

NodeWorthy

A visual state machine for Godot 4, for any node. Version 0.1, 26 September 2026.

NodeWorthy © 2026 StateCraft Games. All rights reserved.

What comes with NodeWorthy

Everything on this page is in the box, working the moment NodeWorthy is switched on, and none of it needs a line of code. Each item says where this manual explains it.

293

actions and checks, in 34 groups, from moving and jumping to quests and saving (chapter 10)

19

nodes to add, such as health bars, hearts, a minimap, an inventory grid, a quest log and a dialogue box (10.35)

6

ready-made players that walk, run and jump from the start, in 2D and 3D (4.3)

8

ready-made things, an enemy, a chest, a villager, a landscape, your own models and more, plus 2 menus (4.3)

92

recipes, small scenes that each show one thing done well (chapter 9)

293

examples, one a click away from every action, ready to run (chapter 10)

7

whole games to play and take apart, from a first person shooter to a village with quests, five of them built step by step in chapter 8

3

books, this manual, the Action Reference with a page for every action and check, and Heroes, Things and Nodes with everything ready-made, one at a time

What you can make with it

- Players that run, jump, climb, crouch and dash (4.3)
- Enemies that patrol, spot you, call their friends, chase and fight (6.13)
- Fighting: health, hits, knockback, guns and bullet patterns (6.17)
- Items: pick up, wear, drink, drop, and chests full of loot (5.5)
- Quests that tick themselves off, with a quest log and marks on the map (6.16)
- Dialogue drawn as a graph, with characters, portraits and answers (6.15)
- Health, mana, experience and levels (6.5), (6.18)
- Tables of weapons, enemies or anything else your game has (6.20)
- Saving and loading, and the player carried from level to level (10.20), (6.19)

What helps you build it

- A whole level in one window, blank or with a player, enemies, loot, a villager, the screen and menus, ready to play (5.1)
- Behaviour you draw, with every state and every way out of it in one graph (4.4)
- Everything added from one right-click, only what suits what you clicked (4.5)
- Every setting in Godot's own Inspector, picked from a list rather than typed (2.3)
- Watch a machine while the game runs, then pause it, step it or stop at a breakpoint (7.1)
- A Problems button that finds what is broken and says how to fix it (7.2)
- Step-by-step walkthroughs for items, quests, loot, data and dialogue (chapter 5)
- 2D and 3D from the same actions, with nothing to look for twice (chapter 1)

- Cameras, menus, a pause screen, and finding a way round walls (10.16), (10.21)
- 3D Terrain, landscapes with hills to walk over and grass that sways (6.21)
- Scripts beside it whenever you want them, never instead of it (chapter 13)

Mix and match. NodeWorthy does not replace scripting, it sits beside it. You can build a whole game without writing a line of code, but you never have to choose. One node can run a NodeWorthy template while the next runs an ordinary Godot script, and a script and a template can talk to each other. Use whichever is easier for each part of your game, and change your mind later. Chapter 13 shows how, for anybody who writes code.

Never change anything inside the `addons/nodeworthy` folder. That folder, and `addons/nodeworthy_editor` beside it, are NodeWorthy itself. When NodeWorthy is updated, both are replaced as a whole, and **anything you changed, added or saved inside them is lost, without a warning**. Do not edit, add, move, rename or delete anything in them.

That includes the ready-made templates, the recipes and the games that come with NodeWorthy. Look at them, run them and learn from them as much as you like, but to change one, make your own copy first. The NodeWorthy panel shows them as they came, and its **Make It Mine** button puts a copy among your own templates in one click, ready to change. Your own scenes, templates and scripts belong anywhere else in your project, never under `addons`.

Contents

What comes with NodeWorthy

1. What NodeWorthy is, in one page
 - 1.1 If you have the NodeWorthy app
2. Before you start, Godot in ten minutes
 - 2.1 The editor window
 - 2.2 Scenes and nodes
 - 2.3 The Inspector and a node's settings
 - 2.4 Running the game
 - 2.5 Pixels, metres and directions in 2D and 3D
 - 2.6 Physics bodies and collision
 - 2.7 Keys, buttons and the Input Map
 - 2.8 Groups
 - 2.9 Signals
 - 2.10 Other nodes you will meet
3. The six ideas, in plain words
 - 3.1 States
 - 3.2 Actions

- 3.3 Transitions
- 3.4 Events
- 3.5 Sub States
- 3.6 Parallel States

4. The NodeWorthy panel

- 4.1 Turning NodeWorthy on
- 4.2 Machines and templates
- 4.3 The ready-made templates
- 4.4 The graph, button by button
- 4.5 The right-click menu, item by item

5. Your first machine

- 5.1 A level you can play, in five minutes
- 5.2 Your first change, a chest that opens when you walk into it
- 5.3 Where to go next
- 5.4 With the NodeWorthy app, a hop on the Space bar in five minutes
- 5.5 An item the player picks up, step by step
- 5.6 A potion the player drinks, step by step
- 5.7 A sword the player wears, step by step
- 5.8 A chest with loot, step by step
- 5.9 A quest from a villager, step by step
- 5.10 A table of weapons, step by step
- 5.11 Someone to talk to, step by step

6. The pieces in depth

- 6.1 The Machine node
- 6.2 States
- 6.3 Actions
- 6.4 Events and transitions
- 6.5 Variables
- 6.6 Asking a question with Checks
- 6.7 The Events tab, with events written down, what they carry, and global events
- 6.8 Sub states
- 6.9 Parallel states, for several things at once
- 6.10 Notes and groups
- 6.11 Which step things run in
- 6.12 Making things move, and animation
- 6.13 AI and what a character notices
- 6.14 The Theme tab and how the game looks
- 6.15 Dialogues and the dialogue graph
- 6.16 Quests, for things to do, ticked off as they are done
- 6.17 Fighting with Hurt Box, Hit Box and weapons
- 6.18 Levelling up with experience and levels
- 6.19 Carrying the player on from scene to scene
- 6.20 Data, for records and tables of your game's things
- 6.21 3D Terrain, a landscape to walk over

7. Testing and fixing problems

- 7.1 Watching a machine while the game runs
- 7.2 The problems button
- 7.3 Can a machine get stuck in a loop?
- 7.4 A checklist for when a machine does nothing

8. How-tos, from small to big

- 8.1 How to make a ball that hops (something physics moves)
- 8.2 Small patterns worth knowing
- 8.3 How to make a twin stick shooter
- 8.4 How to make a first person shooter
- 8.5 How to make a 2D roguelike (Hollow Spire)
- 8.6 How to make a rally game (Handbrake Rally)
- 8.7 How to make an arena shooter (Swarm Break)
- 8.8 How to make a 2D platformer (Lantern Hop)
- 8.9 How to give your game to other people

