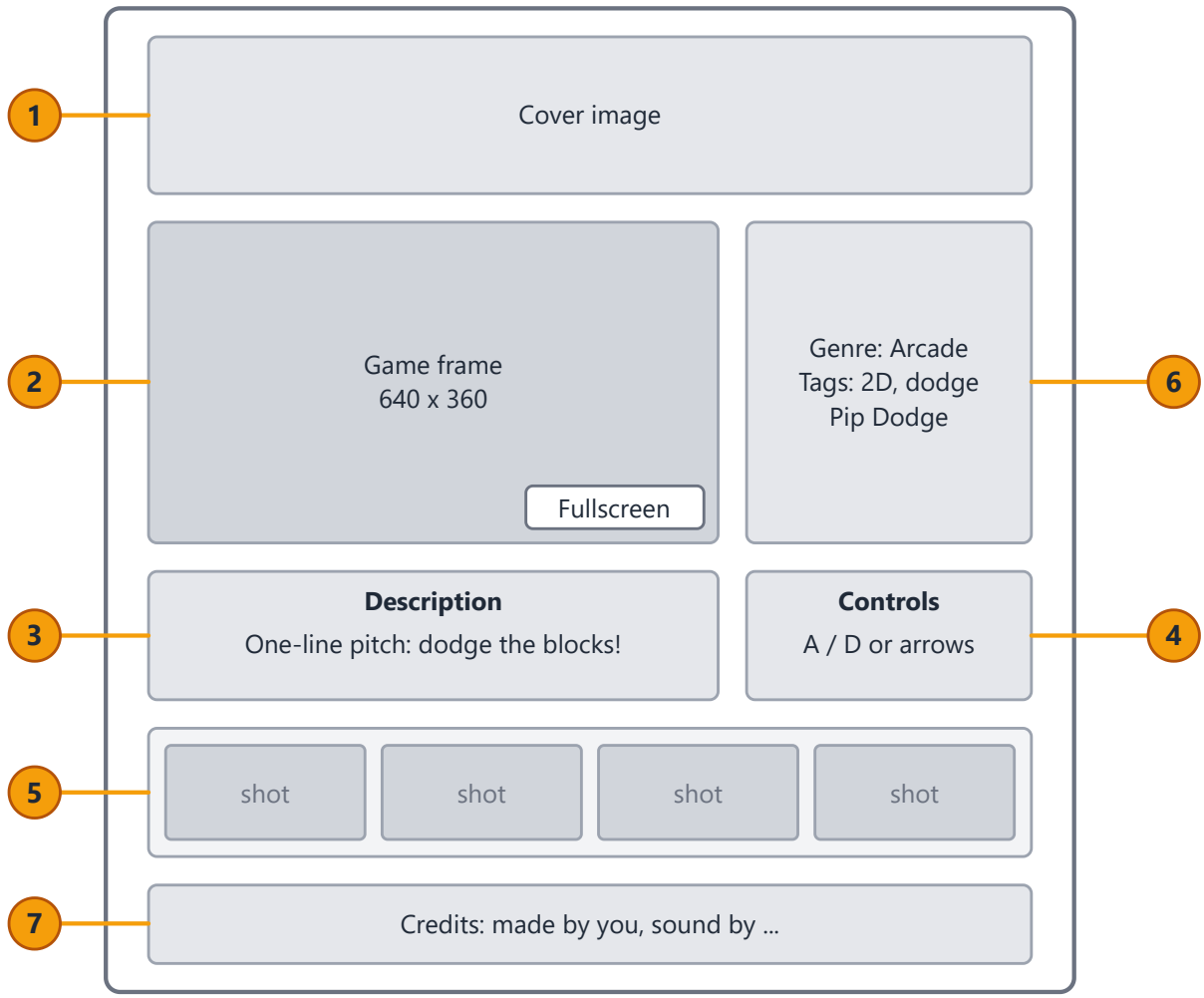


Figure 24.1: The parts of a good game page.

24.2.3 Writing the description

Players decide in seconds. A good description has this order:

1. **One line** that says what the game is: "Dodge falling blocks as long as you can."
2. **Controls** in a short list: "Left/Right arrows to move. Space to restart."
3. **A few details**: what makes it special, how long it takes, what is planned.
4. **Credits**: tools (Godot Engine), art, music, sounds, with links.
5. **An invitation**: "Tell me what you think in the comments."

Keep it short, honest and friendly. Never promise features that do not exist.

24.2.4 Threads and the Web build

If you turned on thread support in your Web export, the page must be served with special headers. itch.io has a setting for this (a **SharedArrayBuffer** option on the project's embed settings) that has to be enabled. Beginners should leave thread support off (Chapter 23), and then no such setting is needed.

24.2.5 Other ways to upload

You can also upload your Windows and Linux builds as extra downloadable files on the same page (zip each separately and name them clearly, for example `PipDodge-windows.zip`). itch.io also has a command-line uploader called **butler**, which is useful when you update often, because it uploads only what changed. It is optional; the web form is enough at first. Setting it up is described in itch.io's own documentation.

24.2.6 After publishing

- Share the link with friends and in communities you already belong to.
- Read comments, and fix the bugs people report. Say thanks.
- Do not take criticism of the game as criticism of you. A bug report is a gift.
- Update the page when you improve the game; mention what changed.

24.3 Game jams

A **game jam** is an event where participants make a complete game within a time limit, commonly 48 hours to ten days. A **theme** is announced at the start. The game is uploaded on a site, and other participants play and rate it.

Jams are one of the best learning tools there are, for these reasons:

- **A deadline forces you to finish.** Finishing is the skill that matters most.
- **A theme removes the blank page.** You no longer have to decide what to make.
- **Small scope is the rule.** Nobody expects a masterpiece.
- **Feedback comes quickly** from other people who understand what you attempted.
- **The community is friendly** to beginners.

24.3.1 Where to find jams

- **itch.io** lists jams of every size at https://itch.io/jams. You can filter by length and by whether they are rated or just for fun.
- Well-known recurring events include **Ludum Dare**, the **Global Game Jam**, and **GMTK Game Jam**. Check each event's own website for current dates and rules; they change from year to year.
- Godot's own community also runs jams from time to time. The community forum and Discord (links on the Godot website) announce them.

Rules differ from jam to jam: some require you to make all the art and sound during the event, some allow premade assets, some require the game to be browser-playable. **Read the rules first.**

24.3.2 A 48-hour plan

A plan keeps you calm. Here is a plan for a weekend jam. The times are a guide.

Time	Goal
Hour 0-2	Read the theme. Brainstorm ten ideas, pick the smallest. Write a one-sentence pitch.
Hour 2-4	Create the project, the main scene and the folder layout. Get one thing moving on the screen.
Hour 4-16	Build the core loop until it works with rectangles and circles. No art yet.
Hour 16-20	Sleep or rest. Really. A rested brain makes fewer bugs.

Time	Goal
Hour 20-32	Add win and lose conditions, restart, score, basic menu.
Hour 32-40	Sound, effects, simple art, polish ("juice", Chapter 20).
Hour 40-44	Export, test on another computer, fix the worst bugs.
Hour 44-47	Write the page, take screenshots, upload.
Hour 47-48	Submit. Buffer for the unexpected.

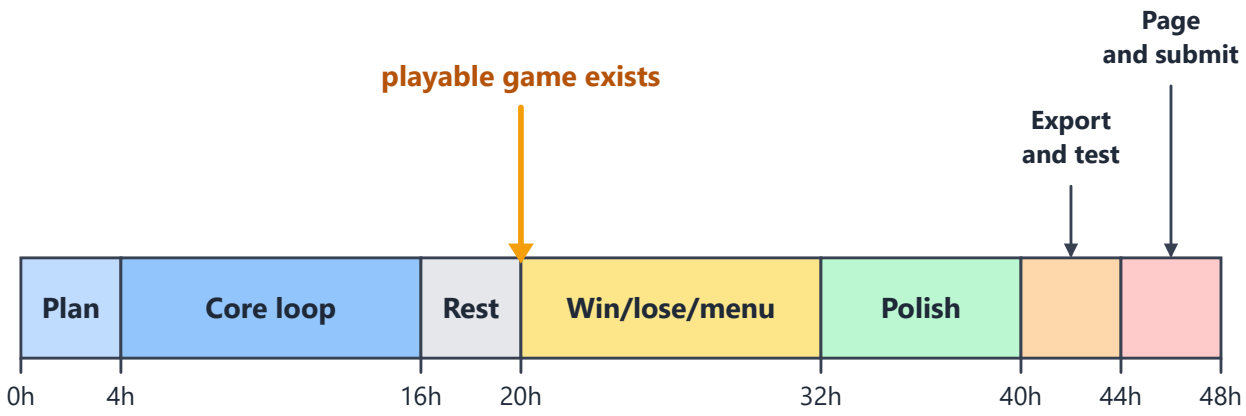


Figure 24.2: The 48-hour plan. A playable game exists by about the middle.

Look at the rule hidden in this table: **a working, complete game exists by the middle of the jam**, even if it is ugly. The rest of the time improves it. The opposite order (art first, gameplay later) is how jam games fail.

24.3.3 Jam survival rules

1. **Cut scope early.** Whatever you think you can finish, cut it in half.
2. **Make it playable first, pretty later.**
3. **Commit to Git often** (Chapter 22). A jam is a bad time to lose work.
4. **Test the export early**, not in the last hour. Do an export in the first few hours so you know the pipeline works.
5. **Use what you know.** A jam is not the time to learn a new system you have never touched.
6. **Eat, drink and sleep.** Tired people write bugs.
7. **Submit something.** A finished small game beats an unfinished big one every time.

24.4 Design thinking for small games

Programming makes a game *work*. Design makes it *worth playing*. You do not need a degree for this. You need a few simple tools of thought.

24.4.1 The core loop

Every good small game has a **core loop**: the cycle of actions the player repeats.

```
Action  ->  Result  ->  Reward or danger  ->  new Action
```

Examples from this book:

- Coin Dash: move -> reach coin -> score goes up and the next coin appears -> move.
- Pip Dodge: slide -> avoid a block -> survive longer, speed increases -> slide.
- Platformer: jump -> land or hit enemy -> collect pickup or lose a life -> jump.

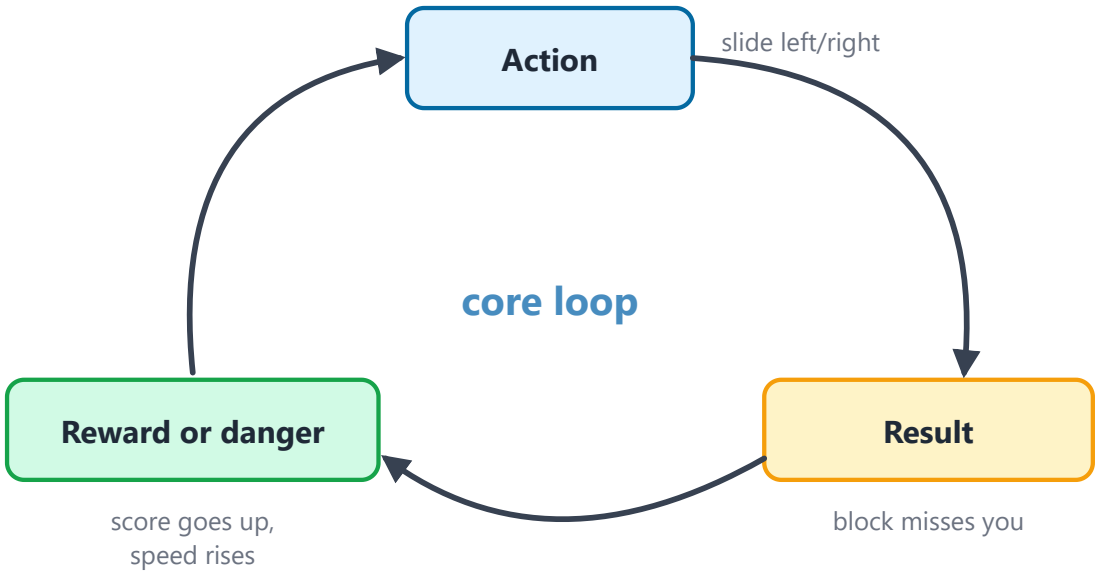


Figure 24.3: The core loop. If this circle is fun, the game is fun. If it is not, no amount of art will fix it.

Test your idea by asking: **what does the player do in the first ten seconds, and will they want to do it again?** If you cannot answer, the idea is not ready. Build the core loop first, with squares and circles, and play it. If a gray square dodging gray squares is fun, you have something. If it is boring, change the rules, not the paint.

24.4.2 One verb

A strong small game has one main verb: jump, dodge, push, collect, match, shoot. Games with five verbs need five times as much work and are rarely better. Start with one verb. Then ask: *how can this one verb be made interesting in more situations?* (Jumping onto platforms, jumping over enemies, jumping to reach coins.) Depth comes from variation, not from the amount of mechanics.

24.4.3 Rules, goals, feedback

Three questions help you check a design:

1. **What is the goal?** The player must know what they are trying to do.
2. **What are the rules?** What can they do, what can't they?
3. **What feedback do they get?** Does the game react to what they did, clearly and immediately (sound, color, motion, score)?

Missing feedback is the most common beginner problem. If a coin disappears silently, the player feels nothing. If it pops, jingles, and the score counter bounces, the player feels a reward. This is the "juice" of Chapter 20, and it is design, not decoration.

24.4.4 Difficulty and flow

If a game is too easy, players are bored. If it is too hard, they are frustrated. Between these lies a feeling often called **flow**, where the challenge matches the player's skill. Good games raise the challenge gradually: Pip Dodge starts slow and speeds up. Introduce one new idea at a time, in a safe place, then test it in harder places.

24.4.5 Scope: the art of cutting

Scope is the size of the job. The usual beginner mistake is to plan a game that would take a team a year. A useful method is to sort ideas into three groups:

- **Must have**: without these, it is not the game.
- **Nice to have**: you add them only if time remains.
- **Cut list**: ideas you write down so you can stop worrying about them. Writing an idea on the cut list is not losing it; it is parking it.

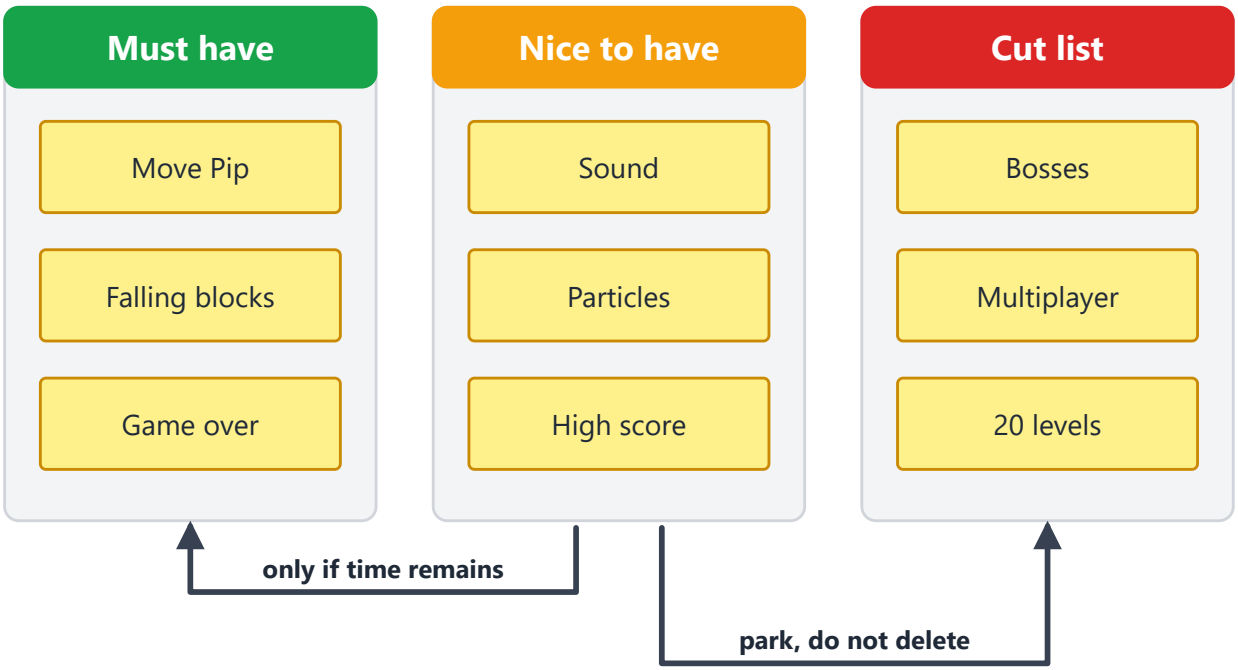


Figure 24.4: Sorting ideas. The cut list is a parking place, not a trash can.

A good size check: *"Could I describe the whole game in two sentences, and build a playable version in a weekend?"* If not, shrink it.

> **Why it works:** Time estimates are always too optimistic, because the unknown work (bugs, tools, confusing problems) does not show up in your plan. A common rule of thumb says to multiply your estimate by two or three. Smaller plans leave room for that.

24.4.6 Playtesting

You cannot see your own game with fresh eyes. You know every rule, so everything seems obvious. Other people do not.

How to playtest well:

1. Find one or two people who have not seen the game.
2. Give them the game and **say almost nothing**. Do not explain.
3. Watch their face and hands. Where do they hesitate? What do they try to press?
4. Write down what happened, not what you think it means.
5. Ask three questions afterwards: "What was the goal?" "What was frustrating?" "What would you change?"
6. Change one or two things, then test again.

If three of three people get stuck at the same place, that place is broken, whatever you meant. The game must explain itself.

24.4.7 Iteration

Games are not designed in one pass. They are *found* through repeated rounds of **build, play, change**. Your first version is a draft. Expect to throw away ideas. Keep each round small: a change you can test in ten minutes teaches you more than a change that takes a week. That is the same loop as the debugging loop of Chapter 22, and not by accident.

24.4.8 A design document

A **design document** is a short written plan. It does not have to be long or fancy. Its real purpose is to force you to decide, before you build, what the game is, and to give you something to check against when scope starts to creep. A template with these sections is in `projects/ch24-capstone/DESIGN-DOCUMENT-TEMPLATE.md`: pitch, player experience, core loop, rules, controls, content list, scene plan, scope, schedule, risks, playtest notes and a publishing checklist.

Write it in plain words. One page for a jam, two or three for a month-long project.

24.5 Learning paths: what to learn next

There is no single correct next step. The right one is the one that serves the game you want to make. Here are several paths. Pick one at a time.

24.5.1 Path A: Make more small games (recommended first)

Nothing beats finishing several small games. Make a new one every one to two weeks. Clone simple classics to practice: a Pong, a Snake, a Breakout, a Flappy-style game, a top-down shooter, a memory-card game. Cloning removes the design question and lets you concentrate on code. Then change one rule to make it your own.

24.5.2 Path B: Deepen 2D

- Learn more about `TileMapLayer`, terrain and parallax backgrounds (Chapter 12).
- Learn **animation trees** (`AnimationTree`) for characters with many states.
- Build a **top-down** action or adventure game with the navigation and AI of Chapter 21.
- Learn **dialogue** and **inventory** systems with custom resources (Chapter 16).
- Try **pixel-art** workflows with a free editor, then bring the art into Godot.

24.5.3 Path C: 3D

Chapter 19 was an introduction. Go further with a first-person or third-person controller, `CharacterBody3D`, level design with `GridMap`, lighting and environment, imported models (for example `.glb` files from free libraries), and animation of imported characters.

24.5.4 Path D: Visual effects and shaders

Learn more about the shading language, post-processing effects, particles and lighting (Chapter 20). A strong portfolio piece is a small scene that looks beautiful, without any game around it.

24.5.5 Path E: Tools and programming depth

- Read the full **GDScript reference** in the documentation, and practice static typing everywhere.
- Learn data structures: arrays, dictionaries, and when to use custom `Resource` classes.
- Write **editor tools** with `@tool` scripts and plugins.
- Learn to read other people's code: open-source Godot projects on GitHub are a great teacher.
- Other languages: Godot also supports **C#** (using the .NET version of the editor), and **GDExtension** for native code such as C++ or Rust. Both are advanced. Do not leave GDScript until you feel its limits.

24.5.6 Path F: Art, sound and music

A game is a mix of code, art, sound and writing. Many solo developers become stronger by learning a second craft: pixel art, 3D modelling in Blender (a free program), music with a free tracker or sequencer, or writing. Free asset sources for practice appear in Appendix F, among them Kenney.nl, itch.io asset packs and OpenGameArt. Always read each asset's licence and credit the author where required.

24.5.7 Path G: Multiplayer, mobile and other platforms

Godot has high-level multiplayer networking. It is a big topic; a good project is a simple two-player game over a local network. Mobile export (Android) needs extra setup; see the export documentation. Do these after you have finished several single-player games.

24.5.8 Where to learn

- **The official documentation** (https://docs.godotengine.org/en/stable/) is excellent: tutorials, class reference, and best-practice pages. Press **F1** inside the editor to open the built-in help, which is the class reference for your exact version.
- **The Godot community**: the official forum and chat servers linked from https://godotengine.org/. Ask questions with a short description, your code, and the exact error text. Say what you already tried.
- **Demo projects**: the official demo projects on GitHub show many features in small, readable projects.
- **Video channels and online courses** by independent teachers. Many Godot teachers exist and quality changes over time. Check that a tutorial says it is for **Godot 4**; Godot 3 tutorials use different names and will not work as written (for example `KinematicBody2D` and `yield`).

24.5.9 How to learn well

1. **Build, do not just watch.** Follow a tutorial once, then build something similar without it.
2. **Read errors.** The debugger is a teacher (Chapter 22).
3. **Keep a "done" list.** Seeing finished things beats feeling behind.
4. **Share early.** Post screenshots. People help those who show their work.
5. **Take rests.** Skills settle during breaks.
6. **Finish.** If you feel stuck on a project, cut it to a smaller game rather than abandoning it.

24.6 Building a portfolio and habits

A **portfolio** is a place that shows your finished work: your itch.io profile is a perfect start. Each finished game, however small, is proof that you can complete things. Add a short note to each page about what you learned or what you would do differently.

Habits that last:

- Pick a small regular time to build (even three hours a week) instead of waiting for large free days.
- Keep a project list file (such as `NOW.md`) with one thing in progress, a few next, and a place for ideas.
- Commit to Git every session.
- Finish a tiny game every month.
- Join a jam when you can.

Not every idea needs to become a game. Not every game needs to be shared. But some should, because finishing and sharing is where the most learning happens.

Project 24: The Capstone

This is the final project and it is yours. You design, build, export and publish **one small game of your own choosing**. The brief and the design document template are in `projects/ch24-capstone/`. No code is provided, because the code is yours to write.