9. Recipes, small scenes that each show one thing

10. The actions and checks that ship

- 10.1 State Machine, for events, states and other machines
- 10.2 Logic, for true and false, choices, and doing something more than once
- 10.3 Math, for numbers and their sums, formulas, limits and chance
- 10.4 Vector, for Vector2 and Vector3, which are places and directions
- 10.5 String, for text and building it, formatting it and comparing it
- 10.6 Array, for many values in one variable
- 10.7 Data, for records and tables of your game's things
- 10.8 Time, for waiting, timers and the speed of time
- 10.9 Node, for finding, making, freeing and changing nodes
- 10.10 Transform, for moving and turning things, paths and distances
- 10.11 Physics, for forces, velocity, gravity, raycasts and collisions
- 10.12 Character, for a character that walks, jumps, dashes, swims and climbs
- 10.13 Input, for keys, mouse, touch and gamepads
- 10.14 Animation, for AnimationPlayer, AnimationTree and animated sprites
- 10.15 Audio, for sounds, music, volume and speech
- 10.16 Camera, for cameras that follow, shake, zoom and look around
- 10.17 Effects, for particles, marks, screen effects and shaders
- 10.18 UI, for labels, textures, values and choices on the screen
- 10.19 Scene, for changing scenes, pausing, quitting and the game window
- 10.20 Save & Files, for saving the game, and files and folders on the computer
- 10.21 Navigation, for finding a way round walls on the navigation mesh
- 10.22 TileMap, for reading and changing tiles and grid cells
- 10.23 Debug, with messages, checks and drawings for finding what is wrong
- 10.24 AI, for guards, hunters and things that run away
- 10.25 Combat, for attacks, bullets, health and stats
- 10.26 Color, for colours and tinting, fading and taking them apart
- 10.27 Lights, for turning lights on and off, and their colour and brightness

- 10.28 Material, for the materials 3D things are drawn with
- 10.29 Convert, for turning one kind of value into another
- 10.30 Application & Display, for the game window, the frame rate and screenshots
- 10.31 Video, for playing videos
- 10.32 Inventory, for items and finding, carrying, wearing and using them
- 10.33 Quests, for things to do, ticked off as they are done
- 10.34 Dialogue, for characters talking, and the answers the player picks
- 10.35 Inventory, Inventory Grid, Crosshair, Minimap, Item Collection, Icon Meter, Stat Panel, Globe, Bar, Skill Bar, Dialogue Box, Quest List, Quest Log, Hurt Box, Hit Box, Perception, Experience, Data Node and Performance Overlay, the nodes to add

11. Where things live

12. Glossary

13. Writing your own action, an optional extra for programmers only

- 13.1 Talking to a machine from a script
- 13.2 An action is one small script
- 13.3 A question for a Check
- 13.4 Changing how an action that ships behaves

1. What NodeWorthy is, in one page

2D and 3D. Nearly every action works in both, and the few made for one kind of game say so in chapter 10. Move To moves a sprite or a mesh, Apply Force pushes a RigidBody2D or a RigidBody3D, Raycast asks whichever world the node lives in, and the Distance check measures in pixels or in metres. There is no 2D version of anything to go looking for, because the action sees what you gave it and does the right thing. Where a setting is a place or a direction, type two numbers for a flat game and three for a 3D one.

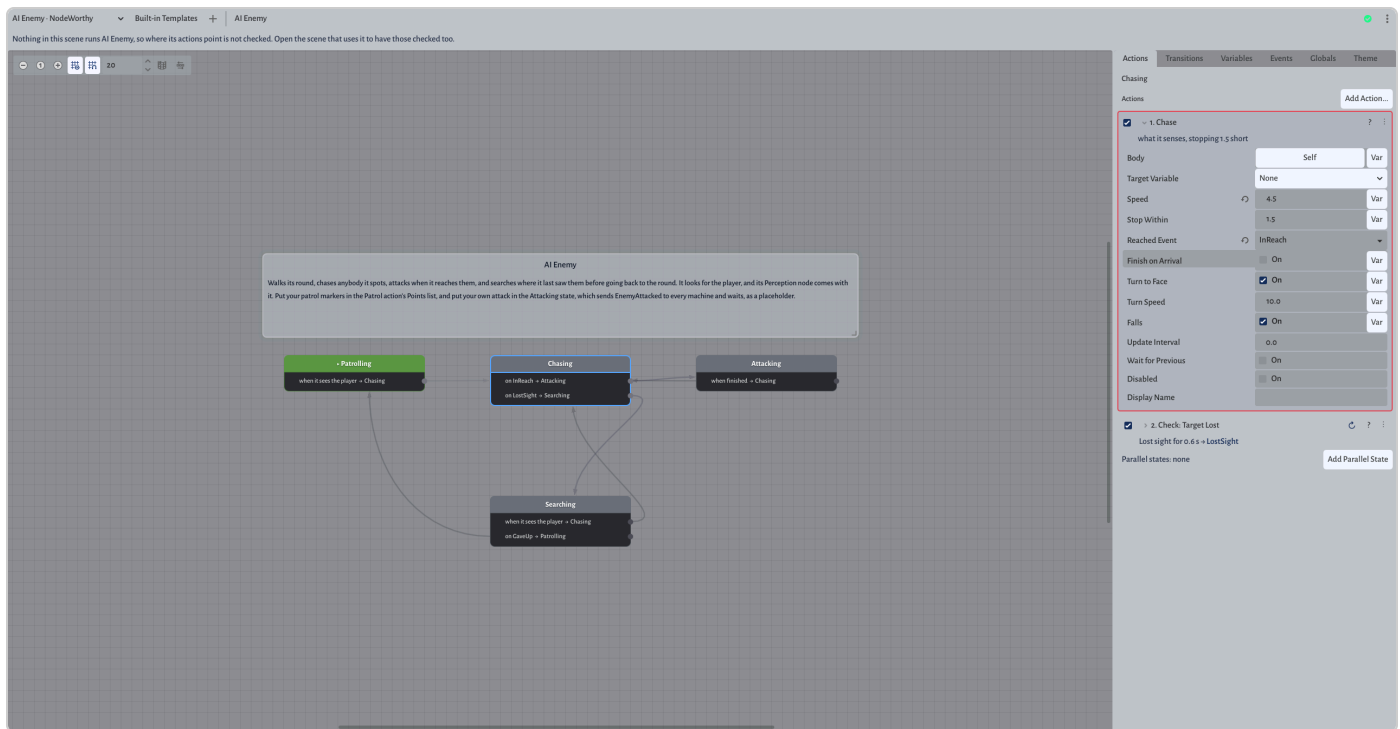
NodeWorthy lets you build behaviour by drawing it instead of writing it. You put a **Machine** node under something in your scene, hand it a template, and it runs that template while the game plays.

A template is made of four things.

- A **state** is a situation the thing it drives can be in, such as Off, On, Closed, Opening, Open or Broken. Only one runs at a time.
- An **action** is something a state does while it is the current one, for example wait two seconds, move up three metres, play a sound, set a property, call a method or spawn a scene. A state's actions run top to bottom.
- A **transition** is a way out of a state, taken when an event arrives, as in "on SwitchPressed, go to On". It leads to exactly one state. That can be another one, or the same state to start it again. A question about the world, such as "is anything within 2 metres of the door?", is asked by a **Check** action in the state, which sends the event.
- An **event** is a named message, like "SwitchPressed" or "Opened". A transition waits for one, and a Check action asks a question and sends one. Every state sends the event **FINISHED** to itself once all of its actions are done.

A state can also hold other states, as **sub states** that take turns or **parallel states** that run side by side. Chapter 3 explains all six ideas in plain words, with no programming needed.

Nothing is tied to characters. The same system drives a door, a lift, a pickup, a camera, a menu, a spawner, an enemy, or the flow of a whole level.



The NodeWorthy panel, with the toolbar along the top, the template in the middle, and the state you are working on in the tabs on the right. This is the AI Enemy template, with its Chasing state picked.

1.1 If you have the NodeWorthy app

The NodeWorthy app comes with everything in one folder, including the Godot editor, NodeWorthy already in place, the games and recipes, and this manual. There is nothing to download or set up, and it keeps itself apart from any copy of Godot you may already have. Start it and this is what you see.

- **New game** asks what your game is called, shows where it will be kept (a NodeWorthy Games folder in your Documents, or a folder you choose), and asks what it should start from, as a row of pictures to click. That can be an empty 2D game, an empty 3D game, or a copy of one of the seven games that ship, to take apart. **Create it** makes the game and opens the editor on it. A moving bar shows the app is working until the editor's window appears. The first time a game opens takes longer, as Godot gets everything in it ready. A name already in use is refused, so nothing of yours is ever written over.
- **Your games** lists the games you have made, the latest first. Click one to open it again. Always open your games from here, because double-clicking a game's files may open them in another copy of Godot, without NodeWorthy's setup.
- **Play a game made with NodeWorthy** is a row of the seven games that ship, each with its picture. Click one and it starts straight away in a window of its own, with no game to make first. Close its window to come back.
- **Read the manual**, **Look up an action** and **See what is ready-made** open the three books that come with NodeWorthy, which are this manual, the Action Reference, and Heroes, Things and Nodes. **About** says what NodeWorthy is built on, and **Quit** closes the app.

The editor starts in NodeWorthy's own look, which is its splash, its font, and a light cool grey with navy for everything picked or highlighted, which reads well on any screen. Press **Dark** at the top right of the editor and the whole editor turns dark in the colours of the NodeWorthy website, NodeWorthy's own panels with it. **Light** in the same place turns it back. The look is made of ordinary Godot settings. To change them, open **Editor Settings** and look under **Interface > Theme** for the colours and **Interface > Editor** for the font. On Windows it is in the **Editor** menu, and on a Mac it is in the **Godot** menu at the top of the screen (or press Cmd and the comma key). Whatever you choose there is kept.

An empty game is not quite empty. It opens on the **NodeWorthy** tab, which shows a note headed **Start here** and a single box, **Hop**. The scene itself, an orange ball resting on a floor, is under **2D** or **3D** at the top. The first thing you do is press **Run**, at the top right of the editor, and see something move. The ball hops, lands and hops again. Its two actions, which do the hopping, are in the **Actions** tab on the right, with the first of them open. Change its **Force**, run it again, and you have edited your first machine. Chapter 5 builds one from nothing, and 5.4 makes this ball wait for the Space bar in five minutes. Section 8.1 builds the same ball from an empty scene, a step at a time.

A game started from Emberdeep, Mossbrook, Hollow Spire, the shooter, Handbrake Rally, Swarm Break or Lantern Hop opens on one of that game's levels, with its hero picked and the hero's machine in the NodeWorthy panel. It is your own copy of that game, in your game's `games/` folder (Hollow Spire in `games/hollow_spire/`, and so on). It sits outside NodeWorthy's own folder and is yours, so take it apart and change it as you like. An update never touches it. A new game started from the same one is always a fresh copy. (The original stays inside `addons/nodeworthy/games/`, to look at but never to change.) Chapter 8 walks through each of them, and 8.9 shows how to turn a game into something you can give to other people.

How this manual is laid out. Each chapter only uses what the chapters before it have explained. Reading from the front is the easy way in. Chapter 2 is a short tour of the Godot editor for anybody who has never used it. Chapter 3 explains the six ideas every template is built from. Chapters 4 and 5 show the NodeWorthy panel and walk you through your first machine. Chapter 6 goes through every piece in depth, and chapter 7 shows how to watch a machine run and find out why something is not happening. Chapters 8 and 9 are worked examples, from a hopping ball to a whole shooter. Chapter 10 is the full list of actions and checks, to look things up in. A glossary of plain definitions is in chapter 12. Chapter 13 is for programmers only, and you never need it to make a game.

2. Before you start, Godot in ten minutes

NodeWorthy lives inside **Godot**, a free game engine. You never write code with NodeWorthy, but you do use the Godot editor to place nodes, change their settings and press play. If you have used Godot before, skip to chapter 3. If you have not, this chapter explains, in plain words, every Godot idea the rest of the manual relies on. You do not need to remember it all now, because each idea is in the glossary (chapter 12) as well.

Worth reading alongside this chapter. Godot's own documentation opens with a **Getting Started** section, free on the Godot website (docs.godotengine.org). Its pages **Nodes and Scenes** and **Creating instances** are the two to read first. Everything you build with NodeWorthy is made of nodes, saved as scenes and placed as instances, and those two short pages show it with pictures. You can skip its pages about writing scripts, since NodeWorthy does that part for you.

Nodes and scenes. In Godot everything in a game is a **node**, and a group of nodes saved together is a **scene**. When this manual talks about a thing in your game, such as a door, a coin or the player, it means the node at the top of that thing together with the nodes under it. That thing is usually saved as a separate scene and placed in the level. If you have used another engine where everything is a game object, a node is the nearest thing to it, and a scene placed in a level is what that engine would call a prefab placed in the world.

2.1 The editor window

When you open a project, the Godot editor fills the screen. It is split into panels that Godot calls **docks**. These are the ones you will use all the time.

- **The Scene dock**, usually at the top left. It lists the nodes in the scene you have open, as a tree. You pick a node here to work on it, right-click it for more choices, and press **+** at its top to add a new node.
- **The FileSystem dock**, usually at the bottom left. It shows the files in your project, such as scenes, pictures, sounds and NodeWorthy templates. Double-click a scene to open it. Clicking a file selects it and shows its settings in the Inspector.
- **The Inspector**, usually on the right. It shows the settings of whatever you have selected. More about it in 2.3.
- **The Node dock**, a tab beside the Inspector. It lists the node's **signals** (2.9) and **groups** (2.8). NodeWorthy picks both from lists of its own, so the NodeWorthy view leaves this dock out, and the Godot view shows it.