Files

```
projects/ch24-capstone/
  CAPSTONE-BRIEF.md             what the capstone must contain and how it is judged
  DESIGN-DOCUMENT-TEMPLATE.md   the document to fill in before you build
```

Steps

1. **Choose a time box.** Pick either a four-week plan (a few hours per week) or a 48-hour jam-style weekend. Write the dates in your `NOW.md` file.
2. **Brainstorm ten ideas** in ten minutes. Write each as one sentence. Do not judge yet.
3. **Pick the smallest idea** that still excites you. Run the scope check from section 24.4.5: can you describe it in two sentences, and build a playable version in a weekend?
4. **Fill in the design document.** Copy `DESIGN-DOCUMENT-TEMPLATE.md` into your own project folder. Complete at least sections 1 to 8: pitch, player experience, core loop, rules, controls, content list, scene plan and scope. Fill in the cut list.
5. **Create the project and a Git repository.** Add a `.gitignore` with `.godot/` and `build/` (Chapter 22). Make your first commit.
6. **Build the core loop with shapes only.** Use `ColorRect`, `Polygon2D` or drawing in `_draw()` (Chapters 6 and 20). Play it. If it is not fun yet, change the rules before adding anything.
7. **Playtest early** with one person (section 24.4.6). Write down what happened. Change one or two things.
8. **Add the checklist items** from the brief, one at a time: win and lose, restart, title screen, pause or settings, saved value, sound.
9. **Polish** with the 'juice' tools from Chapter 20: small shakes, particles, tweens, sound. Keep this to a fixed fraction of the time.
10. **Export** (Chapter 23). Test the exported build outside the editor, on another computer if possible.
11. **Publish** on itch.io (section 24.2). Keep the page on Draft until you have played it as a stranger.

12. **Write the five-line "what I learned" note**: what went well, what was hard, what you cut, what you would do next, what surprised you.
13. Celebrate. You made a game.

Expected result

You have a finished, exported, published game with a title, an icon, a core loop that works, a way to win or lose, a way to restart, saved data, sound, and a page that explains it. You also have a filled-in design document and a Git history showing how the game grew.

Chapter summary

- Finishing is the hardest and most valuable game development skill; you now have the tools to finish.
- itch.io lets you publish a browser game: choose Kind of project HTML, upload a zip with `index.html` at the top, and tick that the file is played in the browser.
- A good page has a one-line pitch, controls, screenshots and credits.
- Game jams give you a deadline, a theme, and a community. Build a playable core loop first, art later.
- A core loop and one strong verb are the heart of a small game.
- Cut scope early: must have, nice to have, and a cut list.
- Playtest by watching in silence, then change one thing and test again.
- Choose a next learning path to match the game you want to make, and keep finishing small games.
- A design document is a short plan that you write before you build.

Review questions

1. What must be true about the structure of the zip file you upload for a browser game on itch.io?
2. Which "Kind of project" do you choose on itch.io for a game that plays in the browser, and what must you tick next to the uploaded file?
3. True or false: In a game jam, you should start with the art and add gameplay at the end.
4. Define "core loop" and give one example from a game in this book.
5. Multiple choice: Which is the best first step when your idea is too big? (a) Work longer hours (b) Cut it down to must-have items and a cut list (c) Add more features to make it special (d) Start a different game
6. Why should you say almost nothing during a playtest?
7. True or false: Godot 3 tutorials work unchanged in Godot 4.
8. Why is it a good idea to export and test your build early in a jam instead of at the end?
9. Name three things a good itch.io game description should include.
10. Give two reasons why a small finished game is more useful for learning than a large unfinished one.

Exercises

1. **Practice (easy):** Take one of your earlier projects (Coin Dash or Pip Dodge). Write a one-line pitch, a controls list and three screenshot ideas for an itch.io page.
2. **Practice (easy):** For three games you know, write the core loop in the form Action, Result, Reward or danger.
3. **Apply (medium):** Brainstorm ten game ideas for the theme "Reflection", then sort the best one into must have, nice to have and cut list.
4. **Apply (medium):** Publish Pip Dodge's Web build on itch.io as a Draft. Open it in a private browser window and play it. List any problems and fix them.

5. **Challenge (hard):** Join a real game jam (even a small one) and submit something, however simple. Afterwards write down which part of the 48-hour plan went differently than expected.

Common mistakes

- **Symptom:** The itch.io page shows "index.html not found" or nothing loads. **Cause:** The zip contains a folder that contains the files. **Fix:** Re-zip with `index.html` at the top level.
- **Symptom:** The embedded game is cut off or has scroll bars. **Cause:** Viewport dimensions on the page do not match the game window. **Fix:** Set the viewport width and height to match, and use `canvas_items` stretch.
- **Symptom:** The project never gets finished. **Cause:** The scope was too big or new ideas keep replacing old ones. **Fix:** Write ideas in a cut list, and finish a smaller version first.
- **Symptom:** Players do not understand what to do. **Cause:** The game only makes sense to its author. **Fix:** Playtest in silence and add clear feedback or a short instruction.
- **Symptom:** Feeling overwhelmed by the many paths. **Cause:** Trying to learn everything at once. **Fix:** Pick one path and one small project; the others will still be there.
- **Symptom:** Losing work in the last hours of a jam. **Cause:** No commits or backups. **Fix:** Commit often and push to an online repository.

Further reading

- https://docs.godotengine.org/en/stable/index.html
- https://docs.godotengine.org/en/stable/getting_started/introduction/index.html
- https://docs.godotengine.org/en/stable/community/index.html
- https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_web.html
- https://docs.godotengine.org/en/stable/tutorials/best_practices/index.html

Appendices

Appendix A: GDScript Quick Reference

This appendix is for looking things up. It is short on explanation on purpose. Each entry points to the chapter that teaches the idea properly. Everything here is Godot 4 GDScript. Snippets marked *fragment* are not complete scripts: they belong inside a function (usually `_ready()`) of a script that starts with `extends` something.

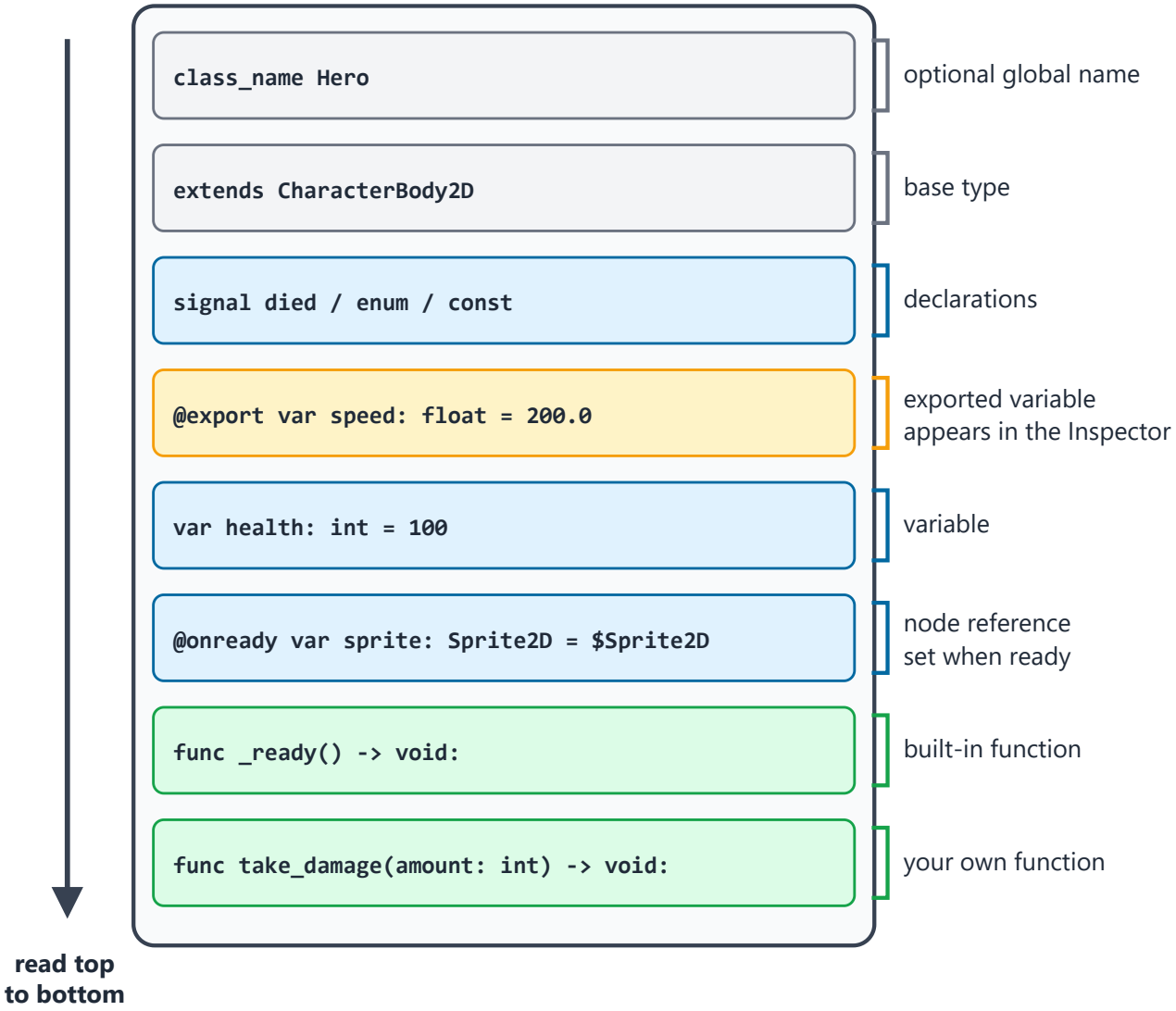


Figure A.1: The usual order of things in a script. Following this order keeps scripts easy to read.

A.1 The shape of a script

A script is a text file ending in `.gd`. It belongs to one node (or to one resource). The usual order is:

```gdscript
class_name Hero                     # optional: gives the script a global name
extends CharacterBody2D             # which node type this script extends

signal died                         # signals
enum State { IDLE, RUN }            # enums
const MAX_SPEED: float = 300.0      # constants

@export var speed: float = 200.0    # exported variables (show in the Inspector)
var health: int = 100               # ordinary variables
```

```gdscript
@onready var sprite: Sprite2D = $Sprite2D   # variables filled in when the node is ready

func _ready() -> void:              # built-in functions first
    pass

func take_damage(amount: int) -> void:   # then your own functions
    health -= amount
```

- Indentation (one Tab per level) shows which lines belong together.
- `#` starts a comment. The computer ignores the rest of the line.
- `pass` means "do nothing here". It is a placeholder for an empty block.
- A class name given with `class_name` must not be the name of a Godot class (do not call yours `Node`, `Timer`, `Player` is fine, `Camera2D` is not).

Taught in Chapters 3 and 5.

A.2 Variables and constants

```gdscript
var score: int = 0              # typed variable
var title := "Pip"              # type is inferred (String)
var anything = 5                # untyped variable: allowed, but avoid it
const GRAVITY: float = 980.0    # a constant never changes
```

- `var` creates a variable. `const` creates a constant.
- `:=` lets Godot work out the type from the value. It only works when Godot can tell the type from the right-hand side.
- A variable declared outside every function is a **member variable**. It belongs to the node and lives as long as the node does.
- A variable declared inside a function is **local**. It disappears when the function ends.
- Names use `snake_case` (lowercase, words joined by underscores). Constants use `UPPER_SNAKE_CASE`. Classes and enums use `PascalCase`.

Taught in Chapter 3.

A.3 Data types

Type	Holds	Example
`bool`	true or false	`var alive: bool = true`
`int`	whole number	`var lives: int = 3`
`float`	number with a decimal point	`var speed: float = 2.5`
`String`	text	`var name_text: String = "Pip"`
`StringName`	a fast, unique name (used for actions, animations)	`&"jump"`
`Vector2`	two numbers, x and y	`Vector2(10, 20)`
`Vector2i`	two whole numbers	`Vector2i(3, 4)`
`Vector3`	three numbers, x, y, z	`Vector3(0, 1, 0)`
`Color`	red, green, blue, alpha, each 0 to 1	`Color(1, 0, 0)`, `Color.RED`
`Rect2`	a rectangle (position and size)	`Rect2(0, 0, 64, 32)`
`Array`	an ordered list	`[1, 2, 3]`

Type	Holds	Example
`Dictionary`	a lookup table of key and value pairs	`{"hp": 10}`
`Callable`	a function you can pass around	`func(x): return x`
`Signal`	an event a node can announce	`died`
`NodePath`	the address of a node	`^"Player/Sprite2D"`
`Object` types	nodes and resources	`Node2D`, `Texture2D`
`Variant`	"any type", used when none is declared	`var x = 5`

Numbers: dividing two `int` values gives an `int` (whole-number division), so `7 / 2` is `3`. Make one side a float to get a decimal: `7 / 2.0` is `3.5`.

```
var a: int = 7 / 2      # 3
var b: float = 7 / 2.0  # 3.5
var r: int = 7 % 3      # 1, the remainder
var p: int = 2 ** 3     # 8, "two to the power of three"
```

Converting between types:

```
var s: String = str(42)         # "42"
var n: int = int("12")          # 12
var f: float = float("1.5")     # 1.5
var n2: int = "12".to_int()     # 12
```

Taught in Chapter 3.

A.4 Operators

Kind	Operators
Arithmetic	`+ - * / % **`
Assignment	`= += -= *= /= %=`
Comparison	`== != < > <= >=`
Logic	`and or not` (also written `&& || !`)
Membership	`in` (for example `"a" in my_array`)
Type test	`is` (for example `node is Node2D`)
Type cast	`as` (for example `node as Node2D`)

Use `==` to compare. A single `=` assigns. `if x = 5:` is an error.

Taught in Chapter 4.

A.5 Decisions

```
# fragment: inside a function
var x: int = 5
if x > 3 and x < 10:
    print("between")
elif x == 3:
    print("three")
```

```
else:
    print("other")

var msg: String = "yes" if x > 2 else "no"   # one-line if (ternary)
```

`match` compares one value against several patterns:

```
enum State { IDLE, WALK, JUMP }
var state: State = State.IDLE

func describe() -> void:
    match state:
        State.IDLE:
            print("idle")
        State.WALK, State.JUMP:
            print("moving")
        _:
            print("something else")   # _ means "anything not listed above"
```

Taught in Chapter 4.

A.6 Loops

```
# fragment: inside a function
for i in 5:                       # 0, 1, 2, 3, 4
    print(i)

for i in range(2, 10, 2):         # 2, 4, 6, 8 (start, stop before, step)
    print(i)

var items: Array[String] = ["a", "b"]
for item in items:                # each value in an array
    print(item)

var stats: Dictionary = {"hp": 10, "mp": 5}
for key in stats:                 # each key in a dictionary
    print(key, " = ", stats[key])

var n: int = 0
while n < 3:                      # repeat while the condition is true
    n += 1
    if n == 2:
        continue                  # skip to the next round
    if n > 5:
        break                     # leave the loop completely
```

`range(n)` counts from 0 up to `n - 1`. Never change the size of an array while looping over it. Collect the changes, then apply them afterward.

Taught in Chapter 4.

A.7 Functions

```
func add(a: int, b: int = 1) -> int:    # b has a default value
    return a + b

func greet(who: String) -> void:        # void: returns nothing
    print("Hello, ", who)

static func double(v: int) -> int:      # static: does not need an instance
    return v * 2
```

- Parameters go in the parentheses. `-> int` states what the function gives back.
- A call: `add(2, 3)` returns `5`; `add(2)` returns `3` because `b` defaults to 1.
- A function without `return` gives back `null`.
- **Lambdas** are small unnamed functions: `var dbl := func(v: int) -> int: return v * 2`, called with `dbl.call(4)`.

```
# fragment: inside a function
var doubled: Array = [1, 2, 3].map(func(v): return v * 2)              # [2, 4, 6]
var evens: Array = [1, 2, 3, 4].filter(func(v): return v % 2 == 0)      # [2, 4]
var total: int = [1, 2, 3].reduce(func(acc, v): return acc + v, 0)      # 6
```

Taught in Chapter 4.

A.8 Strings

```
# fragment: inside a function
var who: String = "Pip"
var coins: int = 7
print("Hello, %s! You have %d coins." % [who, coins])   # %s text, %d whole number, %f decimal
print("Hi {0}".format([who]))
print(who.length())                 # 3
print(who.to_upper())               # PIP
print(who.to_lower())               # pip
print(who.begins_with("Pi"))        # true
print(who.contains("i"))            # true
print(who.substr(1, 2))             # ip   (start, length)
print("a,b,c".split(","))           # ["a", "b", "c"]
```

Join strings with `+`, but only two strings: `"a" + str(5)`. Numbers must be converted first.

Taught in Chapter 3.

A.9 Arrays

```
# fragment: inside a function
var items: Array[String] = ["a", "b"]
items.append("c")             # add to the end (push_back does the same)
items.insert(0, "q")          # insert at a position
items.erase("a")              # remove the first matching value
items.remove_at(0)            # remove by position
print(items[0])               # read by position (counting starts at 0)
print(items.size())           # how many
print(items.has("b"))         # is it in there?
items.sort()                  # sort in place
items.shuffle()               # random order
var pick: String = items.pick_random()   # a random element (the array must not be empty)
print(items.is_empty())
items.clear()
```

Negative positions count from the end: `items[-1]` is the last element. Reading past the end is an error.

Taught in Chapter 4.

A.10 Dictionaries

```
# fragment: inside a function
var stats: Dictionary = {"hp": 10, "mp": 5}
```

```
stats["gold"] = 3                    # add or change
print(stats["hp"])                   # read (error if the key is missing)
print(stats.get("xp", 0))            # read with a default (safe)
print(stats.has("mp"))               # true
stats.erase("mp")                    # remove
print(stats.keys(), stats.values(), stats.size())
```

Taught in Chapter 4.

A.11 Classes, enums, and exports

```
enum State { IDLE, WALK, JUMP }          # named whole numbers
enum { RED, GREEN }                      # anonymous enum: constants RED, GREEN

@export var speed: float = 200.0
@export_range(0, 100, 1) var health: int = 100
@export var tint: Color = Color.WHITE
@export var enemy_scene: PackedScene     # drag a scene here in the Inspector
@export_group("Combat")                  # groups the next exports in the Inspector
@export var damage: int = 5
@export_multiline var notes: String = ""
```

An inner class lives inside a script:

```
class Inner:
    var v: int = 1
    func _init(start: int = 1) -> void:     # _init runs when .new() is called
        v = start
```

Create one with `Inner.new(3)`.

A property with a setter and getter:

```
var _backing: float = 1.0
var power: float:
    get:
        return _backing
    set(value):
        _backing = maxf(value, 0.0)
```

Taught in Chapters 5 and 16.

A.12 Built-in functions that Godot calls for you

These are called automatically at the right moment if you define them. Names start with an underscore.

Function	When it is called
`_init()`	When the object is created, before it is in the scene
`_enter_tree()`	When the node enters the scene tree (before its children)
`_ready()`	Once, after the node and all its children are ready
`_process(delta)`	Every frame; `delta` is the seconds since the last frame
`_physics_process(delta)`	At a fixed rate (60 times a second by default); use for movement and physics
`_input(event)`	For every input event

Function	When it is called
`_unhandled_input(event)`	For input events nobody else used (best for game controls)
`_draw()`	When the node must draw itself (call `queue_redraw()` to ask for it)
`_exit_tree()`	When the node is leaving the scene tree

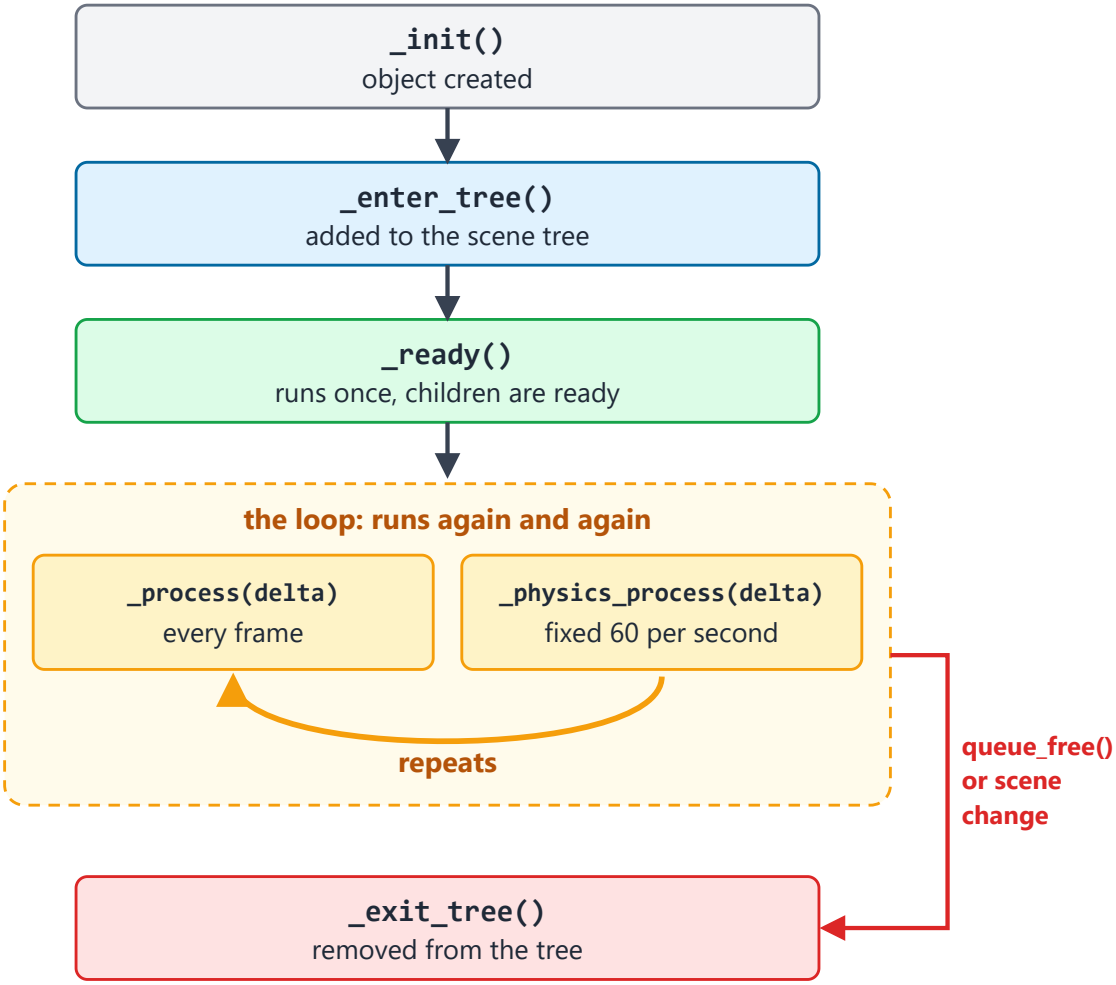


Figure A.2: The life of a node. `_init` first, `_ready` once, then `_process` and `_physics_process` repeating until the node is removed.

```gdscript
func _process(delta: float) -> void:
    position.x += 100.0 * delta        # 100 pixels per second, at any frame rate

func _unhandled_input(event: InputEvent) -> void:
    if event.is_action_pressed("jump"):
        print("jump!")
```

Taught in Chapters 5 and 7.

A.13 Talking to nodes

```gdscript
# script on a node that has a child named Label
@onready var label: Label = $Label          # shorthand for get_node("Label")

func _ready() -> void:
```

```
    var same: Node = get_node("Label")
    var maybe: Node = get_node_or_null("Nope")   # null instead of an error
    var parent: Node = get_parent()
    for child in get_children():
        print(child.name)
```

- `$Name` finds a child by name. `$Path/To/Node` goes deeper. `%Name` finds a node marked *Access as Unique Name* anywhere in the same scene.
- `@onready` waits until `_ready()` time to fill the variable, because children do not exist before then.
- Properties are set with a dot: `$Label.text = "Hi"`.
- Methods are called with a dot: `$Sprite2D.hide()`.