- **The main view** in the middle, where you see and arrange your level. Buttons along the top switch it between **2D**, **3D**, **Script**, **Game** and, in the NodeWorthy view, **NodeWorthy**.
- **The bottom panel**, along the bottom edge. It has tabs such as **Output** (messages and warnings) and **Debugger**, and in the Godot view (below) also **Audio** (the sound mixer), **Animation** and the NodeWorthy panel.

Two views, one button apart. The NodeWorthy app opens your games in the **NodeWorthy view**. The NodeWorthy panel, where you draw your templates, has a **NodeWorthy** tab of its own beside 2D, 3D, Script and Game, and fills the middle of the editor, with Godot's Inspector beside it as on every other tab, so the settings of whatever you click are always in sight. For more room for the graph, press **Hide Inspector** at the top right of the editor. It stays put away while the NodeWorthy tab is up, comes back the moment you go to 2D, 3D, Script or Game, and comes back beside the graph by itself when you open an item or a quest, which are set up nowhere else. **Show Inspector** in the same place keeps it beside the graph again. Every dock and panel of Godot's is there in both views. The **Godot view** button at the top right of the editor shows the NodeWorthy panel as a tab in the bottom panel under your level instead. Press **NodeWorthy view** in the same place to come back. The editor remembers the view you last chose.

The editor also starts in NodeWorthy's colours and font. **Dark**, at the top right, turns the whole editor dark in one press, and **Light** turns it back. They are ordinary editor settings, so you can also change them in Editor Settings like any other.

Everything NodeWorthy adds to Godot's editor. The rest of the editor is Godot as it comes, and works as Godot's own help says. These are the parts that are NodeWorthy's, and where this manual explains each.

Where	What is there	What it does
Top right	Godot view and NodeWorthy view	Switches between the two views above. The editor remembers the one you chose.
Top right, in the NodeWorthy view	Dark and Light	Turns the whole editor dark, and back. The choice is kept.
Top right, while the NodeWorthy tab is up	Show Inspector and Hide Inspector	Puts the Inspector away while the NodeWorthy tab is up, for more room for the graph, or keeps it beside the graph again.
Top middle, in the NodeWorthy view	The NodeWorthy tab, beside 2D, 3D, Script and Game	The NodeWorthy panel, where templates are drawn (chapter 4). A dialogue (6.15) and a Data Table (6.20) open in it too.
The bottom panel, in the Godot view	The NodeWorthy tab	The same panel, under your level (4.1).

The NodeWorthy panel, while the game runs	Debug, Pause and Step	Watch a machine's states light up in the running game, hold it still, and move it on a step at a time (7.1).
Scene dock	NodeWorthy , in a node's right-click menu	Adds players, things and nodes, and gives a character health, senses or a gun. It offers only what suits the node you clicked (4.5).
Scene dock	A warning triangle beside one of NodeWorthy's nodes	Godot's own sign that something is missing. Hover over it and it says what, such as a Bar with nobody to follow.
FileSystem dock	Create New , in a folder's right-click menu	NodeWorthy Item..., Quest..., Dialogue..., Actor..., Data Definition..., Enum..., Data... and Data Table... each make a file of that kind in the folder clicked, or in another picked with Folder... in the window (chapter 5, 6.15, 6.16 and 6.20). Action... and Check... are for programmers (13.2). The four used most are also on the NodeWorthy panel's + menu.
Inspector	The settings of a ready-made player or thing, at the top	Sections that fold, above Godot's own settings, with a button to make them the template's and one to open the template (2.3 and 4.3).
Inspector	NodeWorthy's own nodes and files	A line at the top says what the thing is for, and anything that names a variable, an event or a node is a list to pick from (2.3).
The window that adds a node	NodeWorthy's nodes, such as Machine, Inventory and Bar	Listed among Godot's own, each with its icon (10.35).
Bottom right	Notifications	NodeWorthy says here what it just did, such as what Add Player added to the scene.
Project Settings	Layer Names	The physics layers NodeWorthy uses are given names, which are World, Player, Enemies, Pickups, and Hit And Hurt Boxes (2.6).

Along the very top are the menus. **Project > Project Settings** holds settings for the whole game, such as the **Input Map** (2.7) and **Layer Names** (2.6). Messages from the editor, such as a file being created, pop up as **notifications** at the bottom right.

2.2 Scenes and nodes

Everything in a Godot game is built from **nodes**. A node is one small building block with one job, such as a picture, a sound, a camera, a collision shape, a piece of text or a button. You make a thing in your game by putting several nodes together.

A **scene** is a group of nodes saved together as one file, ending in `.tscn`. A level is a scene. So is a player, a door, a bullet or a menu. A scene can be used inside another scene as many times as

you like. For example, a level scene can hold ten copies of the coin scene. Drag a scene file from the FileSystem dock into the level to place a copy of it. Each copy works on its own.

The nodes in a scene form a tree, shown in the Scene dock. A node placed under another one is its **child**, and the one above is its **parent**. The node at the very top is the scene's **top node**, sometimes called its root. A child is carried along by its parent. Move the parent and every child under it moves too, the way a sticker goes wherever the box it is stuck on goes. It never works the other way round. This matters a great deal for physics bodies (see 8.1).

Door	Node3D	← the top node: the door as a whole
├ MeshInstance3D		← its shape, carried along
└ SCMachine		← a NodeWorthy machine driving the door

Adding a node. Select the node that should be its parent in the Scene dock, press **+** at the top of the dock (or right-click and choose **Add Child Node**), type the kind of node you want, such as *Sprite2D*, and press **Create**. Double-click a node's name to rename it. Drag nodes up and down the tree to change their parents.

Making a new scene. Choose **Scene > New Scene**, pick what its top node should be (2D Scene, 3D Scene or User Interface are the usual starts), then save it with **Scene > Save Scene**.

Paths. Godot writes the place of a file in your project starting with `res://`, which means "the project folder". So `res://nodeworthy/` is a folder called nodeworthy at the top of your project.

Editable Children. Normally you cannot see or change the nodes inside a copy of a scene from the level it is placed in. Right-click the copy in the Scene dock and tick **Editable Children** to open it up, for example to reach a Machine inside it.

A small **warning triangle** beside a node in the Scene dock means Godot thinks something is missing, such as a physics body with no collision shape. Hover over it to read why.

2.3 The Inspector and a node's settings

Click a node and the Inspector shows everything you can set on it, grouped under headings. Godot calls each setting a **property**. A *Sprite2D*'s picture is a property, a *Label*'s text is a property, and so is every node's **Position**, **Rotation** and **Scale**, found under the heading **Transform**. *NodeWorthy*'s actions can read and change any property while the game runs.

Some properties hold a piece of data rather than a plain number, such as the shape of a collision shape or the mesh of a 3D model. These read `<empty>` at first. Click the box and choose **New ...**, such as **New BoxShape3D**, to make one, then click it again to change its size.

Near the bottom of every node's Inspector is a **Process** heading. **Process Mode** decides whether the node runs while the game is paused. **Pausable** stops with the game, **Always** keeps going

(which a pause menu needs), **Disabled** never runs, and **Inherit** does what its parent does.

Process Priority decides which nodes run first in each frame.

NodeWorthy first, the Inspector second. You can set almost anything in the Inspector, but with NodeWorthy you will rarely need to, and it is always best to set a thing up with NodeWorthy's actions first. An action says what it does in plain words, offers lists to pick from, is checked by the problems button, and shows in the NodeWorthy panel. Come back to the template later and it still tells the whole story. A setting changed by hand in the Inspector does none of that. Keep the Inspector for Godot's own settings that no action covers, such as the size of a collision shape or a sprite's picture. To start with you will not need even those, because NodeWorthy's ready-made things (5.1) come with a shape and a picture already made.

NodeWorthy's own nodes, such as the Machine, have their settings in the Inspector too. Selecting a file in the FileSystem dock, such as a template, shows that file's settings there instead.

NodeWorthy's nodes and files, such as an Inventory, a Hurt Box, a Stat Panel, an item or a quest, are set up there like any Godot node or file. A line at the top says what the thing is for, and the settings that name a variable, an event or something in the scene are lists to pick from, never names to type. A Machine opens its template in the NodeWorthy panel, and so do a dialogue file and a Data Table.

A ready-made player or thing (4.3) has its own settings at the top of the Inspector, above Godot's, in sections that fold, such as Moving, Jumping and Loot. They belong to the one you clicked, so two enemies can walk at different speeds. An arrow appears beside a setting you have changed, and pressing it puts the template's value back. **Use These For Every Copy**, under the settings, makes your numbers the template's own, so every copy starts with them. **Open Its State Machine...**, beside it, opens the thing's template, its rules, in the NodeWorthy panel, with a line at the top saying whose rules they are and what else follows them, while the thing stays picked with its settings beside it. **Restore Defaults**, last, puts the template back as it came with NodeWorthy, every state, action and setting, keeping the name you gave it, so nothing you try can leave a player or thing broken for good. Everything that behaves like it goes back with it, and Undo brings your changes back.

2.4 Running the game

At the top right of the editor are the play buttons. **Run Project** starts the game from its main scene (the first time, Godot asks you which scene that is). **Run Current Scene** starts only the scene you have open, so it suits trying a recipe or one thing. **Stop** ends the game. While the game runs, messages it prints appear in the **Output** tab of the bottom panel, and the NodeWorthy panel shows you what every machine is doing (see 7.1).

The game either opens in a separate window or shows inside the editor's **Game** view, depending on how the editor is set up.

2.5 Pixels, metres and directions in 2D and 3D

Godot makes flat (2D) games and 3D games. Nodes made for one have **2D** or **3D** at the end of their name, as in `Sprite2D`, `Camera3D` and `RigidBody2D`.

- **Places and directions** are written as numbers separated by commas. That is two numbers for 2D (across, down) and three for 3D (across, up, forward or back). `0, 5, 0` is five straight up in 3D.
- **In 3D, distances are in metres** and up is a positive Y. **In 2D, distances are in pixels** and Y counts *down* the screen, so up is a negative Y. Where NodeWorthy turns metres into pixels for you, such as when a ready-made template is started on a 2D node (4.3), it counts 32 pixels to a metre.
- **Angles** are in degrees, everywhere, so 90 is a quarter turn. A 2D thing turns only one way, so it has one angle. A 3D thing can turn three ways, and NodeWorthy names them the same everywhere. **Tip** turns it forward or back around its X axis, the way a head nods. **Turn** turns it left or right around its Y axis, the way a door swings. **Roll** tilts it to one side around its Z axis, the way a plane banks. Most turns only need Turn.

2.6 Physics bodies and collision

Godot has a physics system that makes things fall, bump and bounce. Only special nodes, called **bodies**, take part in it. There are four kinds, each in a 2D and a 3D version.

Node	What it is for
StaticBody2D / StaticBody3D	Something solid that never moves, such as a floor or a wall.
RigidBody2D / RigidBody3D	Something the physics moves for you, like a ball, a crate or a barrel. It has weight, falls and can be pushed.
CharacterBody2D / CharacterBody3D	A walking character, such as the player or an enemy. Physics does not move it by itself. Something has to walk it every step, and in NodeWorthy that is the Move Character action (see 10.12).
AnimatableBody2D / AnimatableBody3D	Something solid that you move, such as a moving platform, which carries whatever stands on it.

An **Area2D** or **Area3D** is not solid at all, so things pass through it. It notices what comes in and goes out, which makes it a trigger zone, a pickup's reach or a pressure plate.

Every body and area needs a **CollisionShape2D** or **CollisionShape3D** under it, with its **Shape** set in the Inspector. The shape is how big the body is to physics. Without one, a body falls straight through the floor, and an area notices nothing. The shape you see is a separate node, such as a `Sprite2D` or a `MeshInstance3D`.

Collision layers. Everything solid stands on one or more numbered **layers**, and says which layers it bumps into or notices. Two bodies only touch when one of them bumps into the other's layer, and an area only notices what stands on a layer it watches. NodeWorthy names the first four for you, the same in 2D and 3D. **World** (layer 1) is the ground, walls and doors, and anything you add yourself, because Godot puts everything there. **Player**, **Enemies** and **Pickups** are where the ready-made players, enemies, coins and keys stand. A ready-made player or enemy bumps into every layer but Pickups, and a coin notices the player. The last layer, 32, belongs to Hit Boxes and Hurt Boxes.

Wherever NodeWorthy asks for layers, such as what a **Raycast** hits or what blocks an enemy's sight, it shows one line, such as *World*, *Enemies*. Press it to tick layers by name. Each says how many things in the open scene stand on it. **Name a new layer...** at the bottom gives the next free layer a name you choose, such as *Platforms* or *Water*, and **Rename a Layer** changes one of your names, but NodeWorthy's own four keep theirs. To put something you made yourself on a layer, right-click it and choose **NodeWorthy** → **Collision Layer**. A trigger area you make yourself watches only the World, but an **Event On Collision** listening for the player notices the player wherever they stand, and a bullet notices what it is told to hit. An area whose layers you picked yourself is left as you set it. A layer ticked in a setting that nothing in the open level stands on, and nothing the game makes while it runs is put on, would find nothing there. The field turns the warning colour, the list says which layers have nothing on them yet, and the problems button names it.

Frames and physics steps. A running game redraws the screen many times a second, and each redraw is a **frame**. Physics works on its own fixed beat, the **physics step**. NodeWorthy sorts out which work belongs in which (see 6.11). A machine does its regular work once a frame, or once a physics step when its **Physics Process** is on.

2.7 Keys, buttons and the Input Map

Players control a game with keys, the mouse, gamepads and touch screens. NodeWorthy names every one of them for you, such as **Key: Space**, **Mouse: Right Button** or **Pad: Left Stick Up**, and they work with nothing set up.