Taught in Chapter 5.

A.14 Groups

```
# fragment: inside a function
add_to_group("enemies")
if is_in_group("enemies"):
    print("I am an enemy")
var all_enemies: Array[Node] = get_tree().get_nodes_in_group("enemies")
get_tree().call_group("enemies", "queue_free")   # call a method on every member
```

Taught in Chapter 5.

A.15 Signals

```
signal health_changed(new_value: int)     # declare (top of the script)
signal died

func hurt(amount: int) -> void:
    health_changed.emit(10)                # announce
    died.emit()

func _ready() -> void:
    health_changed.connect(_on_health_changed)            # listen with a named function
    died.connect(func(): print("died"))                    # or with a lambda

func _on_health_changed(new_value: int) -> void:
    print("health is now ", new_value)
```

Connecting another node's signal looks the same: `$Button.pressed.connect(_on_button_pressed)`. Common built-in signals: `pressed` (Button), `body_entered` and `area_entered` (Area2D), `timeout` (Timer), `finished` (AudioStreamPlayer, Tween, AnimationPlayer's `animation_finished`).

Taught in Chapters 5 and 15.

A.16 Waiting with await

```
func _ready() -> void:
    await get_tree().create_timer(1.0).timeout    # pause this function for 1 second
    print("one second later")
```

`await` pauses only the function it is in. The rest of the game keeps running. You can await any signal: `await $Button.pressed`.

Taught in Chapters 15 and 21.

A.17 Input

```gdscript
func _physics_process(delta: float) -> void:
    var dir: Vector2 = Input.get_vector("left", "right", "up", "down")   # arrows or stick
    var h: float = Input.get_axis("left", "right")                        # -1, 0, or 1
    if Input.is_action_pressed("left"):          # held right now
        pass
    if Input.is_action_just_pressed("jump"):     # pressed this frame
        pass
    if Input.is_action_just_released("jump"):    # released this frame
        pass
```

Action names (`"left"`, `"jump"`) are the names you create in *Project* > *Project Settings* > *Input Map*. The built-in actions `ui_left`, `ui_right`, `ui_up`, `ui_down`, `ui_accept`, `ui_cancel` already exist.

Taught in Chapter 7.

A.18 Moving and physics

```gdscript
extends CharacterBody2D

func _physics_process(delta: float) -> void:
    var dir: Vector2 = Input.get_vector("left", "right", "up", "down")
    velocity = dir * 200.0
    if not is_on_floor():
        velocity.y += 980.0 * delta          # gravity
    if Input.is_action_just_pressed("jump") and is_on_floor():
        velocity.y = -400.0                  # up is negative y in 2D
    move_and_slide()
```

Useful `CharacterBody2D` helpers: `is_on_floor()`, `is_on_wall()`, `is_on_ceiling()`. `move_and_slide()` uses the `velocity` property and handles collisions for you.

For a node without physics, change `position` or `global_position` directly. `look_at(point)` turns a `Node2D` to face a point.

Taught in Chapters 7 and 8.

A.19 Vectors, in brief

```gdscript
# fragment: inside a function
var v: Vector2 = Vector2(3, 4)
print(v.length())                       # 5
print(v.normalized())                   # length 1, same direction
print(v.dot(Vector2.RIGHT))             # 3
print(v.angle())                        # radians from the +x axis
print(v.distance_to(Vector2.ZERO))      # 5
print(v.direction_to(Vector2.ZERO))     # unit vector pointing at the target
print(v.rotated(PI / 2))
print(v.lerp(Vector2.ZERO, 0.5))
print(v.move_toward(Vector2.ZERO, 1.0))
print(v.limit_length(2.0))
```

Built-in directions: `Vector2.UP` (0, -1), `DOWN` (0, 1), `LEFT` (-1, 0), `RIGHT` (1, 0), `ZERO`, `ONE`. Appendix G explains the ideas.

A.20 Math helpers

Function	Does
`abs(x)`, `absf(x)`	Absolute value
`sign(x)`	-1, 0, or 1
`min(a, b)`, `max(a, b)`	Smaller or larger (`minf`, `maxf` for floats)
`clamp(x, lo, hi)`	Keep x between lo and hi (`clampf`, `clampi`)
`floor(x)`, `ceil(x)`, `round(x)`	Round down, up, nearest
`snappedf(x, step)`	Round to a multiple of step
`lerp(a, b, t)`, `lerpf`	Blend from a to b by t (0 to 1)
`remap(v, a, b, c, d)`	Convert v from range a..b to range c..d
`sqrt(x)`, `pow(x, y)`	Square root, power
`sin(x)`, `cos(x)`	Trigonometry, angles in radians
`deg_to_rad(d)`, `rad_to_deg(r)`	Convert angles
`randi()`, `randf()`	Random int, random float from 0 to 1
`randi_range(a, b)`, `randf_range(a, b)`	Random number in a range (both ends included for ints)
`randomize()`	Seed the random generator from the clock (Godot 4 already does this on start)

Constants: `PI` (half a turn in radians), `TAU` (a full turn, 2 times PI).

A.21 Scenes and the tree

```
@export var coin_scene: PackedScene

func spawn_coin() -> void:
    var coin: Node2D = coin_scene.instantiate()      # make a copy of the scene
    add_child(coin)                                   # put it in the tree
    coin.position = Vector2(100, 100)

func _ready() -> void:
    var packed: PackedScene = load("res://main.tscn")      # loaded when this line runs
    var fast: PackedScene = preload("res://main.tscn")     # loaded when the script loads
    get_tree().change_scene_to_file("res://main.tscn")     # switch scenes
    get_tree().reload_current_scene()
    get_tree().paused = true                               # pause the game
    get_tree().quit()                                      # close the game
```

- `queue_free()` removes a node safely at the end of the frame.
- `res://` is your project folder. `user://` is a writable per-user folder for save files.

Taught in Chapters 2, 9, and 15.

A.22 Timers and tweens

```
func _ready() -> void:
    var tw: Tween = create_tween()
    tw.tween_property(self, "position", Vector2(100, 0), 1.0)   # move to x=100 over 1 second
```

For repeating timers, add a `Timer` node, set `wait_time`, and connect its `timeout` signal. For one-off delays, use `await get_tree().create_timer(seconds).timeout`.

Taught in Chapters 10 and 21.

A.23 Drawing with code

```
extends Node2D

func _draw() -> void:
    draw_circle(Vector2.ZERO, 20.0, Color.RED)
    draw_line(Vector2.ZERO, Vector2(50, 50), Color.WHITE, 2.0)
    draw_rect(Rect2(0, 0, 10, 10), Color.BLUE)
```

`_draw()` runs once and the result is kept. Call `queue_redraw()` whenever what you draw has changed.

Taught in Chapter 6.

A.24 Saving and loading

```
func save_json() -> void:
    var data: Dictionary = {"high_score": 120}
    var f: FileAccess = FileAccess.open("user://save.json", FileAccess.WRITE)
    f.store_string(JSON.stringify(data))
    f.close()

func load_json() -> void:
    if not FileAccess.file_exists("user://save.json"):
        return
    var f: FileAccess = FileAccess.open("user://save.json", FileAccess.READ)
    var parsed: Variant = JSON.parse_string(f.get_as_text())
    print(parsed)

func save_config() -> void:
    var cfg: ConfigFile = ConfigFile.new()
    cfg.set_value("game", "high_score", 120)
    cfg.save("user://settings.cfg")

func load_config() -> void:
    var cfg: ConfigFile = ConfigFile.new()
    if cfg.load("user://settings.cfg") == OK:
        print(cfg.get_value("game", "high_score", 0))
```

JSON turns whole numbers into floats when read back, so convert with `int(...)` when you need an integer.

Taught in Chapter 17.

A.25 Printing and debugging

```
func _ready() -> void:
    print("plain text")
    print("two ", "values")
    print_rich("[b]bold[/b] text in the Output panel")
    push_warning("yellow warning in the debugger")
    push_error("red error in the debugger")
    assert(1 > 0, "this message appears only if the test is false")
```

Taught in Chapters 3 and 22.

A.26 Godot 3 to Godot 4 translation table

When an old tutorial does not work, look here.

Godot 3	Godot 4
`onready var x`	`@onready var x`
`export var x`	`@export var x`
`KinematicBody2D`	`CharacterBody2D`
`Sprite`	`Sprite2D`
`yield(obj, "signal")`	`await obj.signal_name`
`connect("sig", self, "fn")`	`sig.connect(fn)`
`emit_signal("sig")`	`sig.emit()`
`move_and_slide(velocity)`	set `velocity`, then `move_and_slide()`
`instance()`	`instantiate()`
`TileMap` with one layer	`TileMapLayer`
`OS.get_ticks_msec()`	`Time.get_ticks_msec()`
`rand_range(a, b)`	`randf_range(a, b)`
`deg2rad`, `rad2deg`	`deg_to_rad`, `rad_to_deg`
`str2var`, `parse_json`	`str_to_var`, `JSON.parse_string`

Appendix B: Editor Shortcuts

Keyboard shortcuts save time, but you do not need to memorize them. Learn the five or six you use every hour and look up the rest here.

This appendix lists the default shortcuts of the Godot 4 editor on **Windows** (they are the same on Linux unless noted). Only well-established defaults are listed. If a shortcut does not work for you, it may have been changed or may belong to a newer or older version. Godot lets you see and change every shortcut in *Editor* > *Editor Settings* > *Shortcuts*, and that list is the final authority for your installed version.

B.1 Windows to macOS

Windows / Linux	macOS
Ctrl	**Cmd**
Alt	**Option**
Shift	**Shift**
Right mouse button	Right click, or Control+click
Middle mouse button	Middle mouse button (a mouse with a wheel button is recommended)

In nearly every case the macOS shortcut is the Windows one with **Cmd** where it says **Ctrl** (for example **Ctrl+S** becomes **Cmd+S**). The run and stop keys are the main exceptions, shown below. If a macOS shortcut does not behave as listed, check *Editor* > *Editor Settings* > *Shortcuts*.

Action	Windows / Linux	macOS
Run project	F5	Cmd+B
Run current scene	F6	Cmd+R
Stop	F8	Cmd+.

If you are on a Mac with function keys set to media controls, you may need to hold **Fn** with an F-key.

B.2 Scenes and files

Action	Shortcut
New scene	Ctrl+N
Open scene	Ctrl+O
Save scene	Ctrl+S
Save scene as	Ctrl+Shift+S
Save all scenes	Ctrl+Shift+Alt+S
Reopen the last closed scene	Ctrl+Shift+T
Quick open scene	Ctrl+Shift+O
Quick open script	Ctrl+Alt+O
Close the current scene tab	Ctrl+W
Undo	Ctrl+Z

Action	Shortcut
Redo	Ctrl+Shift+Z (Ctrl+Y also works on Windows)

> **Warning:** Godot does not save automatically. Press **Ctrl+S** often. When you press Run, Godot saves scenes first, but changes in an open script may be unsaved if you edited it elsewhere.

B.3 Running the game

Action	Shortcut
Run project (Play)	F5
Run current scene	F6
Pause running game	F7
Stop running game	F8

B.4 Switching workspaces

Action	Shortcut
2D workspace	Ctrl+F1
3D workspace	Ctrl+F2
Script workspace	Ctrl+F3
Asset Store workspace	Ctrl+F4

You can always click the workspace names in the middle of the top bar.

B.5 The Scene dock (the node tree)

Action	Shortcut
Add a child node	Ctrl+A
Instantiate a child scene	Ctrl+Shift+A
Rename the selected node	F2
Duplicate the selected node	Ctrl+D
Copy / Cut / Paste node	Ctrl+C / Ctrl+X / Ctrl+V
Delete the selected node	Delete
Multi-select nodes	Click, then Shift+click or Ctrl+click others

B.6 The 2D viewport

Action	Shortcut
Select mode	Q
Move mode	W

Action	Shortcut
Rotate mode	E
Scale mode	R
Pan the view	Middle mouse drag, or hold Space and drag
Zoom	Mouse wheel
Zoom to fit the selection	F (focus on selected)

B.7 The 3D viewport

Action	Shortcut
Select / Move / Rotate / Scale	Q / W / E / R
Orbit the view	Middle mouse drag
Pan the view	Shift + middle mouse drag
Zoom	Mouse wheel
Fly around (free look)	Hold the right mouse button, then use W, A, S, D (Q down, E up)
Focus on the selected object	F

B.8 The script editor

Action	Shortcut
Find in the current file	Ctrl+F
Find next / previous	F3 / Shift+F3
Replace	Ctrl+R
Find in all files	Ctrl+Shift+F
Go to line	Ctrl+L
Comment or uncomment the selected lines	Ctrl+K
Show code completion	Ctrl+Space
Duplicate selection or line	Ctrl+D
Delete the current line	Ctrl+Shift+K
Move the line up / down	Alt+Up / Alt+Down
Indent / unindent the selection	Tab / Shift+Tab
Go to the definition of a name	Ctrl+click on the name
Open the help page for a name	Ctrl+click on a built-in class name, or press F1 to search help
Select all	Ctrl+A

B.9 The debugger

Action	Shortcut
Toggle a breakpoint on the current line	F9

Action	Shortcut
Step over (run the current line)	F10
Step into (go inside a function call)	F11
Continue running	F12

> **Note:** Breakpoint shortcuts can differ slightly between versions. If one seems wrong, hover the buttons in the Debugger panel: the tooltip shows the current shortcut.

B.10 The editor itself

Action	Shortcut
Open the Command Palette (search any editor command)	Ctrl+Shift+P
Search help	F1
Toggle fullscreen	Shift+F11
Distraction-free mode (hide docks)	Ctrl+Shift+F11
Open Project Settings	no shortcut: use *Project* > *Project Settings*
Open Editor Settings	no shortcut: use *Editor* > *Editor Settings*

B.11 Mouse tips that are not shortcuts

- **Drag a file from the FileSystem dock** into the Inspector's empty resource slot to assign it, or into the Scene dock to add it as a node.
- **Drag a node from the Scene dock into a script** to paste its path (a `$Name` reference in recent versions).
- **Right-click a node** in the Scene dock for a menu: *Add Child Node...*, *Instantiate Child Scene*, *Attach Script*, *Save Branch as Scene*, and more.

> **Try it:** Pick three shortcuts from this appendix and put them on a sticky note beside your screen: save (**Ctrl+S**), run (**F5**), and add node (**Ctrl+A**). Use only those for a week. Add more when they feel automatic.

Appendix C: Troubleshooting Guide

When something goes wrong, you are in good company. Every programmer spends a large part of every day fixing things that do not work. This appendix lists the problems beginners meet most often, grouped by area. Each entry has the same three parts:

- **Symptom:** what you see.
- **Cause:** why it happens.
- **Fix:** what to do.

Entries are numbered so you can refer to them (for example "C.17"). Use the search function of your reader or the headings below to find yours.

A calm routine to try first

Before you look anything up, do these four things in order. They solve most problems.

1. **Read the message.** Look at the **Output** panel and the **Debugger > Errors** tab at the bottom of the editor. Red text tells you the file, the line number, and what went wrong. Click the line number to jump to it.
2. **Read the first error, not the last.** One mistake often causes a pile of later errors. Fix the top one, run again.
3. **Check the line above.** A missing bracket or colon is often reported on the *next* line.
4. **Save, then run again.** Make sure the script and scene are saved (**Ctrl+S**).

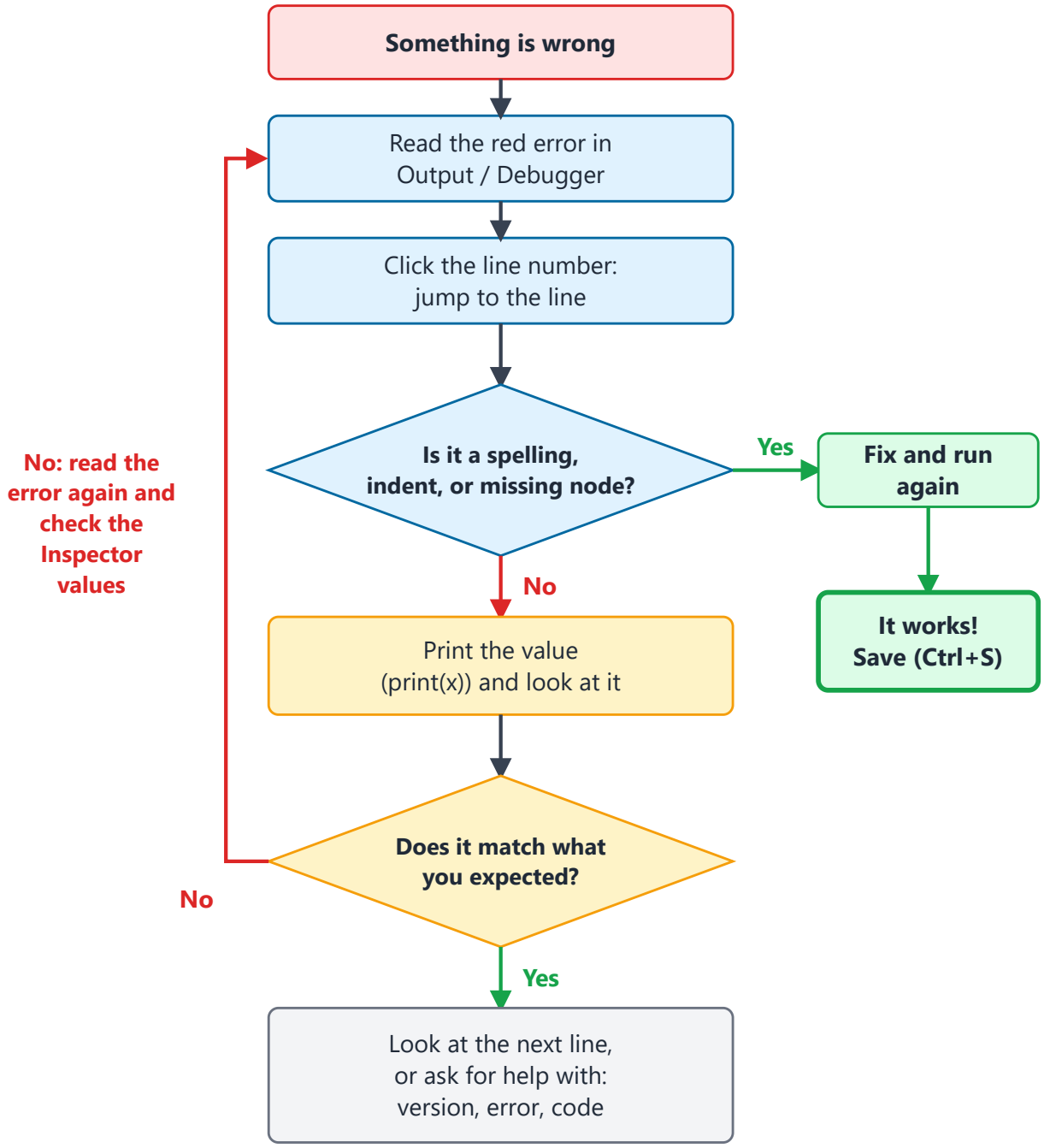


Figure C.1: A simple loop for any problem. Most bugs are found at the "print the value" step.

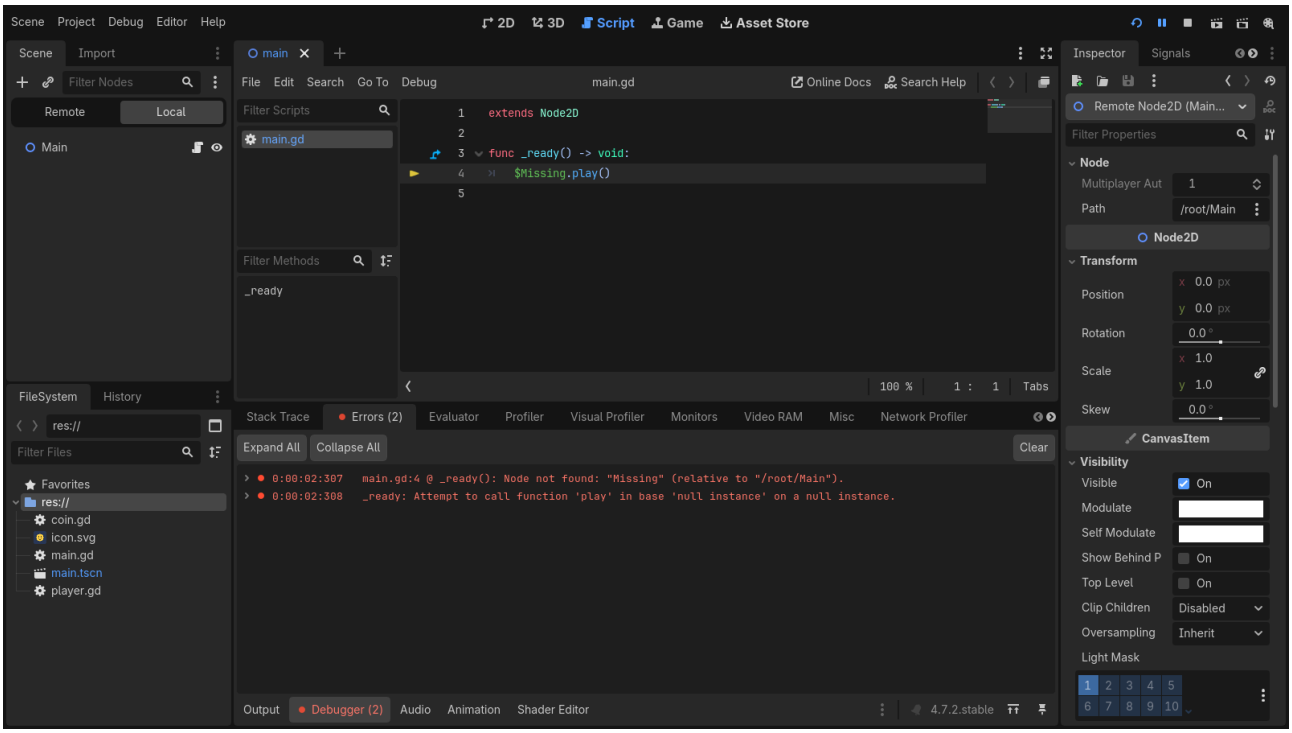


Figure C.2: Where Godot shows errors. The message names the script file and the line number.

Reading an error message: a typical one looks like

```
Attempt to call function 'play' in base 'null instance' on a null instance.
   at: _ready (res://player.gd:8)
```

This says: in `player.gd` at line 8 inside `_ready`, the script tried to use `play` on something that is `Nil` (nothing). See C.12.

Editor and project problems

C.1 The editor is a black or blank window, or it crashes at startup

- **Cause:** The graphics driver does not fully support the default renderer (Vulkan).
- **Fix:** In the Project Manager, create or edit the project and choose the **Compatibility** renderer (it uses OpenGL and works on older hardware). You can also update your graphics driver. In an existing project, change *Project Settings* > *Rendering* > *Renderer*, or start Godot with the command-line option `--rendering-driver opengl3`.