Godot also has an **Input Map**, under **Project > Project Settings > Input Map**. There you can make a named input action, such as *jump*, and give it several keys and buttons at once. NodeWorthy lists your own input actions too, and they work the same way (see 10.13).

2.8 Groups

Some settings are about *something else* in the game, such as what a guard looks for, what a sword or a bullet hurts, who a door opens for, or where to take cover. You never type a name for these. Press the setting and pick the thing in your scene.

- **One thing by its name**, such as *You* for your player.

- **Every Coin (3 here)** for every copy of the coin scene, the ones in this scene and any made while the game runs.
- **Everything under CoverPoints (4)** for all the markers kept under one node, such as a patrol route or the places to hide.

The setting then says what it means, *You* or *Every Coin (3 here)*. If nothing in the scene matches it, it says so in warning colour, and the problems button tells you which setting to fix. Picking what a guard looks for also gives it a Perception node to look with, if it has none.

Behind the scenes Godot finds these things by a **group**, a label shared by several nodes. NodeWorthy sets the groups up for you. If you already know Godot's groups, the line under each of these settings shows the group's name, and **Type a name of your own** at the bottom of its list takes one you have made yourself.

Good to know. The NodeWorthy graph also has "groups", which are notes that hold states together on the graph (see 6.10). They have nothing to do with Godot's groups of nodes. Where this manual means a graph group, it says so.

2.9 Signals

A **signal** is a node's way of announcing that something has just happened to it. A Button sends `pressed` when it is clicked. An Area sends `body_entered` when a body walks in. A slider sends `value_changed` when it moves. In the Godot view, the Node dock lists every signal the selected node has.

In ordinary Godot, a programmer connects signals to code. In NodeWorthy you never do that. A transition can wait for the node's own signals under **Signals** (see 6.4), and the **Event On Signal** action turns any node's signal into an event (see 10.9). Some signals hand over a value with them, such as the body that came in or the slider's new value. NodeWorthy calls these **arguments** and can keep them in a variable.

2.10 Other nodes you will meet

You do not need these to start. This table is here so that the names in the examples and the action list make sense.

Node	What it is
Node2D, Node3D	A plain node with a place, rotation and size, and nothing else. Useful as a parent to hold other nodes together.
Marker2D, Marker3D	An empty node that only marks a spot, such as a gun's muzzle, a patrol point or a spawn point.

Sprite2D, AnimatedSprite2D	A 2D picture, and a 2D picture with named animations made of frames (walk, jump). Sprite3D and AnimatedSprite3D put flat pictures in a 3D world.
MeshInstance3D	A 3D shape you can see, such as a box or a sphere.
Camera2D, Camera3D	What the player sees through.
Controls: Label, Button, RichTextLabel, TextureRect, TextureButton, HSlider, ProgressBar, SpinBox, VBoxContainer	The nodes that make up screens and menus, with text, buttons, pictures, sliders, bars, and boxes that line things up. A control's Mouse Filter setting decides whether it takes clicks, and Stop means it takes them for itself.
CanvasLayer	A layer drawn on top of the game that does not move with the camera. Scores, crosshairs and menus go under one.
ReferenceRect	An outline of a rectangle, used to mark out an area such as a room.
AnimationPlayer, AnimationTree	Play animations made by hand in the Animation tab, and blend a character's animations together (see 6.12).
AudioStreamPlayer (and its 2D and 3D versions)	Plays a sound. Sounds go through buses , the channels of the mixer in the Audio tab. Master is the main one.
GPUParticles2D / 3D	Particle effects such as sparks and smoke.
Path2D, Path3D	A curved line drawn in the level for something to follow.
NavigationRegion and NavigationAgent (2D and 3D)	The walkable area of a level, and the node that lets a character find its way round walls (see 10.21).
TileMapLayer, GridMap	A 2D map made of square tiles, and a 3D map made of pieces (see 10.22).
WorldEnvironment, CanvasModulate	The look of the whole 3D world (glow, fog, brightness), and a tint over a whole 2D scene.
VehicleBody3D	Godot's car on wheels.

3. The six ideas, in plain words

You do not need to know any programming to use NodeWorthy. You need six ideas, and you already know every one of them from everyday things like a lamp, a door or a washing machine. This section takes them one at a time. The rest of the manual goes into the detail when you want it.

3.1 States

A **state** is a situation something can be in. A lamp is either off or on. A door is closed, opening, open or closing. A traffic light is red, amber or green. At any moment the thing is in exactly one of its states, never two, never none.

States make a template easy to think about. Instead of describing everything a door does all at once, you answer one small question at a time. What does the door do while it is closed? Nothing, it waits. What does it do while it is opening? It slides up. Each answer belongs to one state, and each state stays short.

In NodeWorthy: press **+** in the toolbar, then **New State...**, and give it a name. It appears as a box. The first state you make has a green title bar. That is the state the machine is in when the game starts. Right-click another box and choose **Set as Start** to change that.

Good to know. Name a state after the situation, such as Open or Broken, not after what caused it. Every state in a template needs its own name.

3.2 Actions

An **action** is one job a state does while the machine is in it. Wait two seconds. Move up three metres. Play a sound. Change a colour. Show some words on the screen.

A lamp's On state might have two actions, to turn the light on and to play a click. A door's Opening state might have just one, sliding up two metres over one second. A state's actions are a list, like a to-do list, and they are started from the top down as soon as the state begins. When one should wait until the one above it is done, such as "slide the door, *then* play a thud", tick **Wait For Previous** on the second.

In NodeWorthy: click a state's box, then press **Add Action...** on the Actions tab on the right. Type what you want to do, such as "move", "wait" or "sound", pick one from the list and press **Add**. Its settings open underneath it, each with a line of help when you hover over it.

Good to know. When every action in a state is done, the state is done, and it says so with a message called **FINISHED**. You will meet it again under Events. An action set to **Every Frame** keeps going for as long as the state lasts. A state holding one never finishes by itself, so leave it on another event.

3.3 Transitions

A **transition** is a way out of a state. It is a rule that reads like a sentence. "While I am in this state, when *this* happens, go to *that* state."

Take the lamp again. It has two states, Off and On. Off has one transition, to go to On when the switch is pressed. On has one back, to go to Off when the switch is pressed. Press the switch and the lamp comes on. Press it again and it goes off. Two states and two transitions, and the lamp works.

A transition always leads from one state to one other state, or back to the same state to start it again (see 6.4). In the graph it is a row at the bottom of the state's box, such as *on SwitchPressed* → *On*, with an arrow drawn to where it goes.

In NodeWorthy: right-click a box and choose **Add Transition**. Drag the little dot at the end of the new row onto the state it should lead to. Then click the row. On the Transitions tab, press **When**, which reads *Pick what makes it happen...*, and pick the thing it waits for. The list starts with everyday happenings, such as **When the player touches it**, and picking one of those also adds the action that makes it happen to the state.

Good to know. Taking a transition leaves the old state completely, stopping whatever its actions were doing, and the new state starts its actions from the top. A state can have several ways out. If two could be taken at the same moment, the higher row wins. **Move Up** and **Move Down**, on the : menu of a transition in the Transitions tab, change the order.

3.4 Events

An **event** is a message with a name, such as SwitchPressed, Opened, CoinCollected or TimeUp. It says that something has just happened, and that is all it does. On its own an event moves nothing. It only matters when the state the machine is in has a transition waiting for that name.

Think of a kitchen timer going off. What you do about it depends on what you are doing. If you are baking, you take the cake out. If you are reading in the garden with no cake in the oven, you ignore it. The ring is the event. "When the timer rings, take the cake out" is the transition, and only the baking you has it.

Events come from four places.

- **The state itself.** A state sends **FINISHED** when all its actions are done. This is the event you will use most. It means "this step is over, move on".
- **An action. Send Event** sends one by name. A **Check** action asks a question, such as "is the lift at the top floor?", and sends one event for yes and another for no.
- **Godot itself.** Things that happen to the node, such as something walking into it (**body_entered**) or the mouse clicking on it (**pressed**), are listed under **Signals** in a transition's **When** list, with the names Godot gives them.
- **Somewhere else**, which is another machine, or a line of code if you ever write any.

In NodeWorthy: nothing needs setting up first. Give a Send Event action a name, then press a transition's **When** and pick that name from the list, and the two are connected. **Name an event of your own...**, at the bottom of that list, lets you type a new name. The When list, and the ▼ beside every event field on an action, list the names already in use, so you never have to type one twice. The **Events** tab is where you can write down what an event means, give it a value to carry, such as the points a coin is worth, or make it **global**, heard by every machine in the game.

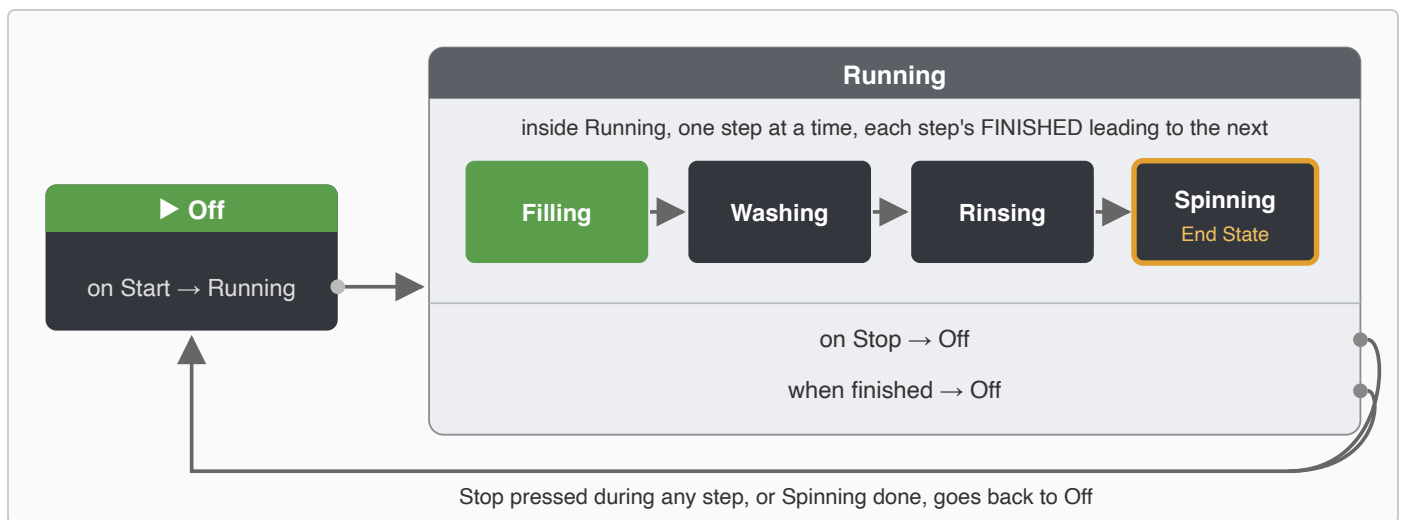
Good to know. Names must match exactly, so Opened and opened are two different events. And an event nobody is waiting for is lost. It is not kept for later, so arriving in the right state a moment afterwards does not bring it back.

3.5 Sub States

A state can have other states inside it. These are **sub states**, and they take turns, one at a time, just like the states outside.

Think of a washing machine. From the outside it is either Off or Running. But inside Running there are four steps, one after the other. They are Filling, Washing, Rinsing and Spinning. Those four are sub states of Running.

Here is why that helps. The machine needs a Stop button that works during every step. Without sub states you would add a "when Stop is pressed, go to Off" rule to all four steps. With sub states you add it once, to Running. Whichever step is going on, pressing Stop leaves Running, and leaving Running leaves the step inside it too.



Running's two rules are written once, on Running, and work whichever step is going on inside it.

Who hears an event first. When an event arrives, the step inside hears it first. Only when that step has no transition for it does Running get its turn. So one step can have its own answer to an event, and Running catches everything the steps leave alone.

In NodeWorthy: right-click a state and choose **Add Sub State**. The graph opens the inside of that state, with a trail of names at the top showing where you are. Build the states inside exactly as you would anywhere else. Press **Up**, or click a name in the trail, to come back out, and double-click a box marked  to go in again.

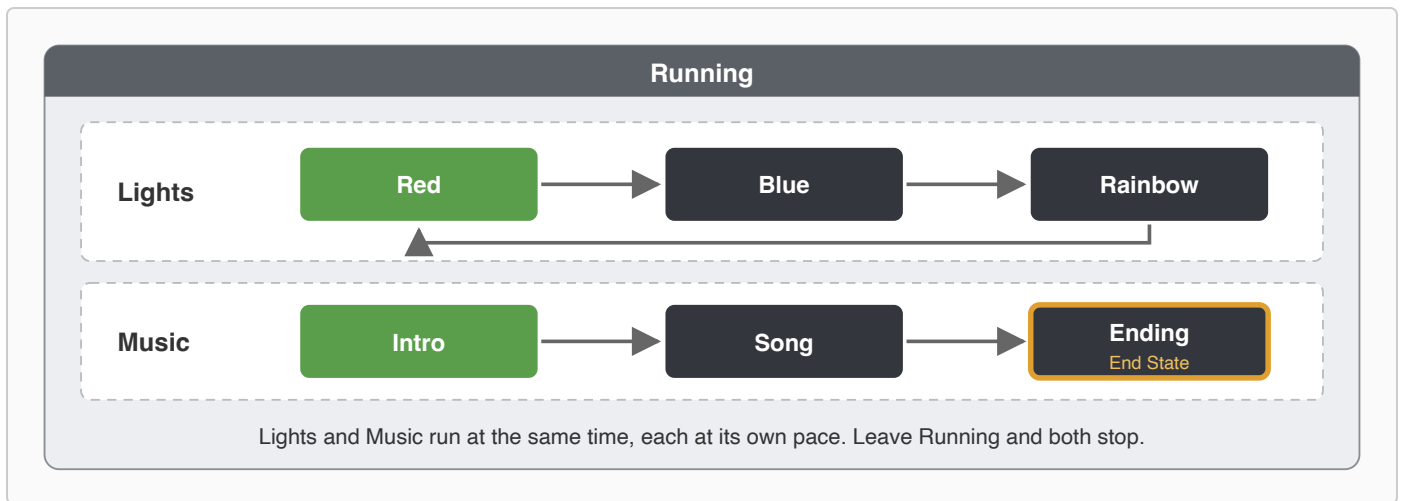
Good to know. To tell Running the wash is over, right-click Spinning and choose **Mark as End State**. When Spinning's actions are done, and Running's own actions too, Running sends FINISHED, so a transition on FINISHED on Running can go back to Off.