C.2 A project does not open: "unsupported" or "created with a newer version"

- **Cause:** The project was saved with a newer Godot than the one you are running, or it is a Godot 3 project.
- **Fix:** Install the newer Godot version, or open the project with the matching version. A Godot 3 project must be converted; for a beginner, it is usually easier to rebuild it in Godot 4 using the translation table in Appendix A.

C.3 "Scene failed to load" or "Missing dependencies"

- **Cause:** A file the scene needs (texture, script, sub-scene) was moved, renamed, or deleted outside Godot.

- **Fix:** In the dialog, use *Fix Dependencies* to point at the new location. In the future, move and rename files inside the **FileSystem dock**, not in Windows Explorer or Finder, so Godot can update the references.

C.4 My changes to a script do not appear when I run the game

- **Cause:** The script is not saved, or you are editing a different copy of the file, or the scene running is not the one you are editing.
- **Fix:** Save (**Ctrl+S**). Check the main scene in *Project Settings* > *Application* > *Run* > *Main Scene*. Use **F6** to run the scene you have open and **F5** to run the main scene.

C.5 Pressing F5 asks me to select a main scene

- **Cause:** No main scene is set.
- **Fix:** Click *Select Current* (if the right scene is open), or choose it. You can change it any time in *Project Settings* > *Application* > *Run* > *Main Scene*.

C.6 An exported variable (`@export`) does not show in the Inspector

- **Cause:** The script has not been saved, or has an error, or the node is not selected again after the change, or the type cannot be shown.
- **Fix:** Fix any red error in the script, save, then click another node and click back. Give the variable a type: `@export var speed: float = 200.0`.

C.7 I changed a value in the Inspector but the code value is used

- **Cause:** The value in a **saved scene** overrides the default in the script. The script's number is only the default.
- **Fix:** Change the value in the Inspector (or click the circular arrow beside the property to revert it to the script's default).

C.8 The scene tree is a mess and I cannot find a node

- **Cause:** Many nodes with similar names.
- **Fix:** Use the **Filter** box at the top of the Scene dock. Give nodes meaningful names (press **F2**). Right-click an important node and choose *Access as Unique Name* so that `%Name` always finds it.

C.9 The Output panel or the bottom panels have vanished

- **Cause:** A dock was closed or you are in distraction-free mode.
- **Fix:** Use the panel names along the bottom edge, or reset with *Editor* > *Editor Layout* (pick a saved layout, or the default). Distraction-free mode toggles with **Ctrl+Shift+F11**.

C.10 Godot says "Resource file not found: res://..." when I run

- **Cause:** The path to a file is wrong (spelling, folder, capital letters), or the file was moved.
- **Fix:** Right-click the file in the FileSystem dock and choose *Copy Path*, then paste it. Paths are case-sensitive in exported games and on Linux, even if Windows accepts the wrong case.

Script problems

C.11 "Expected indented block" or "Mixed use of tabs and spaces for indentation"

- **Cause:** Lines inside a function, `if`, or loop are not indented, or tabs and spaces are mixed.
- **Fix:** Indent the body of every line that ends in a colon with one Tab. Use only tabs. Select all the code and use the editor's convert-indentation command (in the *Edit* menu of the script editor) if a paste brought in spaces.

C.12 "Attempt to call function 'x' in base 'null instance' on a null instance" (older versions: "Invalid call. Nonexistent function 'x' in base 'Nil'")

- **Cause:** A variable that you thought held a node holds nothing (`null`). The usual reasons: `$NodeName` has a typo; the node is named differently; the variable was read before `_ready()` (without `@onready`); the node was already freed.
- **Fix:** Check the spelling and capital letters of the node name against the Scene dock. Use `@onready var sprite: Sprite2D = $Sprite2D`. Print the variable to see if it is `null`.

C.13 "Node not found: 'Name'" (relative to '/root/Main')

- **Cause:** The path given to `get_node()` or `$` does not match the tree, or the node is not a child of this node.
- **Fix:** Compare the path to the tree character by character. The path starts at the node that owns the script. Use `$Parent/Child` for deeper nodes. Names are case-sensitive.

C.14 "Identifier 'x' not declared in the current scope"

- **Cause:** The variable is spelled wrongly, or it was declared in a different function, or it is declared after the line that uses it.
- **Fix:** Check the spelling. A variable declared inside a function cannot be used in another function: move it to the top of the script (outside any function) if it must be shared.

C.15 "Cannot infer the type of 'x' variable because the value doesn't have a set type"

- **Cause:** You used `:=` but Godot cannot know the type from the right-hand side (for example a value returned by `get_node()`, or a value read from a Dictionary).
- **Fix:** Write the type: `var enemy: Node2D = get_node("Enemy")` or `var n: int = my_dict["count"]`.

C.16 "Trying to assign value of type 'X' to a variable of type 'Y'"

- **Cause:** A typed variable was given the wrong kind of value, such as a `float` in an `int`, or text in a number.
- **Fix:** Convert: `int(x)`, `float(x)`, `str(x)`. Or change the type of the variable.

C.17 Division gives a surprising whole number (`7 / 2` is `3`)

- **Cause:** Dividing two `int` values gives an `int`.
- **Fix:** Make one side a float: `7 / 2.0`. Godot may warn about "integer division"; the warning is telling you this.

C.18 "Function 'x' has the same name as a function in the base class" or "signature does not match"

- **Cause:** You wrote a function whose name Godot already uses, or you overrode a built-in function with the wrong parameters.
- **Fix:** Use the exact signature, for example `func _process(delta: float) -> void:`. Rename your function if you did not mean to override.

C.19 My `_process` or `_ready` never runs

- **Cause:** The script is not attached to the node, or the node is not in the running scene, or the function name has a typo (`_Ready`, `_proccess`), or the node's process is turned off.
- **Fix:** Look for the small script icon beside the node. Check capitalization: it is `_ready`, `_process`, `_physics_process`, all lowercase. Add a `print("hi")` to confirm it runs.

C.20 The `class_name` I chose is rejected, or the script is "hiding a global class"

- **Cause:** The name matches a built-in Godot class (`Node`, `Timer`, `Camera2D`) or another script.
- **Fix:** Pick a unique name such as `PipHero`.

C.21 "Signal 'x' is already connected to given callable"

- **Cause:** You connected the same function twice, once in the editor (Signals tab) and once in code, or you connected in a function that runs many times.
- **Fix:** Connect only once. Connect in `_ready()`, not in `_process()`. If needed, check first with `if not sig.is_connected(fn):`.

C.22 "Attempt to call function 'x' in base 'previously freed'"

- **Cause:** You are using a node that has already been removed with `queue_free()`.
- **Fix:** Check before use: `if is_instance_valid(enemy):`. Or clear the variable after freeing it.

C.23 "Can't change this state while flushing queries. Use call_deferred() or set_deferred()"

- **Cause:** You changed a physics property or added/removed a physics body inside a physics signal (for example inside `body_entered`).
- **Fix:** Defer the change: `set_deferred("disabled", true)` for a collision shape, or `call_deferred("add_child", node)`.

C.24 "Parent node is busy setting up children, `add_child()` failed"

- **Cause:** You called `add_child()` on a node while it is still being set up (usually in `_ready()` of a child).
- **Fix:** Use `add_child.call_deferred(node)` or `get_parent().add_child.call_deferred(node)`.

C.25 Old tutorial code gives errors on `onready`, `export`, `yield`, or `KinematicBody2D`

- **Cause:** The tutorial was written for Godot 3.
- **Fix:** See the translation table at the end of Appendix A. In short: `@onready`, `@export`, `await`, `CharacterBody2D`.

C.26 A `print` shows `<Object#123456789>` or `[Node2D:12345]` instead of useful values

- **Cause:** You printed a node, not one of its properties.
- **Fix:** Print the property: `print(enemy.name, enemy.position)`.

C.27 A function using `await` seems to "run twice" or keeps going after the scene changed

- **Cause:** `await` pauses only that function; other calls can start it again, and after the scene is freed it may resume on a dead node.
- **Fix:** Guard with a flag (`var busy: bool = false`), or check `is_inside_tree()` after the `await`.

C.28 Saved `user://` data is not found, or `res://` write fails in the exported game

- **Cause:** In an exported game `res://` is read-only. Only `user://` is writable.
- **Fix:** Save to `user://`. Use *Project* > *Open User Data Folder* to find where it is on your computer.

Movement and physics problems

C.29 My character does not move

- **Cause:** One of: the input action does not exist or has a different name; the movement is in `_process` but sets `velocity` without `move_and_slide()`; the script is on the wrong node; the Input Map key is not assigned.
- **Fix:** In *Project Settings* > *Input Map*, check the action name against your code. Put `print(Input.get_vector("left", "right", "up", "down"))` in `_physics_process` to see if the input arrives. Always call `move_and_slide()` after setting `velocity`.

C.30 My character moves much too fast or much too slow

- **Cause:** The speed does not use `delta` (for direct `position` changes), or it uses `delta` when you set `velocity` for `move_and_slide()`.
- **Fix:** For `position += direction * speed * delta`, include `delta`. For `velocity = direction * speed` followed by `move_and_slide()`, do **not** multiply by `delta`: the function does that itself. Gravity (`velocity.y += gravity * delta`) does use `delta`.

C.31 The character falls through the floor

- **Cause:** The floor has no collision shape, the shape has no size, the layers and masks do not match, or the body moves very fast (tunneling).
- **Fix:** Add a `CollisionShape2D` with a shape to the floor body. A node with a yellow warning triangle is missing something: hover over it. Check collision layers and masks (see C.34). For fast bodies, enable *Continuous CD* on `RigidBody2D`, or slow the speed down.

C.32 The character cannot jump, or `is_on_floor()` is always false

- **Cause:** `move_and_slide()` has not been called yet this frame, the floor is not a static/physics body, or the floor's angle is too steep for the `floor_max_angle`.
- **Fix:** Call `move_and_slide()` each physics frame, before checking. Make the ground a `StaticBody2D` (or `TileMapLayer` with physics). Keep `up_direction` at the default `Vector2.UP` for side-view games.

C.33 The character sticks to walls, or gets stuck on tile edges

- **Cause:** A box-shaped collision with square corners catches on seams between neighboring tiles or shapes.
- **Fix:** Use a capsule shape for the character. Make tile collision shapes cover the full tile, and avoid very thin gaps.

C.34 Two things overlap but nothing happens (no collision)

- **Cause:** Collision **layers** and **masks** are set so that neither sees the other. A body's *layer* says "I am on this layer". Its *mask* says "I look for things on these layers".
- **Fix:** In the Inspector's *Collision* section, make sure at least one object has the other's layer checked in its mask. Layer 1 and mask 1 on both is the default and works.

C.35 `body_entered` or `area_entered` never fires

- **Cause:** The area's `monitoring` is off, the other object's layer is not in the area's mask, the area has no `CollisionShape2D` with a shape, or the signal is not connected.
- **Fix:** Check *Monitoring* is on, the mask matches, and the shape exists. In the Signals tab confirm the connection is there (a green icon appears beside the function). Print inside the handler to confirm.

C.36 `body_entered` fires too many times or at the start of the game

- **Cause:** The player or spawned object starts inside the area, or the signal handler runs for everything that touches it.
- **Fix:** Check which body entered: `if body.is_in_group("player"):`. Move objects so they do not start overlapped.

C.37 A `RigidBody2D` ignores the position I set in code

- **Cause:** The physics engine owns a rigid body's position. Setting `position` fights with it.
- **Fix:** Set the starting position before it enters the scene, or use `apply_impulse()` / `apply_force()` for movement. Use a `CharacterBody2D` when you want full control.

C.38 A bullet or fast object passes through a thin wall

- **Cause:** In one physics step the object moves farther than the wall is thick, so it never overlaps (**tunneling**).
- **Fix:** Make walls thicker, slow the object down, or for `RigidBody2D` enable *Continuous CD*. For bullets, a `RayCast2D` is often a better tool.

C.39 Things look jittery when the camera follows the player

- **Cause:** The camera follows in `_process` while the player moves in `_physics_process`, so they update at different rates.
- **Fix:** Use a `Camera2D` as a child of the player, and in its Inspector try enabling position smoothing. Keep all movement in `_physics_process`.

C.40 Objects spawned in the wrong place, at (0, 0)

- **Cause:** You set `position` before the node was added, or used `position` when you meant `global_position`.

- **Fix:** Use `add_child(node)`, then set `node.global_position = where`. Remember that `position` is relative to the parent.

User-interface problems

C.41 A Label or Button cannot be seen

- **Cause:** It is under another node, has a size of zero, has text the same color as the background, or is a child of a node that is not a `CanvasItem` (for example a plain `Node`) or is inside a hidden parent.
- **Fix:** Select it and look at its rectangle in the viewport. Set the *Text* property. Check *Visible* on it and all its parents. Check the draw order: nodes lower in the tree draw on top.

C.42 My buttons or UI do not respond to clicks

- **Cause:** Another Control (often a full-screen `ColorRect` or `Panel`) sits on top and catches the mouse. Its *Mouse* > *Filter* is *Stop*.
- **Fix:** Set the *Mouse Filter* of the covering node to *Ignore*. Check that the button is not disabled. Check the signal is connected.

C.43 UI elements do not stay where I put them when the window is resized

- **Cause:** Controls are not anchored, or they are inside a `Node2D` (which does not provide layout).
- **Fix:** Use the **Anchor preset** menu (the layout button in the viewport toolbar) to pin a control to a corner, edge, or center. Put UI inside a `Control` or a `CanvasLayer`, and use containers (`VBoxContainer`, `HBoxContainer`, `MarginContainer`, `CenterContainer`) for automatic layout.

C.44 I moved a Control in a container and it jumps back

- **Cause:** Containers control the position and size of their children. Manual moves are overwritten.
- **Fix:** Use the child's *Custom Minimum Size* and *Size Flags*, or put the node outside the container. Add spacing with `MarginContainer` or the container's *Separation* theme constant.

C.45 My game looks tiny, blurry, or stretched in a bigger window

- **Cause:** The window size and the game's base size are different, and the Stretch settings are not configured.
- **Fix:** In *Project Settings* > *Display* > *Window*, set the base *Viewport Width/Height* and under *Stretch* set *Mode* to `canvas_items` (a good default for most 2D games), and *Aspect* to `keep` or `expand`. For crisp pixel art, set the default texture filter to *Nearest* (*Rendering* > *Textures*).

C.46 Text is cut off or does not wrap

- **Cause:** The Label is too small.
- **Fix:** Set *Autowrap Mode* to *Word*, give the label a minimum size, or use a `RichTextLabel`.

C.47 My theme changes do not apply

- **Cause:** The theme was set on a node that is not an ancestor of the controls, or a Theme Override on the node itself takes precedence.

- **Fix:** Assign the theme to the **root Control** of the UI (it passes to all children), or set a project-wide theme in *Project Settings* > *GUI* > *Theme* > *Custom*. Clear overrides on the specific node.

C.48 The UI shows in the wrong place when the camera moves

- **Cause:** The UI is in the same layer as the game world, so the camera moves it.
- **Fix:** Put UI under a `CanvasLayer` node. A `CanvasLayer` ignores the camera.

Sound and animation problems

C.49 No sound plays

- **Cause:** The player has no stream assigned, `Autoplay` is off and `play()` is never called, the bus or volume is muted, the audio device is wrong, or the player was freed (for example inside a coin that was removed).
- **Fix:** Assign an *Stream* in the Inspector. Call `$AudioStreamPlayer.play()`. Check the bus volume in the Audio panel at the bottom of the editor. If the sound belongs to a node that is deleted, play it from a node that stays.

C.50 An animation does not play

- **Cause:** The animation name is misspelled, `play()` is not called, or another animation overrides it.
- **Fix:** Use the exact name from the Animation panel. For `AnimatedSprite2D`, make sure the animation has frames and *Playing* or `play("name")` is used. For `AnimationPlayer`, set *Autoplay* or call `play("name")`.

C.51 A Tween does nothing

- **Cause:** The tween was created and then killed or the node was paused, the property name is wrong, or it ended instantly with a duration of 0.
- **Fix:** Use `create_tween()` on a node that is in the tree. Use the exact property name (`"position"`, `"modulate"`, `"scale"`). Give a duration above zero.

Export problems

C.52 "No export template found at the expected path"

- **Cause:** Export templates are not installed (they are a separate download).
- **Fix:** Choose *Editor* > *Manage Export Templates* and download them for your exact Godot version.

C.53 The exported game runs but things are missing (data files, JSON, sounds)

- **Cause:** Godot exports only resources it knows about. Other files (like `.json` or `.txt`) are left out unless you include them.
- **Fix:** In the export preset's *Resources* tab, add a pattern such as `*.json` to *Filters to export non-resource files*.

C.54 The exported game crashes or shows a black window on another computer

- **Cause:** An older graphics card does not support the renderer you chose, or a path uses the wrong capitalization.

- **Fix:** Use the Compatibility renderer for maximum reach. Fix capitalization in paths (see C.10). Run the exported game from a terminal to see error output.

C.55 The web export does not run when I double-click `index.html`

- **Cause:** Browsers block a web export when opened from a file. It has to be served over HTTP.
- **Fix:** Use *Remote Debug* > *Run in Browser* from the editor, or serve the folder with a local web server. When you upload it, a host that sends the required headers (itch.io has a setting for this) is needed for versions that use threads.

C.56 My game's icon or name is wrong in the exported file

- **Cause:** The icon and name come from Project Settings and the export preset, not from the file name.
- **Fix:** Set *Project Settings* > *Application* > *Config* > *Name* and *Icon*. In the Windows export preset, set an icon in the *Application* section (a `.ico` file) if you want a custom Windows icon.

C.57 Windows warns "Windows protected your PC" when a friend opens the exported game

- **Cause:** The game is not code-signed, so Windows does not recognize the publisher.
- **Fix:** This is normal for unsigned games. Your friend can choose *More info* > *Run anyway*. Code signing is an advanced, paid step and is not needed for learning projects.

Last resorts

If none of the above helps:

1. **Delete the `.godot` folder** in your project (close Godot first). It is a cache that Godot rebuilds on the next open. Your scenes and scripts are not in it.
2. **Isolate the problem.** Make a new scene with only the part that fails. If it works there, something else in the old scene is interfering.
3. **Print what you think is true.** `print(node)`, `print(value)`, `print(is_on_floor())`. A surprise in the output is the bug.
4. **Compare with the finished project.** If you have the optional project files, open the one for the chapter and compare it with yours. Otherwise, re-read the chapter's steps and the complete scripts.
5. **Take a break.** The error is often obvious after ten minutes away.
6. **Ask for help well.** When posting to a forum or chat, include your Godot version, the exact error text, the few lines of code involved, and what you expected to happen.

Appendix D: Answers to the Review Questions

The review questions at the end of each chapter are meant to be answered from memory first, and checked afterward. To keep this book from giving the answers away, the answers are not printed here. They are in separate files, one per chapter:

Chapter	Answer file
1	`answers-ch01.md`
2	`answers-ch02.md`
3	`answers-ch03.md`
...	...
24	`answers-ch24.md`

Every chapter from 1 to 24 has its own file, named `answers-chNN.md`, where NN is the chapter number with a leading zero (for example `answers-ch07.md`).

How to use them

1. Finish the chapter and its project.
2. Answer all the review questions on paper or in a text file, even if you are unsure. A wrong guess teaches more than a skipped question.
3. Open that chapter's answer file and check each answer. Read the short explanation next to it, not only the letter or word.
4. For any question you missed, go back to the section named in the answer and read it again.

The Exercises at the end of each chapter are open-ended: there is usually more than one good way to build them. Compare your result with the chapter's project and with the "Expected result" text instead of looking for a single right answer.

> **Note:** If you got a question wrong, nothing is broken. The review questions are for you, not for a grade. Missing a question tells you exactly what to reread.

Answers: Chapter 1 review questions

1. A game engine is a toolkit that does the jobs nearly every game needs so you do not rebuild them. Any three of: drawing (rendering), playing sound, reading input, physics and collisions, loading and saving files, packaging the game for different systems.
2. Input, then Update, then Draw. The cycle repeats and each trip produces one frame.
3. About 1/60 of a second, roughly 16 milliseconds. If the work takes longer, the game feels choppy.
4. **False.** Godot is free and open source (MIT license). There are no fees on the games you make or sell.
5. **(b)** The standard version. The .NET version is only for C#, a 3.x version is a different, older engine with different code, and unofficial copies may be unsafe or modified.
6. `res://` is the top folder of the current project (short for "resources"); `res://main.tscn` is the file `main.tscn` at the top of the project. The file `project.godot` marks a folder as a Godot project.

7. The Scene dock shows *which nodes exist* and how they are arranged in a tree. The Inspector shows *the properties* (settings) of the one node you have selected.
8. **(b)** Remove only takes the project off the list. The files stay on disk, and you can import the project again.
9. Nodes are drawn in tree order, so nodes lower in the list are drawn on top. If `Background` came after `Icon`, the dark rectangle would be drawn over the icon and hide it.
10. Run Project (F5) starts the game from the project's main scene. Run Current Scene (F6) runs only the scene open in the editor, no matter what the main scene is. This is handy for testing one piece on its own.

Answers: Chapter 2 review questions

1. Properties (settings), functions or methods (things it can do), and signals (announcements it can make).
2. `Node2D`. (`CanvasItem` adds drawing and visibility; `Sprite2D` adds the texture.)
3. **False.** A node has at most one parent. That is what makes the structure a tree.
4. Mouth's parent is `Body`. Its siblings are `LeftEye` and `RightEye`. The root is `Pip`.
5. Child position is relative to its parent, so it appears at `(100 + 40, 100 + 0) = (140, 100)`. After the parent moves to `(200, 150)`, the child is at `(240, 150)`. It keeps the offset `(40, 0)` from the parent.
6. Names are used in paths such as `Body/LeftEye` to find a node. If two siblings shared a name, a path could not say which one is meant. Godot adds a number to duplicates (`Sprite2D2`).
7. The scene file is the saved blueprint (`pip.tscn`). An instance is one live copy of it placed inside another scene, with its own position, scale, and rotation. Many instances can come from one file.
8. **(c)** All three change, because each instance reads the blueprint.
9. Degrees; radians.
10. Down. In Godot 2D, `y` grows toward the bottom of the screen.

Answers: Chapter 2 exercises (hints)

1. A negative x scale flips Pip left-to-right (a mirror image). It is a quick way to make a character face the other way without drawing a second picture.
2. A plain `Node2D` root (`Door`) with children `Body`, `Window`, and `Knob`. Anything that needs a position is fine; `Node2D` is the simplest choice.
3. Yes. Hiding a parent hides all its children, so a hidden node disappears from the running game as well. (The eye icon in the Scene dock hides it only in the editor view.)
4. Any of the five coins changes when `coin.tscn` is changed and saved, as long as none has an Editable Children override.
5. Only instances without an override follow the new blueprint color. The overridden `Body` keeps its local color, because Godot remembers the override. Reverting the property on that instance makes it follow the blueprint again.

Answers: Chapter 3 review questions

1. A program is the whole list of instructions a computer follows. In Godot you do not write the whole program: the engine already runs the game loop, and a script (a `.gd` file) is one piece of code attached to a node that adds behavior to it.
2. `_ready()` runs the instructions in its body once, when the node has entered the scene tree and is ready. Godot calls it automatically; you never call it yourself. (The name must be spelled exactly `_ready`.)
3. `=` stores the value on the right into the variable on the left (`health` becomes 70). `==` asks a question, "are these equal?", and answers `true` or `false`.
4. (a) `int`; (b) `float`; (c) `String`; (d) `bool`.
5. It prints `18`. `n` starts at 10, `n -= 4` makes it 6, `n *= 3` makes it 18.
6. `7 / 2` is `3` (integer division drops the decimals), `7 / 2.0` is `3.5`, and `7 % 2` is `1`.
7. **False.** A name cannot begin with a digit (and `2fast` is not allowed).
8. Line (b). You cannot add a number to text with `+`. Line (a) is fine because `str(5)` converts the number to text first.
9. The name explains the number, and if the maximum changes you edit one line instead of hunting for every `100`. A bare unexplained number is called a magic number.
10. A local variable is created inside a function and exists only while that function runs (good for temporary working values, like `bonus`). A member variable is created at the top of the script, belongs to the node, and every function can use it (good for facts about the thing, like a hero's `health`).

Answers: Chapter 3 exercises (hints)

3. `17 / 5` is `3` each and `17 % 5` is `2` left over. The exact answer is `17 / 5.0` or `float(17) / 5`, which is `3.4`.
4. A receipt line might be `print("%s x %d = %.2f" % [item, qty, price * qty])`. With tax: `price * qty * (1.0 + TAX_RATE)`.

Answers to Chapter 4 review questions

1. `true` and `false`. A boolean can hold nothing else.

2. `=` is assignment: it stores a value in a variable. `==` is comparison: it asks whether two values are equal and gives a boolean. The comparison `==` belongs inside an `if` condition; writing `if x = 5:` is an error.

3. `false`. `5 > 3` is `true`, and `not` flips it.

4. **(b) B.** The first condition (`85 >= 90`) is false. The second (`85 >= 80`) is true, so "B" prints and the rest of the chain is skipped. Only the first true branch runs, so (d) is wrong.

5. **False.** `match` runs only the first matching pattern and then stops. There is no fall-through, so no `break` is needed.

6. The body runs **6 times** (`i` takes the values 2, 3, 4, 5, 6, 7). The last value is 7, because `range(2, 8)` stops before 8.

7. `break` ends the whole loop immediately. `continue` abandons only the current iteration and jumps to the next one; the loop then carries on.

8.

```
func triple(number: int) -> int:
    return number * 3
```

9. `names[1]` is `"Slime"` (indexes start at 0). `names[3]` is out of bounds because the valid indexes are 0, 1 and 2, so the game reports an error ("index out of bounds") and the script stops at that line.

10. Use `get` with a default: `stats.get("mana", 0)`. It returns 0 when the key is missing. (You could also check first with `stats.has("mana")`.)

Answers to Chapter 5 review questions

1. Godot calls `_ready()` **once**, when the node and all of its children have been added to the scene tree and are ready. It is the place for start-up work.

2. `_process` runs once per frame, and faster computers draw more frames per second, so adding a fixed 2 each frame gives a faster spin on a faster computer. Fix it by multiplying by the frame time: `rotation += 2.0 * delta`. Now 2 means "2 radians per second" everywhere.

3. **Children first.** Every child's `_ready()` runs before the parent's, so when the parent's `_ready()` runs, all its children are set up.

4. **(b)** `@onready var score_label: Label = $ScoreLabel`. Option (a) runs too early, so the variable would be `null`. Option (c) is wrong because `@export` is for Inspector settings and the lookup still runs too early. Option (d) is not GDScript.

5. `../Lamp3`. The `..` goes up to `Lamps`, then `Lamp3` goes down to the sibling.

6. A unique name finds the node from anywhere in the same scene without a path, so rearranging the tree (moving the label into another container) does not break the script.

7. **True.** A node can belong to any number of groups.

8. It calls the method `hide()` on every node in the group `enemies` (every node that has that method), which makes them all invisible.

9. The parentheses call the function immediately and pass its result (nothing) to `connect`. You must pass the function itself, without parentheses: `button.pressed.connect(_on_pressed)`.

10. *Emitting* is the sending of the announcement: the node says "this just happened" (the button emits `pressed` when clicked). *Connecting* is the one-time setup that says "when that announcement is made, call this function of mine". A signal with no connections can still be emitted; nothing will react.

Answers to Chapter 6 review questions

1. **Downward.** y increases going down the screen, so "up" is negative y.

2. `Vector2(100, 150)`. Moving up means subtracting from y: 200 - 50 = 150.

3. **10.** The length is the square root of 6 squared plus 8 squared: the square root of 36 + 64 = 100, which is 10 (a 6-8-10 triangle).

4. **No.** A child's position is measured in the parent's turned axes. With the parent rotated 90 degrees, the offset (0, -90) is turned too, so the child ends up 90 pixels to the *side* of the parent (to the right of it, since the rotation is clockwise on screen), not above it. The child's `global_position` is the parent's position plus that rotated offset.

5. `position` is measured from the parent's origin in the parent's space (local). `global_position` is measured from the world origin (global). They are equal only when no parent moves, turns or scales the node.

6. The one that is **lower in the Scene dock** (later in the tree) is drawn in front, because `z_index` ties are broken by tree order.

7. Set `hframes = 6` (columns) and `vframes = 3` (rows). The `frame` property then picks a cell (0 to 17, counted left to right and then top to bottom).

8. A `CanvasLayer`. It draws on its own layer independent of the camera, so its children stay fixed on screen.

9. **False.** `draw_*` functions work only inside `_draw()` (or a function it calls). Calling them in `_ready()` has no effect.

10. `queue_redraw()` asks Godot to call `_draw()` again, so the node repaints. You need it when the data that the drawing depends on changes after the first draw (for example a radius that grows in `_process`).

Exercise hints (selected)

- Exercise 4: with `hframes = 2`, `vframes = 2`, frames count left to right then top to bottom, so frame 2 is the **bottom-left** quarter of the icon (the sheet is 2 columns by 2 rows: 0 top-left, 1 top-right, 2 bottom-left, 3 bottom-right).

Answers to Chapter 7 review questions

1. **Polling** means your script asks about the current state on every frame ("is the key down right now?"). It is better for continuous behaviour, such as walking while a key is held. **Event-driven** input means Godot calls your function only when something happens (`_input`, `_unhandled_input`). It is better for one-off happenings or ones that carry details, such as "the player clicked at this position" or "Escape was pressed, open the menu". Any reasonable pair of examples is correct.

2. **b)** `Input.is_action_just_pressed("jump")`. It is true only for the single frame in which the key goes down. Option a) is true on every frame while held. Option c) is true only when the key is released.

3. **False.** An action can have any number of inputs (keys, mouse buttons, gamepad buttons, stick directions) bound to it. Any one of them triggers the action.

4. About **41 percent faster** (the diagonal vector `(1, 1)` has length about 1.41). `Input.get_vector()` avoids it by limiting the length of the result to 1. Calling `.normalized()` or `.limit_length(1.0)` on a hand-built vector also works.

5. **c)** "c". The arguments are, in order: negative x, positive x, negative y, positive y. Because the y axis grows downward in 2D, "up" on screen is negative y, so the third argument is the up action.

6. `delta` holds the **number of seconds** that passed since the previous frame (a float). At 60 FPS it is about 0.0167 seconds.

7. `position.x += 5.0` moves 5 pixels per *frame*. At 120 FPS that is 600 pixels per second; at 30 FPS only 150 pixels per second, so the game runs four times faster on the first computer. The fix is to define speed per second and multiply by `delta`: `position.x += 300.0 * delta` (where 300 is the desired pixels per second). Any sensible speed constant is fine as long as `delta` is used once.

8. **False.** `_unhandled_input()` only receives events that no other node, such as a button, has consumed. That is its purpose. `_input()` sees every event first.

9. If the remaining distance is no larger than one step, moving a full step would overshoot the target. Pip would then turn around, overshoot again, and jitter. Checking first lets the code place him exactly on the target and stop.

10. A physical key is identified by its **position** on the keyboard, not by the letter printed on it. The W-A-S-D cluster is then in the same comfortable place on every keyboard layout (for example on a French AZERTY keyboard, where those positions are labelled Z, Q, S, D), so the controls feel the same for everyone.

Answers to Chapter 8 review questions

1. (a) A floor: `StaticBody2D`. (b) The player: `CharacterBody2D`, because you steer it with code. (c) A coin: `Area2D`, because it should detect the player rather than block him. (d) A barrel that rolls downhill: `RigidBody2D`, because the physics engine should move it with gravity and collisions.

2. **False.** A physics body without a collision shape has no edge, so it is invisible to physics. The editor shows a warning triangle for this.

3. `move_and_slide()` already applies the length of the physics step to `velocity`, so multiplying by `delta` as well makes the movement far too slow (the velocity becomes about 1/60 of what you intended). The player would notice that Pip crawls. The correct code is `velocity = direction * speed`.

4. **b)** A direction of length 1 pointing out of the surface that was hit (toward the body that did the hitting).

5. A `CharacterBody2D` treats a `RigidBody2D` as a solid wall and does not push it on its own. The loop finds the rigid bodies that Pip touched during `move_and_slide()` and gives them an impulse in the direction away from Pip (the negative of the collision normal).

6. **a)** The coin's mask (2) includes Pip's layer (2), so the coin notices Pip. (Pip does not need the coin in his mask for the coin, an `Area2D`, to detect him.) The coin is an area and not solid.

7. **Once.** `body_entered` fires only when the overlap begins. Standing still for ten seconds produces no further signals. `body_exited` fires once when he leaves.

8. **False.** `gravity_scale = 0` only removes the effect of gravity. The body still responds to impulses, forces, and collisions.

9. The ghost wall is on layer 5. The crates' mask includes layer 5, so crates notice it and are blocked. Pip's mask (layers 1 and 3) does not include layer 5, and the ghost wall's own mask is empty, so nothing makes the physics engine treat the pair as colliding, and Pip passes through.

10. Godot prints: "Can't change this state while flushing queries. Use call_deferred() or set_deferred() to change monitoring state instead." The fix is to change the property after the physics step, for example `set_deferred("monitoring", true)`, or to call a function with `call_deferred("function_name")`.

Answers to Chapter 9 review questions

1. A coin is a small, reusable piece that should not depend on how the game keeps score. If the coin changed the score itself, you could not reuse it in a game without a score, and every change to the rules would mean editing the coin. Instead, Pip announces `coin_collected`, and `Main`, which owns all the rules (score, level, time), decides what that means. This is "signals up, calls down".

2. **b)** A `PackedScene`, a blueprint you can call `instantiate()` on. It is not a node in the game yet.

3. First `instantiate()` (build the copy), then set properties such as `position`, then `add_child()` (put it in the tree). Configuring first means that when the node enters the tree and its `_ready()` runs, it already has the right values, and it never appears for a frame at the wrong place.

4. **False.** `queue_free()` only schedules the node for deletion at the end of the current frame. The node still exists until then, which is why the coin needs a guard against being picked up twice.

5. `_picked` makes `pickup()` safe to call more than once. Because `queue_free()` is not immediate, a second overlap report in the same frame could otherwise count the same coin again and give double points.

6. `stop()` is called while handling the `hit` signal, which comes from inside the physics callback `area_entered`. Godot does not allow monitoring to be changed in the middle of the physics step ("Can't change this state while flushing queries"). `set_deferred` waits until it is safe.

7. **b)** It names the phases and lets you compare against readable names such as `State.PLAYING`. The values are stored as numbers, there is no limit of two states, and it does not affect speed.

8. Coins: `4 + level * 2 = 4 + 3 * 2 = 10`. Slimes: `mini(level, 5) = mini(3, 5) = 3`. So 10 coins and 3 slimes.

9. Yes. A mask of 6 is 2 + 4, so it contains layer values 2 and 4, meaning layers 2 (coins) and 3 (hazards). The coin's layer value is 2, which is in the player's mask, so the player notices the coin and `area_entered` fires.

10. The HUD only displays numbers and raises one signal, `start_requested`, when the button is pressed; it does not know who listens (signal going up). `Main` knows the HUD as its child and calls its functions such as `update_score(value)` and `show_game_over(score, best)` to tell it what to show (calls going down). The child never reaches up into the parent's business, so the HUD can be changed or reused without touching `Main`.

Answers: Chapter 10 review questions

1. **6 game frames per animation frame, on average.** 60 game frames per second divided by 10 animation frames per second is 6. The animation does not need to match the game's frame rate; the engine simply shows each animation picture for about 6 screen redraws.

2. **b) `AnimatedSprite2D`.** A walk cycle is a list of drawn pictures, which is exactly what `AnimatedSprite2D` and its `SpriteFrames` are for. A `Tween` is for one-off property changes, an `AnimationPlayer` is better when many properties change together, and a `Timer` does not animate anything.

3. **False.** Calling `play("run")` while `run` is already playing keeps it running; it does not go back to frame 0. This is why you can safely call it every frame from `_process` to describe what the character should be doing. (The project still checks `body.animation` first, which makes the intention clear.)

4. **It mirrors the picture left to right.** Artists draw a character facing one direction. Setting `flip_h = true` shows the same frames mirrored, so you only need one set of frames for both walking directions. In the project it is set with `body.flip_h = dir < 0.0`.

5. **b) Discrete.** A frame number is a whole number that cannot be blended; frame 2.4 does not exist. Discrete update mode writes the value only at the keyframes. Continuous is for values that glide, such as positions and colors.

6. **Left of the colon: the node (`Body`); right of the colon: the property (`modulate`).** The node path is measured from the `AnimationPlayer`'s root node, which by default is the player's parent. So `Body:modulate` means "the child named `Body` of the player's parent, its `modulate` property."

7. **a)** `tween.tween_property(self, "modulate:a", 0.0, 0.5)`. The `:a` selects the alpha (transparency) part of the `modulate` color. There is no property called `alpha` on a node, `fade` is not a tween method, and `visible` is a true/false value that cannot gradually change.

8. **Each step starts after the previous one finishes.** That is the default: steps are chained in order. To run a step at the same time as the one before, put `.parallel()` in front of it (`tween.parallel().tween_property(...)`), or call `tween.set_parallel(true)` once to make all steps run together.

9. **False.** A tween that has finished is done and cannot be restarted. To play an effect again you create a new tween, usually inside a function that builds it. Only a tween that is still running can be paused and resumed. If you need to cancel a running tween, call `kill()`.

10. **Two tweens are fighting over the same property.** When the mouse leaves before the grow tween has finished, the shrink tween starts while the old one is still running, and both write to `scale`. The fix is to store the tween in a variable and call `kill()` on it before creating the next one, as the project's `_on_button_hover` does.

Answers: Chapter 11 review questions

1. **A sample is one measurement of the sound wave at one instant, stored as a number. The sample rate is how many of those measurements are stored or played each second (in hertz).** A higher sample rate can capture higher frequencies; the highest representable frequency is about half the sample rate. 44,100 Hz is common; the project uses 22,050 Hz.

2. **b) WAV.** WAV stores raw samples, so playing it needs almost no processing and starts instantly. Compressed formats (Ogg, MP3) must be decoded while playing and are better for longer music. A `.txt` file is not audio.

3. **True.** Each 6 dB step roughly halves or doubles the amplitude, so -6 dB is about half. 0 dB is the "no change" setting.

4. **Audio buses.** Put music players on a `Music` bus and effects on an `SFX` bus (both sending to `Master`). Then one slider per bus changes all sounds on that bus at once, without touching individual players.

5. **b) `linear_to_db(v)`.** Sliders are linear (0 to 1) while buses and `volume_db` use decibels. `db_to_linear` does the opposite conversion.

6. **To avoid clicks.** If the wave starts or stops at a nonzero value, the speaker jumps instantly and you hear a click or pop. A short fade-in and fade-out (an envelope) lets the wave begin and end smoothly at zero.

7. **c) 880 Hz.** Twelve semitones (one octave) doubles the frequency: 440 x 2 = 880. The formula is `440 * 2^(semitones / 12)` with semitones = 12.

8. **One `AudioStreamPlayer` plays one sound at a time; each new `play()` restarts it and cuts the previous one off.** The fix is a pool of several players, using whichever one is free for each new sound (or `max_polyphony` on the player).

9. **False.** `AudioStreamGenerator` is for live sound. You call `play()` first, then each frame push new samples into its playback object with `push_frame`. The sound is computed while it plays.

10. **It is positional: it gets quieter with distance from the listener (attenuation, `max_distance`), and it pans between the left and right speakers according to where the node is.** (`panning_strength` adjusts the panning.)

Answers: Chapter 12 review questions

1. **Any two of:** it is much faster to build (paint with a brush instead of placing nodes); it runs faster (tiles are drawn in batches, not as thousands of nodes); it is small to store (just a list of cell and tile); changing the tile art changes every use at once; tiles can carry collision shapes and custom data.

2. **b) `(3, 1)`.** Divide each pixel coordinate by the tile size and round down: 100 / 32 = 3.1 gives 3, and 40 / 32 = 1.25 gives 1. In code: `layer.local_to_map(Vector2(100, 40))`.

3. **False.** Cell coordinates are whole numbers (`Vector2i`). There is no cell 3.5. Pixel positions are the ones that can have decimals (`Vector2`).

4. **Atlas coordinates say where a tile lives in the atlas image (a position in the tile picture sheet, in tiles); map coordinates say which grid cell of the level the tile is placed in.** For example, `set_cell(Vector2i(10, 4), 0, Vector2i(4, 1))` puts the tile from atlas position (4, 1) into level cell (10, 4).

5. **b) The TileSet's Tile Size does not match the atlas tiles.** The default tile size is 16 by 16; if your atlas uses 32 by 32 tiles, they get sliced at the wrong places.

6. **So that only the walls need collision, and so that code can ask two separate questions.** The Walls layer carries the physics shapes; the Ground layer stays cheap and has none. It also allows two tiles in the same cell (grass and a bush), and in code "what is the floor here?" and "is something blocking here?" are answered by different layers.

7. **c) -1.** `get_cell_source_id` returns -1 for an empty cell. (`get_cell_tile_data` returns `null` for an empty cell, not -1.)

8. **Because each of the four sides is either connected or not, giving 2 x 2 x 2 x 2 = 16 combinations.** Each combination needs its own tile picture (from "no neighbors" to "all four neighbors"), and the terrain system picks the one that matches.

9. **True.** An empty cell has no tile data, so the function returns `null`. Reading `get_custom_data` on `null` would cause an error, so write `if data:` first.

10. **Any two of:** the TileSet has a Physics Layer (and its Collision Layer is on); the wall tile actually has a collision polygon drawn on that physics layer; the layer's *Collision Enabled* is on; the character's collision mask includes the physics layer's collision layer; the character is a physics body (`CharacterBody2D`) that moves with `move_and_slide` rather than by setting `position` directly.

Answers to Chapter 13 review questions

1. **Up.** The y axis points down in Godot 2D, so a negative `velocity.y` moves the character toward the top of the screen. That is why `jump_velocity` is `-520.0`.

2. **True.** `is_on_floor()` reports the result of the most recent `move_and_slide()`. That is why we call it at the start of the next physics frame and it still gives a correct answer: little changes in 1/60 of a second.

3. **90 pixels.** Height = v^2 / (2g) = 600 × 600 / (2 × 2000) = 360,000 / 4,000 = 90.

4. **150 pixels.** Air time = 2v / g = 1200 / 2000 = 0.6 seconds. Distance = run speed × air time = 250 × 0.6 = 150.

5. Coyote time solves the **late** press: the player presses jump a moment after walking off a ledge, and the game still allows it. The jump buffer solves the **early** press: the player presses jump a moment before landing, and the game remembers the press and jumps on landing. Both are implemented with a countdown variable that is refilled by an event and reduced by `delta`.

6. **(b).** When the button is released while the character is still rising (`velocity.y < 0`), multiplying the upward speed by 0.5 cuts the jump short. (a) would make the whole game's gravity change, (c) would cut every jump, and (d) only affects falling.

7. Height alone would let Pip destroy a slime while he is rising through its top edge, or hurt him when he lands exactly on it a frame late. Velocity alone would let a falling Pip who merely brushes the side of a slime squash it. Requiring both (moving down AND clearly above the slime's center) matches what a player means by "I landed on its head".

8. The `body_entered` signal is emitted during the physics step, when Godot is busy processing collisions. Changing a collision shape at that moment is not allowed and prints an error. `set_deferred` postpones the change to a safe moment at the end of the frame.

9. **(b).** Objects that only emit signals stay reusable: the same coin could be used in a game with no score. The controller (`game.gd`) is the only place that knows the rules. (a) is false, GDScript allows direct calls; (c) is false, areas can call functions; (d) is false, signals are fast.

10. A `CanvasLayer` draws its children on a separate sheet that is independent of the camera, so its content stays fixed on the screen. The HUD uses one so the labels do not scroll away with the level. A second layer with a negative `layer` number draws behind the world and is used for the sky.

Answers to Chapter 14 review questions

1. A `Control` (1) has a rectangle with a position and a size, not just a point; (2) has layout rules (anchors, offsets, minimum size, size flags) so it adapts when its parent changes; (3) takes part in mouse and keyboard input and focus. Interface nodes are green; 2D world nodes are blue.

2. It stays attached to the **top-right corner**. Anchors 1 on the horizontal axis mean "the right edge of the parent" and anchors 0 on the vertical axis mean "the top edge". This is the Top Right preset, ideal for a score label.

3. **False.** A container overrides the position and size of its children each time it re-sorts. Change the child's minimum size, size flags, or the container's own constants instead (or take the child out of the container).

4. Put an empty `Control` (a **spacer**) between them in the `HBoxContainer` and give it the horizontal size flags **Fill + Expand**. It takes all leftover width and pushes the score label to the right edge.

5. **(b)** `OptionButton`**.** `LineEdit` is for typed text, `ProgressBar` displays a filled bar, `GridContainer` is a layout container.

6. Audio buses use decibels, a logarithmic scale, while a slider is linear (0 to 1). Passing the slider value through `linear_to_db()` makes the loudness change feel even as the slider moves, and 0 becomes minus infinity (silence), which is why the code also mutes the bus at zero.

7. Strongest first: (1) theme overrides set on the control itself; (2) a theme assigned to the control, then to its parent, and up the tree; (3) the project's default theme; (4) Godot's built-in default theme.

8. `grab_focus()` gives keyboard and gamepad focus to that control, so the next Enter or accept press activates it and arrow keys move away from it. Menus call it when they appear because otherwise no control has focus and keyboard or gamepad players cannot start using the menu.

9. **(c) When Paused.** It must run while the tree is paused (so it can respond to clicks) and stay idle otherwise. Pausable and Disabled would freeze it during pause, and Inherit from a pausable parent would do the same.

10. Keeping the truth in variables and redrawing in one function means the display can never disagree with the data: there is only one place that writes to the labels, bars, hearts and slots. Updating labels inside each function would repeat code and is easy to forget in one of them, which produces out-of-date numbers.

Answers to Chapter 15 review questions

1. A coin that updates the score label has to know the exact path of that label, so it only works in scenes that contain it, and it breaks when the label is renamed. A coin that emits `collected` knows nothing about the outside world: it works in any scene, and any number of listeners (score, sound, achievements) can react, or none at all. Reuse and testing become easy.

2. **(b)** `button.pressed.connect(_on_pressed)`. Without parentheses you hand over the function itself. Option (a) calls the function immediately and passes its result, (c) is Godot 3 style with a string, and (d) has the arguments in the wrong order.

3. Bind the argument: `button.pressed.connect(_on_choice.bind(3))`. The bound value `3` is added to the call, so `_on_choice(3)` runs. A lambda such as `func() -> void: _on_choice(3)` would also work.

4. **False.** Connecting the same callable to the same signal twice prints an error ("already connected") and the second connection is refused. Guard with `is_connected()` or connect only once, in `_ready()`.

5. `CONNECT_ONE_SHOT` removes the connection automatically after the first time it is called. Example: the door's `entered` handler in the Key Room, which should finish the room only once; or "play the intro cutscene the first time the player enters this area".

6. Signal names should be lowercase `snake_case` and describe something that already happened, and a signal without parameters needs no parentheses. It should be `signal died`. (Parentheses are allowed but unnecessary when empty; the capital letter breaks the naming convention and `died` is the past-tense form.)

7. (a) The function pauses at that line and continues after 1.5 seconds (it becomes a coroutine; a caller that does not await it carries on immediately). (b) Nothing happens to the rest of the game: everything else keeps running normally.

8. The old scene is removed from the tree and **freed**, so all its variables are lost. To keep a score you can (any two): pass it as data when creating the new scene; keep it in a permanent parent such as a scene manager that only swaps its child; keep it in an autoload (Chapter 16); or save it to disk (Chapter 17).

9. **(b).** Calling `setup(data)` before `add_child` puts the data in place before `_ready()` runs, so `_ready()` can use it right away. (It does not make the scene hidden, `add_child` is not slow, and Godot does not require the order; the order is a design choice.)

10. Before `add_child`, the node is not in the tree, so `_ready()` has not run and the `@onready` variables (the labels) are still empty. Writing to them in `setup()` would cause a null-instance error. Storing the values in plain variables and showing them in `_ready()` is safe, because by then all the nodes exist.

Answers: Chapter 16 review questions

1. Changing scenes frees the old scene, and every node in it goes with its script's variables. The new scene is a fresh copy, so its variables start again from their default values.
2. Godot (a) creates the autoload automatically when the game starts, before the main scene; (b) adds it to `root` and keeps it there until the game closes, through every scene change; (c) gives it a global name so any script can use it directly.
3. In **Project > Project Settings**, on the **Globals** tab, **Autoload** sub-tab. Click **Select Script/Scene**, pick the file, and leave **Enable** ticked in the Global Variable column.
4. **False.** If the Global Variable option is on (the default), you simply write the name, for example `GameState.gold`. No `get_node()` is needed.
5. **b.** Autoloads load top to bottom, so if one uses another in its `_ready()`, the used one must be listed first.
6. The function is the single doorway into the data. It can enforce rules (gold never below zero) and emit `gold_changed`, so displays update. Writing to the variable directly skips both.
7. Game state stores data about now (gold, lives, inventory) and emits signals when it changes. An event bus stores nothing and only announces things that happened, such as `toast` or `boss_defeated`. Any sensible example of each is correct.
8. The other script sees the changed value. `load()` returns the same cached object each time, so both scripts hold a reference to one resource.
9. **c.** `duplicate()` makes a separate copy. `load()` and `preload()` return the shared object.
10. **False.** The label may be created after the gold was earned (for example in a new scene), so it missed earlier signals. Reading the current value in `_ready()` and then connecting is the correct pattern.

Answers: Chapter 16 exercises (hints)

1. Copy `gem.tres` in the FileSystem dock (or make a new resource), set Id `apple`, Value `2`, a red color. Add `"res://items/apple.tres"` to `ITEM_PATHS`. No other change is needed because the room builds its buttons from the catalog.
2. Add `@export var rarity: String = "common"` to `item_data.gd`, set it on `gem.tres`, and extend the label format string in `bag.gd`, for example `"%s x%d [%s] (%d gold each)" % [item.display_name, count, item.rarity, item.value]`.
3. A `Button` in `main.tscn` connected to a function that calls `GameState.reset()`. For `spend_gold`:

```
func spend_gold(amount: int) -> bool:
    if amount > gold:
        return false
    gold -= amount
    gold_changed.emit(gold)
    return true
```

4. In `events.gd` add `signal item_sold(item_id: String, total: int)` (with the `@warning_ignore` line above it). Emit it at the end of `sell_all` with `Events.item_sold.emit(item.id, count * item.value)`. The achievements label connects in `_ready()` and checks `total >= 10`.
5. Add `@export var buy_price: int = 0` to `ItemData` and set it on each `.tres`. In the shop button handler, call `if GameState.spend_gold(item.buy_price): GameState.add_item(item) else: Events.toast.emit("Not enough gold")`. Since the Hud listens to `gold_changed`, it updates in every scene without extra work.

Answers: Chapter 17 review questions

1. Variables live in memory (RAM), which is emptied when the program closes. To keep data you must serialize it (turn it into text or bytes) and write it to a file on disk, then read it back when the game starts.
2. `res://` is the project's own files; in an exported game they are packed inside the game and are read-only. `user://` is a writable, per-user folder made for saved data, and Godot maps it correctly on every system.
3. **False.** The prefix is the same in code, but the real folder differs: `%APPDATA%\Godot\app_userdata\...` on Windows, `~/Library/Application Support/Godot/app_userdata/...` on macOS, `~/.local/share/godot/app_userdata/...` on Linux, and browser storage on the web.
4. It is the default returned when the key (or the whole file) is missing. It matters on the first run, when no settings file exists yet, and when an older file lacks a key that a newer version of the game added.
5. It returns an `Error` code. `OK` (the number 0) means the file was written successfully; any other value says what went wrong (for example `ERR_FILE_CANT_WRITE`). `error_string()` turns it into words.
6. **b.** JSON has one "number" type and Godot's parser returns it as a `float`, so `42` becomes `42.0`. Convert with `int()`.
7. **c.** JSON has no `Vector2`. Convert it to an array of two numbers (or a string) before saving.
8. So that a bad file cannot leave the game half-changed. If the third check failed after the first two values had already been assigned, the game would be in a mixed state. Checking everything first and then applying means the change is all-or-nothing.
9. Write the new data to a temporary file first. Only when that has finished successfully, rename the temporary file over the real save. If the game or computer dies during the write, only the temporary file is damaged and the old save is untouched. It prevents losing both the old and new save to a half-written file.
10. **False.** A `.tres` file can reference scripts, so a hand-made file could cause code to run. JSON and ConfigFile files only produce plain data.

Answers: Chapter 17 exercises (hints)

1. In `to_dict` add `"high_score": high_score`; in `from_dict` check `_is_number(data.get("high_score"))` along with the other checks and assign `high_score = int(...)` in the apply section. Add `check(sm.high_score == 9, "high score restored")` to the test.
2. `config.set_value("player", "favorite_color", favorite_color)` in `save_settings`, `favorite_color = str(config.get_value("player", "favorite_color", favorite_color))` in `load_settings`, then a `LineEdit` that connects `text_changed` like the name field.
3. Make `save_path` a variable that is set when a slot is chosen, for example `save_path = "user://save_%d.json" % slot`. Use `FileAccess.file_exists()` on each slot's path to label the option ("Slot 2 (used)" or "Slot 2 (empty)"). Remember the `.tmp` file name is built from `save_path`, so it follows automatically.
4. A `Timer` with `wait_time = 10.0` and `autostart = true`, connected to a function that calls `SaveManager.save_game()` and sets the status text. If the returned error is not `OK`, show `error_string(err)` and consider trying again later rather than ignoring it.
5. In `from_dict`, read `version`; accept 1 and 2. For version 2 validate `lives` as a number; for version 1 skip it and set `lives = 3`. Change `to_dict` to write `"version": 2`. Tests: save a hand-made version-1 text file and check `lives == 3` after loading; save normally and check `lives` round-trips.

Answers: Chapter 18 review questions

1. (1) The screens and the arrows between them (screen flow); (2) the rules of the game; (3) the shared data and who owns each piece.
2. The score and lives are needed by several scenes (play screen, HUD, game over screen), and the game over screen appears after the play scene has been deleted. Only an autoload survives the scene change, so the numbers are still there to show.
3. **False.** A `CanvasLayer` only controls drawing order. The menu keeps working because its **Process Mode** is set to **Always**; its children inherit that setting.
4. The node keeps running (`_process`, input, buttons) even when the tree is paused. In this game, the `PauseMenu` (and, through its own setting in code, the `Sfx` autoload) uses it.
5. The `paused` flag belongs to the whole tree and survives a scene change. The title screen would load frozen: buttons would not respond and nothing would run, except nodes set to Always.
6. `body_entered` fires for any physics body that touches the area, not only Pip. Only the player has a `hurt()` function, so checking the group first prevents errors (and wrong reactions) if something else ever enters.
7. **b.** `maxf(0.9, x)` returns `x` but never less than 0.9. Option (a) caps the value from above instead (it would always be at most 0.9).
8. Writing the file only when needed is cheaper and avoids needless disk writes. It also means the file holds exactly one number that only goes up, so there is nothing to compare or repair.
9. The slime needs to know whether the hit counted. If Pip was still invulnerable, `hurt()` returns `false` and the slime does not pop; otherwise it returns `true` and the slime is removed.
10. **True.** The node is added to the scene at the position it has at that moment (0, 0 by default), so setting the position first avoids a flash in the wrong place.

Answers: Chapter 18 exercises (hints)

1. Add `var points: int = 10` to `coin.gd` and use `Game.add_score(points)` in `_on_body_entered`. In `_spawn_coin`, `if randi() % 5 == 0: coin.points = 50; coin.scale = Vector2(1.5, 1.5)`. Remember that the coin's `_process` overwrites `scale.x` for the spin effect; to keep a bigger coin, draw a larger circle by adding a `size` variable in `_draw` instead of scaling the node.
2. There is no single right answer. Keep notes: the starting interval (3.0), the ramp (0.03 per second), the speed ramp (1.2 per second), and `START_LIVES`. Change one number at a time.
3. Create `shield.tscn` as an Area2D like the coin; on `body_entered` call `body.shield()` and `queue_free()`. In `player.gd` add `func shield() -> void: _invulnerable = 4.0`. Spawn it with a rare timer or a one-in-twelve chance in `_spawn_coin`.
4. Turn off Autostart on both timers; in `_ready()` of the play scene show a Label and a one-second Timer that counts 3, 2, 1, "Go!", then call `coin_timer.start()` and `slime_timer.start()`. To freeze Pip, set `player.set_physics_process(false)` until the count is over, then `true`.
5. Add `difficulty` (int) to `Save` with a ConfigFile key, store `high_score_easy`, `high_score_normal`, `high_score_hard` (or a Dictionary keyed by difficulty), and have `Game.start_run()` set lives from a small table. The slime speed ramp in `play_scene.gd` can read a multiplier from `Game`.

Answers: Chapter 19 review questions

1. Y points up. "Forward" is the negative Z direction (X is right, and +Z points toward the viewer).
2. **False.** A `Mesh` is a resource that holds the shape. A `MeshInstance3D` is a node that places a mesh in the world. One mesh can be shown by many instances.
3. A 2-unit-tall box is centered on its node, so at `y = 0` half of it (1 unit) sinks into the floor. To rest on a floor whose top is at y = 0, give it `y = 1` (half its height).
4. `DirectionalLight3D`. The sun is so far away that its rays are parallel everywhere, so only the light's rotation matters; moving the node changes nothing.
5. **(b)** Metallic up and Roughness down. Roughness up blurs the shine, Emission makes a surface glow, and Scale only resizes it.
6. There is no current `Camera3D` (add one and tick Current). Without a camera the game window shows nothing of the 3D world.
7. If the whole `Player` turned, the `CameraPivot` and camera (its children) would turn with it, and the view would spin whenever Pip changed direction. Turning only `Body` keeps the camera steady.
8. `lerp_angle` understands that angles wrap around, so it turns the short way (for example from 350 to 10 degrees through 0), where plain `lerp` would swing the long way round.
9. **False.** The mesh and the collision shape are separate resources. Changing one never changes the other; you must set both.
10. After a 90 degree turn, "forward" points sideways in the world (along X). The line `camera_pivot.global_transform.basis * Vector3(input_dir.x, 0.0, input_dir.y)` rotates the input by the pivot's turn, so forward always means "away from the camera."

Answers: Chapter 20 review questions

1. Juice is extra feedback (movement, flashes, particles, shake, sound) that makes actions feel good without changing the rules. Examples: a sprite squashing when hit; a floating damage number; sparks when a coin is picked up.
2. **False.** Screen shake only changes how the hit feels. Scores, health, and all rules stay the same.

3. The `GPUParticles2D` node controls how many particles there are, how long they live, whether it is one-shot, and the texture. The `ParticleProcessMaterial` controls how each particle moves and changes: direction, speed, gravity, size, and color over its life.
4. **(b)** `One Shot` makes it emit a single batch, and `Explosiveness` of 1 makes all particles appear at the same moment. Preprocess pre-runs a steady effect, Local Coords decides whether particles follow the emitter, and Texture and Emitting do not control timing of the burst.
5. Squaring makes small trauma values give a very small shake and large values a strong shake, and it makes the shake die away smoothly at the end, instead of buzzing at a low level for a long time.
6. With the time scale at 0.05, a normal timer also runs at 5 percent speed, so 0.1 seconds takes 20 times longer (2 seconds). Fix: pass `true` as the fourth argument, `create_timer(0.1, true, false, true)`, so it ignores the time scale.
7. `COLOR`.
8. A blend a quarter of the way from `a` to `b`: 75 percent `a` plus 25 percent `b`.
9. Get the `ShaderMaterial` and call `set_shader_parameter("flash_amount", value)` on it, for example `(material as ShaderMaterial).set_shader_parameter("flash_amount", 1.0)`. A tween can also animate the path `"shader_parameter/flash_amount"`.
10. All twenty enemies share the same `ShaderMaterial` resource, so they share its uniform values. Make the material unique per enemy: tick `Resource > Local to Scene`, or in code use `material = material.duplicate()`.

Answers: Chapter 21 review questions

1. Game AI is the code that decides what a computer-controlled character does next. No, it is usually simple rules (like a state machine), made to be believable and fair rather than clever.
2. Sense, think, act.
3. **True.** With One Shot off, the timer restarts by itself each time it reaches zero and emits `timeout` every time.
4. For a quick one-off pause inside a single function (wait, then continue). Use a `Timer` node when the timer belongs to something that must be started, stopped, or restarted, such as a state's duration or a cooldown.
5. So every state starts from a clean slate in one place: timers are stopped, the velocity is cleared, and the new state's setup code runs. If states changed from many places, a timer from the old state could still fire, or setup could be skipped.
6. **False.** A state machine is in exactly one state at a time. That is its main rule.
7. **(b)** The mesh should leave a margin about the size of the character's radius, because paths are planned for the center point and the body has a width. It need not hug walls exactly, it is used in 2D as well, and it does not replace collision shapes (collision stops bodies, navigation plans routes).
8. The navigation map is not ready on the first frame, so no path is found. Fix: `await get_tree().physics_frame` before setting the target.
9. It returns an empty dictionary. The slime treats `hit.is_empty()` as "nothing is blocking the view," so it can see the player.
10. The player moves, but a path is a fixed list of points. Setting the target once would send the slime to where Pip was. A repeating timer (every 0.25 seconds) updates the target often enough to feel smart without recalculating a path on every frame.

Answers: Chapter 22

1. Reproduce it; narrow it down; guess and test; fix one thing and retest. (Then repeat if it is still broken.)

2. A loud bug stops the program or prints an error (for example "Node not found: Shield" when pressing Space). A silent bug runs without any error but gives wrong behavior (for example the score label never changing). The engine cannot know what you meant, so silent bugs need printing, breakpoints or the Remote tree.
3. `null instance` means the object you used is `null`, "nothing is there". First check that the node or variable really exists: is the node name and path spelled correctly (capital letters too), and was the variable ever assigned?
4. False. Start with the first error. Later ones are often fallout; for example, a missing node (first error) produces a null-instance error (second).
5. (b) F10 is Step Over. F11 is Step Into, F12 is Continue, F9 toggles a breakpoint.
6. The stack trace lists the functions that were running, newest first. The first line is where the error happened. The second line shows which function called it, which helps when a function is called from many places and only one caller passes the bad value.
7. False. Remote tree changes affect only the running copy of the game and are lost when it stops. They never change your saved scene.
8. Removing an item shifts the later items down one place, but the loop moves on to the next position, so one item is skipped. The usual fix is to loop over a copy: `for coin in coins.duplicate():`.
9. (c) `git commit`. `git add` puts changes on the waiting list; `git commit` saves them as a snapshot.
10. `.godot/` holds caches that Godot rebuilds by itself; it is large and changes constantly, so it only clutters history. Lowercase `snake_case` file names avoid errors on case-sensitive systems (Linux, Web exports), where `Player.png` and `player.png` are different files even though Windows treats them as the same.

Answers: Chapter 23

1. An export template (the engine built for one platform) and your game data (scenes, scripts, resources, packed in a `.pck` file, possibly embedded in the executable).
2. False. Templates are a separate download (Editor > Manage Export Templates...).
3. The template is the engine that will run your game. A different version can differ in how it reads project data, so Godot refuses to export or the game may behave wrongly.
4. (b) `export_presets.cfg`. `project.godot` holds project settings, not presets.
5. Create the folder `build/web` first (Godot does not create destination folders), then export again.
6. Browsers block a page opened from disk (`file://`) from loading the game's other files. Serve the folder over HTTP, for example with `python -m http.server`, then open `http://localhost:<port>/`.
7. It returns `true`. Practical use: hide a Quit button in the browser, where a page cannot quit.
8. False. `res://` is read-only in an export. Use `user://` for anything the game saves.
9. (b) `--headless`.
10. Any two of: a debug build includes debugging features and can print to a console (an extra `.console.exe` on Windows) while a release build is the one for players; debug builds are bigger and slower; asserts and debugger features run only in debug; release exports store scripts compiled, and are what you give to players.

Answers: Chapter 24

1. `index.html` must be at the top level of the zip, not inside a folder within the zip.
2. Choose **HTML** as the kind of project, and tick **This file will be played in the browser** next to the uploaded zip.
3. False. Make a playable core loop first (with plain shapes), and add art and polish afterwards; otherwise you risk having no working game at the deadline.

4. The core loop is the cycle of actions the player repeats: action, result, reward or danger, then action again. Example: in Pip Dodge, slide, avoid a block, survive longer while the speed rises, slide again.
5. (b). Cut it to the must-have items and write the rest on a cut list. Working longer, adding features or switching games all keep the scope problem.
6. The game must be able to explain itself. If you coach the tester, you hide the places where a stranger would be confused.
7. False. Godot 3 uses different names and syntax (for example `KinematicBody2D`, `yield`, `export var`), so Godot 3 tutorials need translating and will not run as written.
8. Export problems (missing files, wrong paths, case errors, saving to `res://`) only show up in the exported build. Finding them early leaves time to fix them.
9. Any three of: a one-line description of the game, the controls, details such as length or features, credits for tools and assets, screenshots, an invitation for feedback.
10. Any two of: finishing teaches the whole pipeline (design, build, export, publish), gives playable proof for a portfolio, produces feedback from real players, builds the habit of finishing, and avoids the discouragement of abandoned projects.

Appendix E: Glossary

Terms are listed alphabetically. Chapter numbers show where the idea is mainly taught. Terms in *italics* inside a definition have their own entry.

A

Action (input action). A named input such as `jump` that can be triggered by several keys, buttons, or sticks. Set up in the *Input Map*. (Ch. 7)

Algorithm. A step-by-step recipe for solving a problem. (Ch. 4)

Alpha. The fourth part of a *color*, controlling see-through-ness. 0 is invisible, 1 is solid. (Ch. 6)

Anchor. A setting that pins a *Control* to a part of its parent (a corner, an edge, the middle) so that it keeps its place when the parent is resized. (Ch. 14)

AnimatedSprite2D. A node that plays a sequence of images (frames) to animate a sprite. (Ch. 10)

Animation. Changing a property of a node over time, either frame-by-frame (*AnimatedSprite2D*) or through keyframes (*AnimationPlayer*). (Ch. 10)

AnimationPlayer. A node that records and plays changes to any property of any node in a scene. (Ch. 10)

Area2D. A node that detects when other bodies or areas enter or leave its shape. It does not block movement. Used for pickups, hazards, and triggers. (Ch. 8)

Argument. The actual value passed to a *function* when you call it. See also *parameter*. (Ch. 4)

Array. An ordered list of values, with positions counted from 0. (Ch. 4)

Asset. Any file used by a game that is not code: images, sounds, music, fonts, models. (Ch. 6)

Assignment. Storing a value in a *variable* using `=`. (Ch. 3)

Atlas. One big image that contains many smaller pictures (frames, tiles), used for efficiency. (Ch. 12)

Audio bus. A channel that sounds are sent to, where volume and effects are applied. (Ch. 11)

Autoload. A script or scene that Godot loads automatically at startup and keeps alive across scene changes. Also called a *singleton*. (Ch. 16)

B

Body. A physics object that has a shape and can collide. In 2D: *StaticBody2D*, *RigidBody2D*, *CharacterBody2D*. (Ch. 8)

Bool (Boolean). A type that is either `true` or `false`. (Ch. 3)

Branch. A node and all its descendants in the tree. Also: a separate line of work in *Git*. (Ch. 2, 22)

Breakpoint. A marker on a line of code that pauses the game there so you can inspect it. (Ch. 22)

Bug. A mistake in a program that makes it behave in a way you did not intend. (Ch. 3)

Build. To produce a runnable program or game from a project. (Ch. 23)

Button. A *Control* that the player can click or press. (Ch. 14)

C

Call. To ask a *function* to run, by writing its name with parentheses. (Ch. 4)

Callable. A value that refers to a function, so it can be stored or passed around. (Ch. 15)

Camera2D. A node that decides which part of a 2D world is shown on screen. (Ch. 6)

Canvas. The surface that 2D nodes and UI draw on. (Ch. 6)

CanvasLayer. A node that draws its children on a separate layer that ignores the camera, used for UI and backgrounds. (Ch. 14)

CharacterBody2D. A node for a character that you move with code and that slides along solid things. (Ch. 8)

Child. A node placed directly beneath another node (its *parent*) in the tree. (Ch. 2)

Class. A blueprint that describes what an object is and does. Every node type is a class. (Ch. 5)

class_name. A line at the top of a script that gives the script a global name. (Ch. 5)

Collision. Two physics objects touching or overlapping. (Ch. 8)

Collision layer. A setting that says which numbered group(s) a body belongs to. (Ch. 8)

Collision mask. A setting that says which numbered group(s) a body looks for. (Ch. 8)

CollisionShape2D. A node that gives a body or area its shape. Needs a *Shape2D* resource. (Ch. 8)

Color. A value of red, green, blue, and *alpha*, each from 0 to 1. (Ch. 6)

ColorRect. A *Control* that draws a solid rectangle of one color. (Ch. 6)

Comment. Text in code, starting with `#`, that the computer ignores. (Ch. 3)

Comparison operator. `==`, `!=`, `<`, `>`, `<=`, `>=`: gives `true` or `false`. (Ch. 4)

Compile. Turn code into something a computer runs. GDScript is read by Godot as the game runs. (Ch. 3)

ConfigFile. A built-in class that saves settings in a simple text file of sections and keys. (Ch. 17)

Constant. A named value that never changes while the game runs, declared with `const`. (Ch. 3)

Container. A *Control* that arranges its children automatically (for example `HBoxContainer`). (Ch. 14)

Control. The base type of all user-interface nodes. (Ch. 14)

Coordinates. A pair of numbers (x, y) that say where something is. (Ch. 6, App. G)

CPU. The main processor of a computer. It runs your scripts. (Ch. 1)

CPUParticles2D. A particle node that does its work on the CPU. (Ch. 20)

D

Debugger. The part of the editor that shows errors, lets you pause and step through code, and inspects a running game. (Ch. 22)

Delta. The time, in seconds, since the last frame. Passed to `_process` and `_physics_process`. (Ch. 7)

Dictionary. A collection that stores values under names (keys). (Ch. 4)

Dock. A panel of the editor, such as the Scene dock or the Inspector. (Ch. 1)

Dot product. A number that says how much two vectors point the same way. (App. G)

Draw call. One request from the engine to the graphics hardware to draw something. (Ch. 22)

E

Editor. The Godot program you use to build your game. (Ch. 1)

Emit. To announce a *signal*. (Ch. 5)

Engine. The software that provides ready-made parts (drawing, physics, sound, input) so you do not write them yourself. (Ch. 1)

Enum. A list of named choices, such as `IDLE`, `WALK`, `JUMP`, stored as numbers. (Ch. 4)

Event. Something that happens, such as a key press or a mouse click. (Ch. 7)

Export (game). To package your game as a program others can run. (Ch. 23)

Export (script). To make a variable show up in the Inspector using `@export`. (Ch. 5)

Export preset. Saved settings for exporting to one platform. (Ch. 23)

Export template. The prebuilt game engine runtime that gets bundled into an exported game. (Ch. 23)

F

FileAccess. A built-in class for reading and writing files. (Ch. 17)

FileSystem dock. The editor panel that shows your project's files. (Ch. 1)

Float. A number with a decimal point, such as `2.5`. (Ch. 3)

Font. A style of lettering. (Ch. 14)

Frame. One picture shown on screen. A game shows many per second. (Ch. 1)

Frame rate (FPS). How many frames are shown each second. (Ch. 7)

Function. A named, reusable group of instructions. (Ch. 4)

G

Game loop. The cycle a game repeats: read input, update the world, draw. (Ch. 1)

GDScript. Godot's own programming language. (Ch. 3)

Git. A program that records the history of your project's files. (Ch. 22)

Gizmo. The handles drawn on a selected object in the viewport for moving, rotating, and scaling. (Ch. 2)

Global position. A node's position in the world, rather than relative to its parent. (Ch. 6)

GPU. The graphics processor, which draws images. (Ch. 1)

GPUParticles2D. A particle node that does its work on the graphics processor. (Ch. 20)

Gravity. A constant downward pull, added to velocity each frame in a platformer. (Ch. 13)

Group. A tag you can put on nodes so that you can find or command them all at once. (Ch. 5)

H

Hitbox. The area where something can be hit. (Ch. 8)

HUD. "Heads-up display": the score, health, and other information drawn over the game. (Ch. 14)

I

Identifier. The name of a variable, function, or class. (Ch. 3)

If statement. Code that runs only when a condition is true. (Ch. 4)

Import. To bring a file into the project, or bring in an existing project. (Ch. 1)

Indentation. Spaces or tabs at the start of a line. In GDScript it shows which lines belong together. (Ch. 3)

Index. The position of an item in an array, starting at 0. (Ch. 4)

Inheritance. One class getting the abilities of another. `extends` expresses it. (Ch. 5)

Input Map. The list of named actions and the keys or buttons that trigger them. (Ch. 7)

Inspector. The editor panel that shows and edits the properties of the selected node. (Ch. 1)

Instance. One actual copy made from a blueprint (a class or scene). (Ch. 2)

Instantiate. To make an instance of a scene while the game runs. (Ch. 9)

Int (integer). A whole number. (Ch. 3)

Interpolation. Finding values between two known values. See *lerp*. (App. G)

J

Jitter. Visible shaking of a moving object caused by timing mismatches. (Ch. 22)

JSON. A text format for storing data as lists and key-value pairs, readable by many programs. (Ch. 17)

K

Key (dictionary). The name under which a value is stored in a dictionary. (Ch. 4)

Keyframe. A point in an animation that records a value at a time. (Ch. 10)

L

Label. A *Control* that shows text. (Ch. 14)

Layer (physics). See *collision layer*. (Ch. 8)

Lerp. "Linear interpolation": a value part of the way between two others. (App. G)

Local position. A node's position relative to its parent. (Ch. 6)

Loop. Code that repeats. `for` and `while` are loops. (Ch. 4)

M

Main scene. The scene Godot runs when you press Play. (Ch. 1)

Margin. Empty space around the edge of something, especially in UI. (Ch. 14)

Material. Describes how a surface looks (color, shininess, shader). (Ch. 19)

Mesh. The shape of a 3D object, made of triangles. (Ch. 19)

MeshInstance3D. A node that shows a mesh in 3D. (Ch. 19)

Method. A function that belongs to a class or object. (Ch. 5)

Modulate. A property that tints a node and its children by a color. (Ch. 10)

Move and slide. The `CharacterBody2D` function that moves the body and slides along surfaces. (Ch. 8)

N

NavigationAgent2D. A node that finds and follows a path around obstacles. (Ch. 21)

Node. The basic building block of a Godot game. Every node has one job. (Ch. 2)

Node2D. The base of all 2D nodes that have a position, rotation, and scale. (Ch. 6)

Node3D. The base of all 3D nodes. (Ch. 19)

NodePath. The address of a node in the tree. (Ch. 5)

Normal (vector). See *unit vector*. (App. G)

Normalize. Change a vector to length 1, keeping its direction. (App. G)

Null. The value meaning "nothing here". (Ch. 5)

O

Object. A thing in your program that has data and abilities. (Ch. 5)

Operator. A symbol that does something with values, such as `+` or `==`. (Ch. 3)

Origin. The point (0, 0) of a coordinate system. (App. G)

Output panel. The editor panel that shows what `print()` writes. (Ch. 3)

Override. To write your own version of a function that a parent class already has. (Ch. 5)

P

PackedScene. A saved scene, ready to be copied into the game. (Ch. 9)

Parameter. A named slot in a function definition that receives an *argument*. (Ch. 4)

Parent. The node directly above a node in the tree. (Ch. 2)

Particle. A tiny, short-lived graphic created in large numbers for effects like sparks and smoke. (Ch. 20)

Path. The location of a file (`res://player.gd`) or node (`Player/Sprite2D`). (Ch. 2)

Pause. Stopping gameplay while keeping the game open. (Ch. 18)

Physics body. See *body*. (Ch. 8)

Physics process. The function `_physics_process`, called at a steady rate for movement and physics. (Ch. 7)

Pivot. The point an object rotates or scales around. (Ch. 6)

Pixel. The smallest dot of a picture. (Ch. 6)

Pixel art. Art drawn with deliberately visible pixels. (Ch. 6)

Polygon2D. A node that draws a filled shape from a list of points. (Ch. 6)

Preload. Load a file when the script is loaded. (Ch. 9)

Process. The function `_process`, called once every frame. (Ch. 5)

Project. The folder of everything that makes up your game. (Ch. 1)

Project Manager. The first window of Godot, which lists and creates projects. (Ch. 1)

Project Settings. The window where game-wide options live. (Ch. 1)

Property. A named value belonging to a node, such as `position` or `text`. (Ch. 2)

Q

Queue free. `queue_free()`: remove a node safely at the end of the frame. (Ch. 9)

R

Radian. A unit for angles. A full turn is `TAU` (about 6.28). (App. G)

Random. Unpredictable (within rules) values from functions like `randi()`. (Ch. 9)

RayCast2D. A node that shoots an invisible line to see what it hits. (Ch. 21)

Ready. `_ready()`: runs once when a node and its children have entered the tree. (Ch. 5)

Rect2. A rectangle given by a position and a size. (Ch. 6)

Reference. A value that points to an object rather than holding a copy of it. (Ch. 5)

Render. To draw the picture the player sees. (Ch. 6)

Renderer. The part of the engine that draws. Godot 4 has Forward+, Mobile, and Compatibility. (Ch. 1)

Repository (repo). A project folder tracked by Git. (Ch. 22)

res://. The path prefix for files inside your project. (Ch. 2)

Resource. Saved data a node can use, such as a texture, font, or shape. (Ch. 16)

RichTextLabel. A *Control* that shows text with formatting. (Ch. 14)

RigidBody2D. A body fully controlled by the physics engine. (Ch. 8)

Root node. The top node of a scene. (Ch. 2)

S

Scale. How much a node is stretched. 1 is normal size. (Ch. 6)

Scene. A saved tree of nodes that can be reused. (Ch. 2)

Scene tree. The tree of all nodes currently running in the game. (Ch. 2)

Script. A text file of code attached to a node. (Ch. 3)

Setter / getter. Functions that run when a property is written or read. (Ch. 16)

Shader. A small program that runs on the graphics processor to decide how things look. (Ch. 20)

Shape2D. A resource that describes a collision shape, such as a rectangle or circle. (Ch. 8)

Signal. A message a node sends when something happens. (Ch. 5)

Singleton. An object of which there is exactly one. See *autoload*. (Ch. 16)

Sprite. A 2D image that moves around a game. (Ch. 6)

Sprite2D. The node that shows a sprite. (Ch. 6)

SpriteFrames. A resource that holds the animations and frames for an *AnimatedSprite2D*. (Ch. 10)

State machine. A design where something is always in exactly one state (idle, chase, attack) with rules for changing. (Ch. 21)

Static typing. Writing the type of a variable or function so Godot can check it. (Ch. 3)

StaticBody2D. A body that does not move, such as a wall. (Ch. 8)

String. Text. (Ch. 3)

Subtree. See *branch*. (Ch. 2)

Syntax. The spelling and grammar rules of a language. (Ch. 3)

Syntax error. A mistake in syntax that stops the script from loading. (Ch. 3)

T

Texture. An image used for drawing. (Ch. 6)

TextureRect. A *Control* that shows an image. (Ch. 14)

Theme. A set of styles (colors, fonts, spacing) applied to UI. (Ch. 14)

Tile. A small picture that is repeated to build a level. (Ch. 12)

TileMapLayer. A node that paints a grid of tiles. (Ch. 12)

TileSet. A resource that holds the tiles that a *TileMapLayer* can paint. (Ch. 12)

Timer. A node that waits and then emits `timeout`. (Ch. 9)

Transform. A node's position, rotation, and scale together. (Ch. 6)

Tween. A temporary object that smoothly changes a property from one value to another. (Ch. 10)

Type. The kind of value something is (int, float, String, and so on). (Ch. 3)

Type inference. Letting Godot work out a type from the value, using `:=`. (Ch. 3)

U

UI (user interface). The buttons, text, and menus the player uses. (Ch. 14)

Unit vector. A vector of length 1. (App. G)

user://. The path prefix for the folder where your game saves player data. (Ch. 17)

V

Variable. A named box that holds a value. (Ch. 3)

Variant. The type that can hold any value; used when no type is declared. (Ch. 3)

VBoxContainer / HBoxContainer. Containers that stack children vertically or horizontally. (Ch. 14)

Vector2. A pair of numbers (x, y), used for positions, directions, and sizes. (Ch. 6, App. G)

Vector3. Three numbers (x, y, z) for 3D. (Ch. 19)

Velocity. How fast, and in which direction, something moves. (Ch. 7)

Version control. A system that records changes to files over time. See *Git*. (Ch. 22)

Viewport. The area where the scene is shown, in the editor or in the running game. (Ch. 1)

Visible. A property that decides whether a node is drawn. (Ch. 6)

Void. The return type of a function that gives back nothing. (Ch. 4)

W

While loop. A loop that repeats as long as a condition is true. (Ch. 4)

World. The collection of everything in a game, used for the game space. (Ch. 6)

X, Y, Z

X axis / Y axis. The two directions of the 2D plane. In Godot 2D, y grows downward. (App. G)

Y-sort. Drawing nodes in order of their y position so nearer ones appear on top. (Ch. 12)

Z index. A number that decides which 2D node draws on top of another. (Ch. 6)

Appendix F: Free Assets and Tools

Every project in this book can be built without downloading any art or sound. Shapes, colors, and generated tones are enough to learn everything. When you want your own games to look and sound better, this appendix points to well-known places to find free materials and free tools.

Website addresses and prices change. The sites below have been around for years, but always check the current page for each file before you use it.

F.1 Licenses in plain words

An asset's **license** tells you what you may do with it. **Always read the license of every file you use.** The most common ones you will meet:

License	In plain words	What you must do
CC0 (Creative Commons Zero, "public domain dedication")	The creator gives up their rights as far as the law allows.	Nothing. Crediting is kind but not required. Commercial use is allowed.
CC-BY (Attribution)	You may use it, including commercially, but you must give credit.	Credit the creator in your game or its page, say what license it is, and say if you changed it.
CC-BY-SA (Attribution, ShareAlike)	Like CC-BY, but anything you make that is based on it must be shared under the same license.	Credit, and release derived *assets* under the same license. Check how it applies to your whole game before using.
CC-BY-NC (NonCommercial)	Free to use, but **not** for commercial projects.	Credit, and do not sell or monetize your game.
OGA-BY	OpenGameArt's own variant of attribution, designed for games.	Credit the creator.
GPL and similar	Strong "copyleft": works that include it may need to be released as open source.	Read carefully before using in a game you do not want to open-source.
Custom / "free to use"	The author's own rules.	Read their terms.
MIT, Apache 2.0	Permissive licenses for code and tools.	Keep the copyright notice and license text.

A good habit: when you use an asset, immediately add a line to a `CREDITS.txt` file in your project with the name of the asset, the author, the link, and the license. You will thank yourself later.

> **Note:** The safest choice for beginners is **CC0** assets, because there is nothing to track. The next best is **CC-BY**, because you only need to credit.
>
> **Warning:** "Free to download" does not mean "free to use however you like". A file on a website without a clear license is not safe to use in a game you share. This is general information, not legal advice. When in doubt, pick a different asset or ask the creator.

F.2 Free art, sound, and music libraries

Kenney (kenney.nl)

A large library of assets made by Kenney Vleugels, nearly all released under **CC0**. It includes 2D sprites, tilesets, UI packs, icon packs, fonts, sound effects, and 3D models. It is the best place for a beginner to start, because everything is consistent in style and has no credit requirement. The individual asset page states the license; confirm it on the asset page when you download.

OpenGameArt.org (opengameart.org)

A huge community site for game art, sound, and music, run by volunteers. Licenses vary by file (CC0, CC-BY, CC-BY-SA, GPL, OGA-BY, and others). The site has a filter for license type, so you can search for CC0 only. Always check the license listed on each page.

itch.io asset pages (itch.io)

The game-publishing site also hosts thousands of asset packs, many free. The license is chosen by each creator and is stated on their page, and it varies widely, so read each one. Sort by free and look for tags like "CC0" or "royalty-free". The same site is where you will publish your own game in Chapter 24.

Freesound (freesound.org)

A database of sound effects and recordings shared by the community. Licenses vary per sound (CC0, CC-BY, and CC-BY-NC are common). You need a free account to download. Filter by license to avoid NonCommercial sounds if you plan to sell your game.

Google Fonts (fonts.google.com)

Hundreds of fonts, most under the SIL Open Font License, which allows use in games and apps. Godot can import `.ttf` and `.otf` font files.

Other well-known sources

- **Poly Haven** (polyhaven.com): free **CC0** 3D models, textures, and HDRIs. Useful for Chapter 19 and beyond.
- **ambientCG** (ambientcg.com): free **CC0** materials and textures.
- **Incompetech** (incompetech.com): music by Kevin MacLeod, free with credit (CC-BY) or for purchase without credit. Check the current terms.
- **Free Music Archive**: music with many license types; filter before downloading.
- **Quaternius** (quaternius.com): low-polygon 3D models, with generous free licenses; check the page.

F.3 Free tools for making your own art, sound, and music

You do not need all of these. Pick one for each job when you feel the need.

Pixel art and drawing

Tool	Good for	Cost
LibreSprite	Pixel art and sprite animation. A free, open-source tool based on an older version of Aseprite.	Free
Aseprite	The most popular pixel art and animation tool. The source code can be built for free, or you can buy a ready-made copy.	Paid (or build it yourself)
Krita	Painting and drawing, including pixel art and animation.	Free, open source
GIMP	Photo and image editing.	Free, open source
Inkscape	Vector drawing (it makes SVG files, which Godot can import).	Free, open source
Piskel (piskelapp.com)	Simple pixel art in the browser.	Free
Pixelorama	A pixel-art editor made with Godot itself.	Free, open source

Sound and music

Tool	Good for	Cost
Audacity	Recording and editing sounds, trimming, volume, effects.	Free, open source
LMMS	Making music with instruments, patterns, and a piano roll.	Free, open source
Bfxr / jsfxr (sfxr.me)	Generating retro sound effects (jumps, coins, explosions) with one click.	Free
BeepBox (beepbox.co)	Writing short chiptune songs in the browser.	Free
Cakewalk by BandLab	A full music studio for Windows.	Free

3D

Tool	Good for	Cost
Blender	Modeling, animating, and exporting 3D for Godot. Godot imports its `.blend` files and glTF exports.	Free, open source
Kenney and **Quaternius**	Ready-made 3D models.	Free

Maps and level design

Tool	Good for	Cost
Tiled (mapeditor.org)	A popular external tile map editor. Godot's own TileMapLayer editor is enough for this book.	Free, open source

Project tools

Tool	Good for	Cost
Git	Version control, covered in Chapter 22.	Free
GitHub / GitLab	Online home for your repository and backups.	Free tiers
Visual Studio Code	An optional external code editor. Godot's built-in script editor is enough for this book.	Free

F.4 Getting assets into Godot

1. Download the pack and unzip it. Unzipping gives you folders of `.png`, `.ogg`, `.wav`, or `.glb` files.
2. Copy the files you want into your project folder (for example `res://assets/art/`). You can do this in Windows Explorer or Finder; Godot imports them automatically the next time it has focus. Keep assets in your own folder, not inside the Godot installation.
3. In the **FileSystem** dock, click the file to see its import options in the **Import** dock. For pixel art, change the texture filter to *Nearest* (in Project Settings under *Rendering* > *Textures*, or on the node under *CanvasItem* > *Texture* > *Filter*).
4. Drag the image onto a `Sprite2D`'s Texture property, or drag a sound onto an `AudioStreamPlayer`'s Stream property.

Common file formats Godot reads: images `.png`, `.jpg`, `.svg`, `.webp`; audio `.ogg`, `.wav`, `.mp3`; 3D `.glb`, `.gltf`, `.blend`, `.obj`; fonts `.ttf`, `.otf`, `.woff`. For short sound effects, `.wav` or `.ogg` are the usual choices. For music, `.ogg` keeps files small.

F.5 Crediting assets in your game

Add a "Credits" screen or a text file next to your game. A simple format:

```
Pip hero sprite - "Tiny Heroes" by Example Author - CC-BY 4.0 - link to the page
Coin sound - "coin pickup" by Example Author - CC0
Music - "Example Song" by Example Author - CC-BY 4.0
Made with Godot Engine (MIT License) - godotengine.org
```

Where possible, include the exact title, the author's name, the link, and the license. Chapter 24 shows how to mention these on your itch.io page.

F.6 Learning resources

Beyond this book:

- **The official Godot documentation** (docs.godotengine.org): the manual, the full class reference, and a "Getting started" section. The class reference is also built into the editor: press **F1** to search it.
- **Godot's official demo projects**: small example projects maintained by the Godot team in the project's public repositories. They show many features in a few files each.
- **The Godot community**: the official forum and community chat servers are friendly to beginners. When asking a question, include your Godot version, the error text, and the code involved.
- **itch.io game jams** (itch.io/jams): short contests where you make a game in a weekend. A fine way to practice finishing small games.

F.7 A small starting kit

If you want just one recommendation for each job, this kit is free and widely used:

Job	Tool or source
Starting art	Kenney asset packs (CC0)
Sound effects	sfxr-style generator, or Freesound filtered by CC0

Job	Tool or source
Music	LMMS for your own, or CC0/CC-BY tracks
Drawing pixel art	LibreSprite, Piskel, or Krita
Editing audio	Audacity
Version control	Git
Publishing	itch.io

Appendix G: Math for Games, Just Enough

You do not need to be good at math to make games. You need about six ideas, and most of them are pictures more than numbers. This appendix explains those six ideas with drawings first and code second. Each section ends with the Godot function that does the work for you, so you rarely have to calculate anything by hand.

The six ideas are: coordinates, vectors, normalizing, the dot product, angles in radians, and lerp.

G.1 Coordinates: where things are

A 2D game is a flat sheet of graph paper. Every position on it is described by two numbers: **x** (how far across) and **y** (how far down). Written as a pair, `(x, y)`, this is a **coordinate**. In Godot it is stored in a `Vector2`.

Two things surprise beginners:

1. The **origin** `(0, 0)` is the **top-left** corner of the screen, not the center or the bottom-left.
2. **y grows downward.** A bigger y means lower on the screen. This is the opposite of the graph paper you used in school.

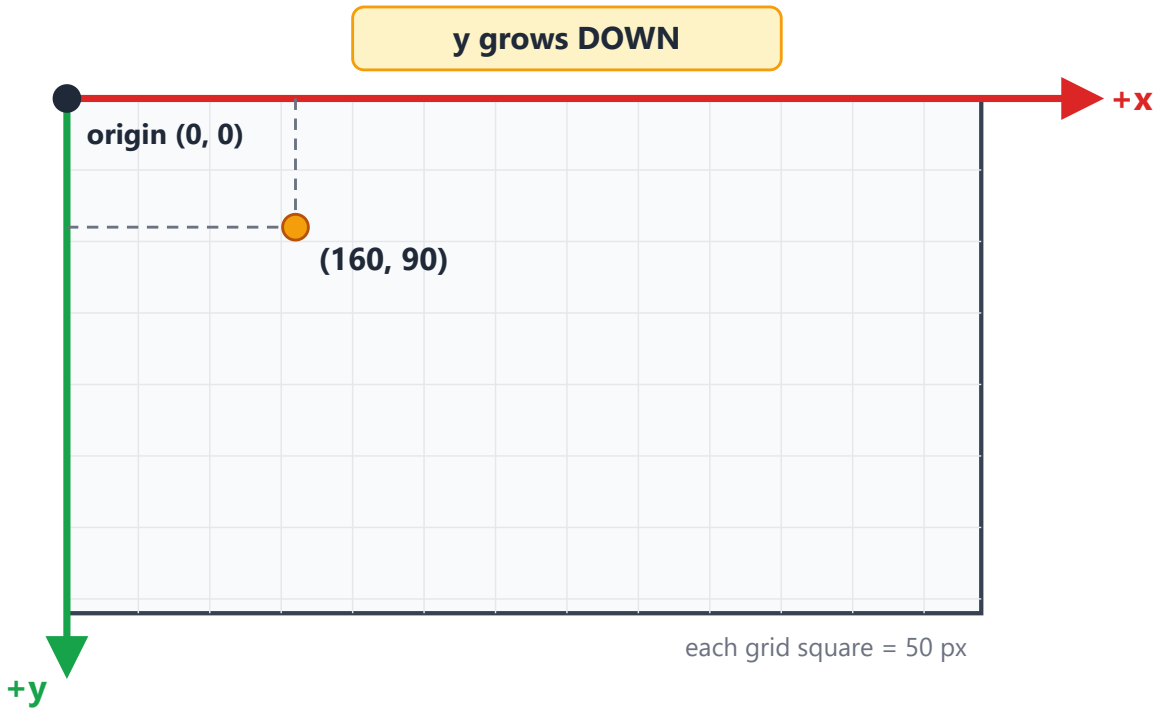


Figure G.1: x grows to the right, y grows downward. Moving "up" means a negative y.

So to move a sprite up by 10 pixels you subtract: `position.y -= 10`. That is why jumping code sets `velocity.y` to a negative number.

A node's `position` is measured from its **parent's** origin. Its `global_position` is measured from the world's origin. If a child node sits at `position = (10, 0)` inside a parent at `(100, 50)`, the child's `global_position` is `(110, 50)`.

```gdscript
extends Node2D

func _ready() -> void:
    position = Vector2(100, 50)        # relative to the parent
    print(global_position)             # where it really is in the world
```

Try it: Add a `Sprite2D` with the Godot icon to a scene. In the Inspector, type `0` and `0` into its position. Then try `200` and `0`, then `0` and `200`. Which way does it move each time?

In 3D, a third number, **z**, is added. In Godot 3D, **y points up** (the opposite of 2D), and the forward direction of a camera or object is **negative z**. Chapter 19 covers this.

G.2 Vectors: arrows with a length

A **vector** is a pair of numbers that you can think of in two ways: as a point (a place) or as an arrow (a movement). The vector `(3, 4)` can mean "the point 3 across and 4 down" or "move 3 to the right and 4 down".

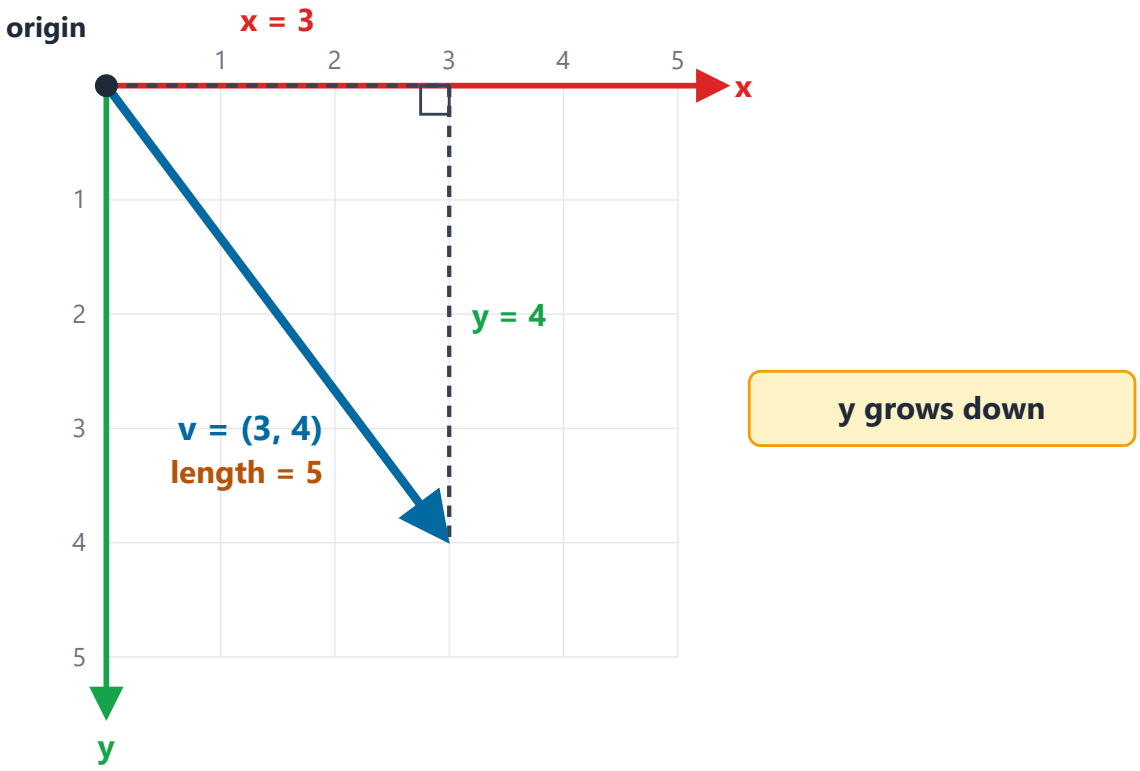


Figure G.2: The vector (3, 4). Its two components form the legs of a right triangle. Its length is the long side.

Things you do with vectors all the time:

Idea	Code	Meaning
Add	`a + b`	Do one movement, then the other
Subtract	`b - a`	The arrow pointing **from a to b**
Scale	`v * 3.0`	Make the arrow three times longer
Length	`v.length()`	How long the arrow is

Idea	Code	Meaning
Distance	`a.distance_to(b)`	How far apart two points are

The most useful of these is the subtraction. "From a to b" is `b - a`. Memorize the order: **destination minus start**.

```
extends Node2D

@export var speed: float = 200.0
@onready var target: Node2D = $Target

func _process(delta: float) -> void:
    var to_target: Vector2 = target.global_position - global_position   # arrow from me to the target
    print(to_target.length())                                            # how far away it is
```

The length of `(3, 4)` is 5. That comes from the Pythagorean theorem (the square of the long side equals the sum of the squares of the two short sides: 3 times 3 plus 4 times 4 is 25, and the square root of 25 is 5). You will never need to do this yourself, because `length()` does it. If you only need to *compare* distances, use `distance_squared_to()`, which skips the square root and is faster:

```
func close_enough(a: Vector2, b: Vector2) -> bool:
    return a.distance_squared_to(b) < 10.0 * 10.0     # closer than 10 pixels?
```

G.3 Normalizing: keeping direction, forgetting length

Often you only want the **direction** of an arrow, not how long it is. For example: "move toward the target at 200 pixels per second". If you add the whole `to_target` arrow, the object moves quickly when far away and slowly when close. You want a fixed speed.

The fix is to **normalize** the vector: shrink or stretch it until its length is exactly 1, without changing the direction it points. A vector of length 1 is called a **unit vector**.

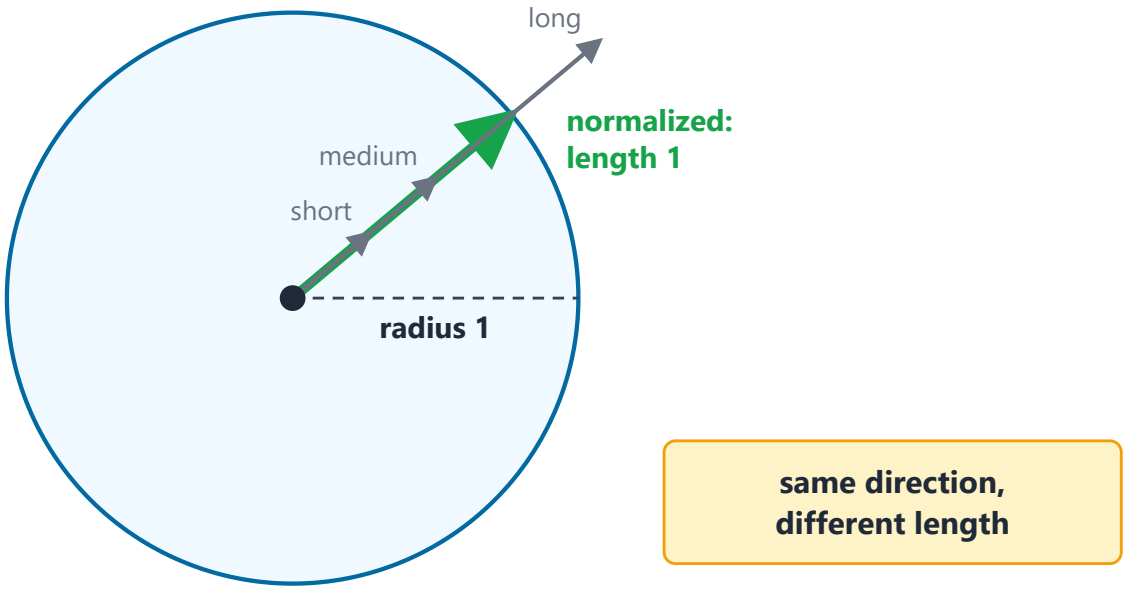


Figure G.3: Normalizing keeps the direction and sets the length to 1. All unit vectors end on a circle of radius 1.

Then multiply the unit vector by the speed you want:

```
extends Node2D

@export var speed: float = 200.0
@onready var target: Node2D = $Target

func _process(delta: float) -> void:
    var to_target: Vector2 = target.global_position - global_position
    var direction: Vector2 = to_target.normalized()      # length is now 1
    global_position += direction * speed * delta         # 200 pixels per second
```

`direction * speed * delta` reads as "which way, how fast, for how long this frame". This is the standard recipe for movement, and Chapter 7 explains why `delta` is there.

> **Warning:** Normalizing the zero vector `(0, 0)` has no direction to keep. Godot returns `(0, 0)` rather than crashing, so the object simply stays still. This is usually what you want, but be aware of it.

Normalizing also fixes the "diagonal is faster" bug. If you add "right 1" and "down 1", the result `(1, 1)` has length about 1.41, so diagonal movement would be 41% faster. `Input.get_vector()` returns an already-normalized result for you.

G.4 The dot product: how much do two arrows agree?

The **dot product** takes two vectors and gives back **one number**. For two unit vectors, that number tells you how much they point the same way:

Result	Meaning
`1`	Pointing in exactly the same direction
`0`	At a right angle (perpendicular)
`-1`	Pointing in exactly opposite directions
between	Somewhere in between

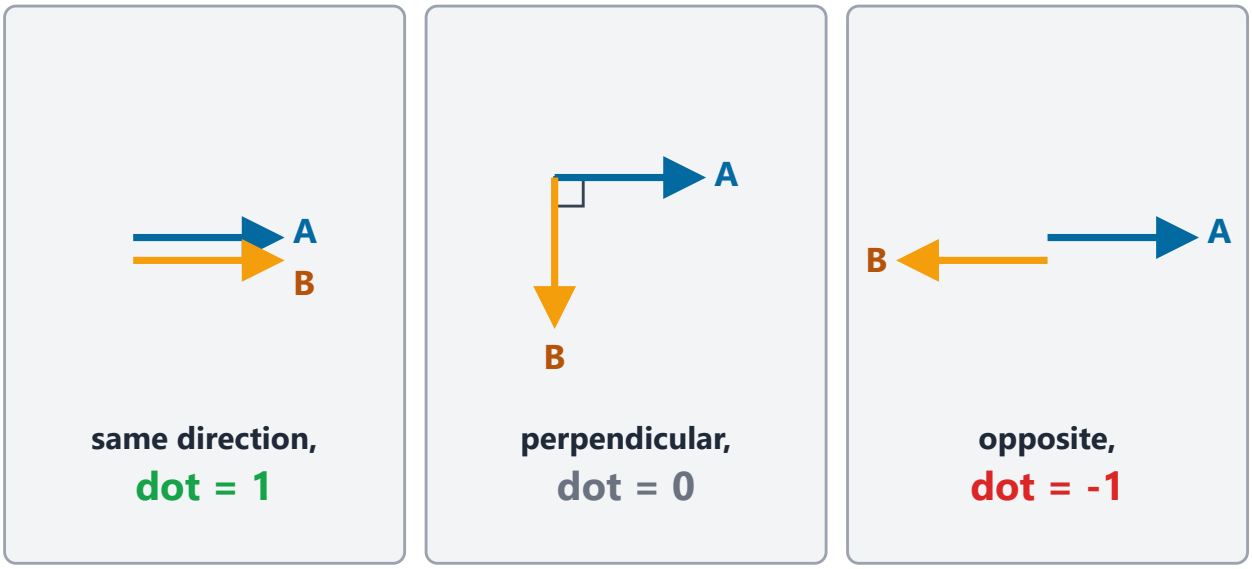


Figure G.4: The dot product of two unit vectors is 1, 0, or -1 for the three pictured cases, and a smooth value in between for all other angles.

You do not need the formula to use it. The intuition is enough: **positive means "mostly the same way", negative means "mostly opposite"**. In Godot: `a.dot(b)`. Both vectors should be normalized for the table above to hold.

A classic use is "is the enemy in front of me?":

```
func in_front_of(a: Node2D, b: Node2D) -> bool:
    var facing: Vector2 = Vector2.RIGHT.rotated(a.rotation)         # which way a is pointing
    var to_b: Vector2 = (b.global_position - a.global_position).normalized()
    return facing.dot(to_b) > 0.0                                    # positive means in front
```

To limit the check to a cone, such as a 90 degree field of view (45 degrees to each side), compare against the cosine of the half-angle:

```
func fov_check(a: Node2D, b: Node2D) -> bool:
    var facing: Vector2 = Vector2.RIGHT.rotated(a.rotation)
    var to_b: Vector2 = (b.global_position - a.global_position).normalized()
    return facing.dot(to_b) > cos(deg_to_rad(45.0))
```

Other uses: checking whether a character is moving toward or away from something, and lighting (surfaces facing the light are brighter).

G.5 Angles and radians

You know angles in **degrees**: a full turn is 360. Godot, like most programming languages, measures angles in **radians**: a full turn is `TAU`, which is about 6.28. Half a turn is `PI` (about 3.14), a quarter turn is `PI / 2`.

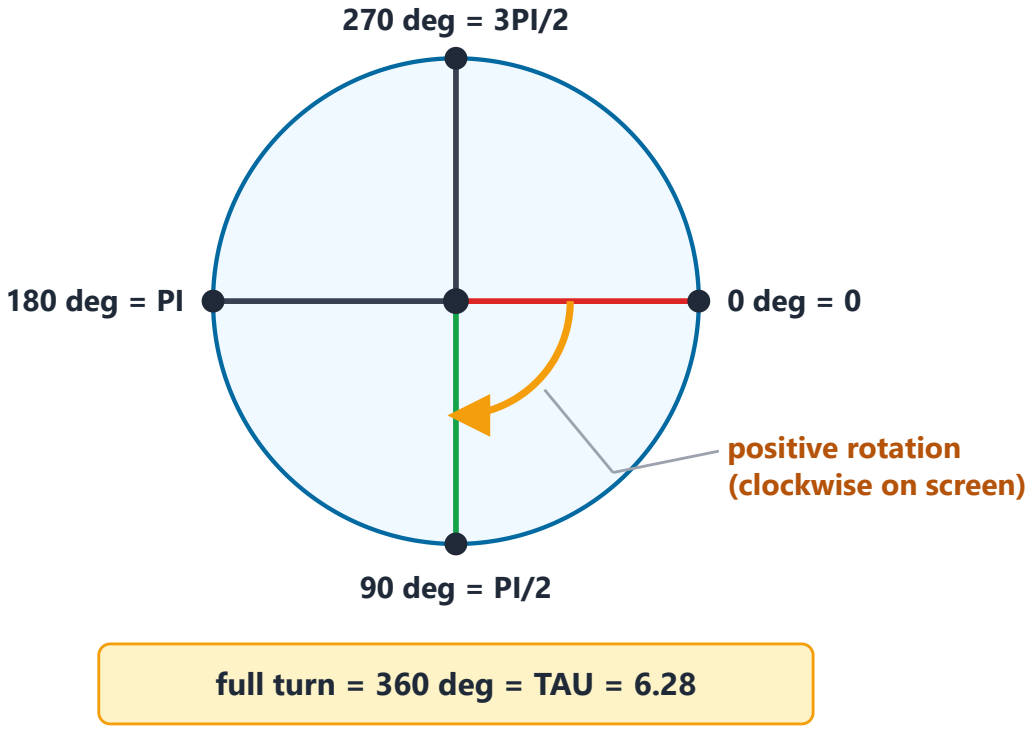


Figure G.5: The same angles in degrees and in radians. On screen, because y points down, positive rotation turns clockwise.

Degrees	Radians	Written in code
0	0	`0.0`
45	about 0.785	`PI / 4`
90	about 1.571	`PI / 2`
180	about 3.142	`PI`
360	about 6.283	`TAU`

You rarely need to convert by hand:

```
extends Node2D

func spin(delta: float) -> void:
    rotation += TAU * 0.25 * delta        # a quarter turn per second
    rotation_degrees = 90.0               # or set it in degrees directly
    var r: float = deg_to_rad(90.0)       # degrees to radians
    var d: float = rad_to_deg(PI)         # radians to degrees: 180.0
```

The `rotation` property of a `Node2D` is in radians. `rotation_degrees` is the same value in degrees, for convenience. Positive rotation turns **clockwise** on screen, because y points down.

Angles and directions

An angle and a direction are two ways of saying the same thing:

```
func circle_pos(center: Vector2, radius: float, angle_deg: float) -> Vector2:
    var a: float = deg_to_rad(angle_deg)
    return center + Vector2(cos(a), sin(a)) * radius      # a point on a circle
```

`cos(a)` is the x part and `sin(a)` is the y part of the unit vector at angle `a`. Going the other way, `vector.angle()` gives the angle of a vector in radians, so `look_at()`-style turning is one line:

```
func face(to_target: Vector2) -> void:
    rotation = to_target.angle()        # turn to face the target (sprite drawn facing right)
```

Rotating smoothly toward an angle uses `rotate_toward()` or `angle_difference()`, which take care of the jump at the full-turn point (where 359 degrees is right next to 1 degree):

```
func steer(delta: float) -> void:
    rotation = rotate_toward(rotation, 0.0, 2.0 * delta)    # turn toward angle 0 at 2 radians per second
```

G.6 Lerp: sliding between two values

Lerp (linear interpolation) finds a value part of the way between two others. `lerp(a, b, t)` returns `a` when `t` is `0`, returns `b` when `t` is `1`, and returns the value halfway when `t` is `0.5`.

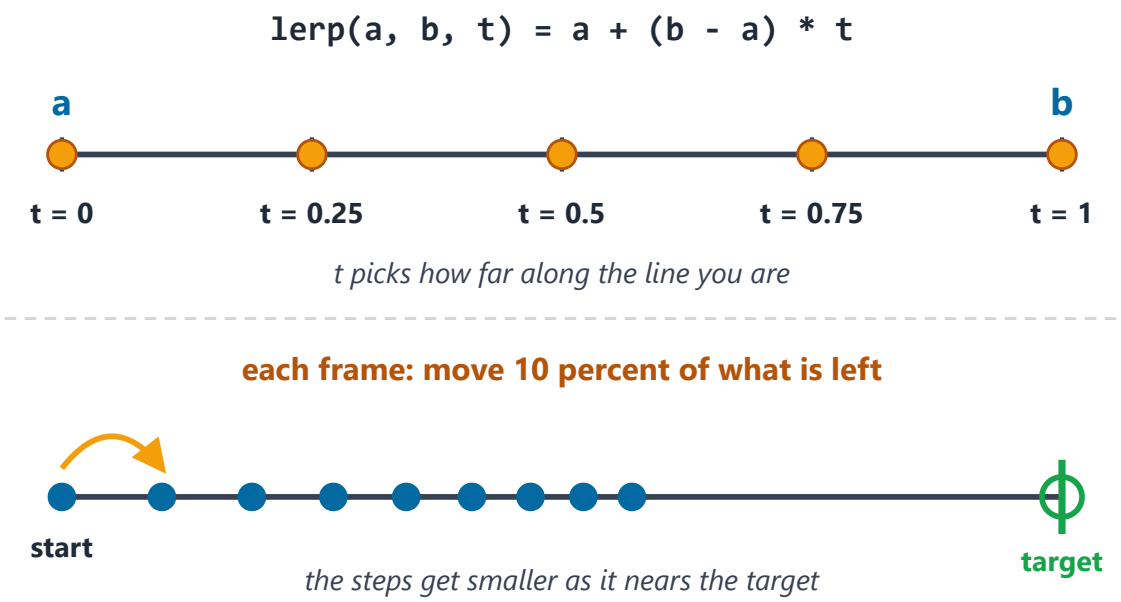


Figure G.6: Top: lerp picks a point a fraction t of the way from a to b. Bottom: lerping a little each frame makes an object slow down as it arrives (an "ease-out").

```
extends Node2D

func demo() -> void:
    print(lerp(0.0, 10.0, 0.5))                          # 5.0
    print(lerp(Vector2(0, 0), Vector2(100, 0), 0.25))    # (25, 0)
    print(Color.RED.lerp(Color.BLUE, 0.5))               # purple halfway
```

Lerp works on numbers, vectors, and colors.

Smooth follow

Lerping a little each frame produces a pleasant glide, because each step covers a fixed *fraction* of the remaining distance, so steps shrink as you near the target:

```
extends Node2D

@onready var target: Node2D = $Target

func _process(delta: float) -> void:
    global_position = global_position.lerp(target.global_position, 0.1)
```

The weakness: `0.1` per frame feels different at 30 and 144 frames per second. A frame-rate independent version uses `exp`:

```
extends Node2D

@onready var target: Node2D = $Target

func _process(delta: float) -> void:
    var t: float = 1.0 - exp(-10.0 * delta)       # bigger number = snappier
    global_position = global_position.lerp(target.global_position, t)
```

If you want a **fixed speed** instead of a glide, use `move_toward`, which never overshoots:

```
extends Node2D

func _process(delta: float) -> void:
    position = position.move_toward(Vector2.ZERO, 50.0 * delta)    # 50 pixels per second toward the
origin
```

Remapping a range

`remap()` converts a value from one range to another. It is perfect for health bars and meters:

```
func health_to_bar(health: float) -> float:
    return remap(health, 0.0, 100.0, 0.0, 1.0)     # 50 health becomes 0.5
```

Related helpers: `clamp()` keeps a value inside limits, `smoothstep()` makes an S-shaped fade, `wrapf()` and `posmod()` loop a value around a range (useful for angles and for counting around a list).

G.7 Putting it together

Here are the six ideas in one task: "a turret at the middle of the screen turns toward the mouse and fires only when the mouse is in front."

Step	Idea used
Find the mouse's position in the world	Coordinates
Make the arrow from the turret to the mouse (`mouse - turret`)	Vector subtraction
Get the angle of that arrow	Radians (`angle()`)
Turn the turret toward it, smoothly	Lerp-style helper (`rotate_toward`)

Step	Idea used
Spawn bullets that fly in the facing direction	Normalized direction times speed
Check "in front" before firing	Dot product

Every problem in the rest of the book that "looks like math" is a mix of these.

G.8 Quick cheat sheet

I want to…	Use
Point from A to B	`b - a`
Know how far apart A and B are	`a.distance_to(b)`
Move at a steady speed toward B	`(b - a).normalized() * speed * delta`
Get the angle toward B	`(b - a).angle()`
Turn to face B	`look_at(b)` (for `Node2D`)
Know if something is ahead	`facing.dot(direction_to_it) > 0`
Glide toward a target	`lerp` with a fraction each frame
Move at a fixed speed toward a target	`move_toward`
Convert degrees and radians	`deg_to_rad`, `rad_to_deg`
Keep a number inside limits	`clamp`