3.6 Parallel States

Sometimes one thing does two jobs at the same time, and each job has its own steps. That is what **parallel states** are for. Sub states take turns, but parallel states run side by side.

Think of a fairground ride. While it is Running, its lights go round their patterns, from red to blue to rainbow and round again. At the same time its music works through its own tracks, playing an intro, a song and an ending. The lights do not wait for the music, and the music does not wait for the lights. Each goes at its own pace.

So inside Running you make two parallel states, Lights and Music. Each one is a little set of states, with its own first state, and is always in one of them. Both run for as long as Running does.



Two parallel states inside Running. Each one is always in one of its own states, whatever the other is doing.

- **They stop together.** When Running is left, Lights and Music both stop, wherever each had got to.
- **Every one hears an event.** If Lights and Music both have a transition waiting for the same event, both take it. Running itself gets the event only when none of them did.
- **Waiting for all of them.** Mark the last state of each as an **End State** and give Running a transition on **FINISHED**. It waits until every parallel state has reached its end.

In NodeWorthy, click the state, then press **Add Parallel State** at the bottom of its Actions tab, or right-click its box and choose **Add Parallel State**. Name it after its job, such as Lights. The graph opens it, empty, ready for its states. Do the same for Music. The box then reads *2 parallel states: Lights, Music*. Double-click it to open the first, or right-click it and use **Open Parallel State** to pick one.

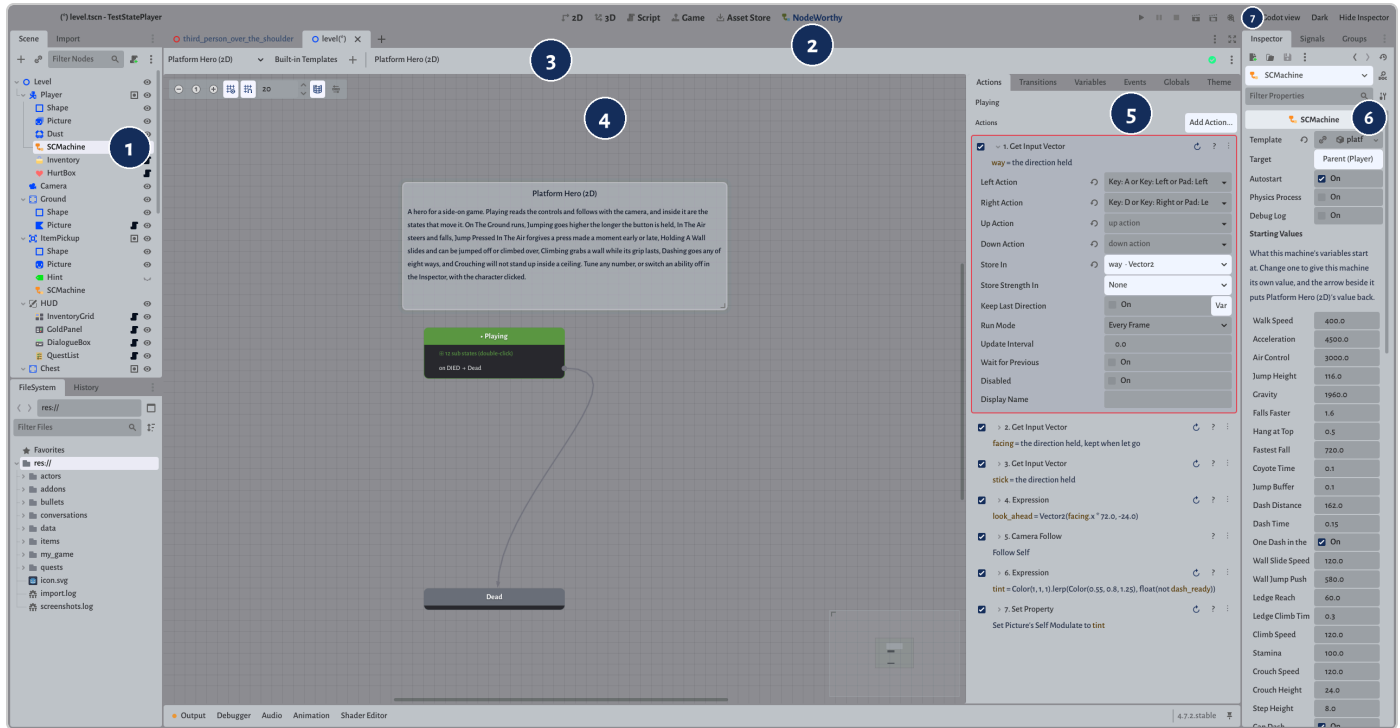
Good to know. A state holds sub states *or* parallel states, not both. Steps that take turns are sub states, and jobs that happen side by side are parallel states.

In short, a **state** is a situation, **actions** are the jobs it does, a **transition** is a way out, an **event** is the message that takes it, **sub states** are steps that take turns inside a state, and **parallel states** are jobs that run side by side inside one.

Chapter 6 comes back to each of these in full, with states in 6.2, actions in 6.3, transitions and events in 6.4, sub states in 6.8 and parallel states in 6.9.

4. The NodeWorthy panel

This chapter shows where NodeWorthy lives in the editor and what each part of its panel does. Some of the words in the button table belong to later chapters, so skim it now and come back to it whenever you need it.



The editor in the NodeWorthy view, the way a new game opens, with the player's Machine picked.

1. **The Scene dock.** Pick a Machine here (it is called SCMachine) and the NodeWorthy tab shows the template it runs.
2. **The NodeWorthy tab,** beside 2D, 3D, Script and Game. Press 2D or 3D to go back to your level.
3. **The template bar.** Which template is on show, Built-in Templates for the ones that came with NodeWorthy, and + to make something new (4.4).
4. **The graph.** Each box is a state, each arrow a transition, and the green box is where the machine starts.
5. **The tabs on the right.** Actions, Transitions, Variables, Events, Globals and Theme, for the state or the template you have picked (chapter 6).
6. **The Inspector,** beside the graph, with the picked Machine's own settings and starting values. **Hide Inspector** at the top right puts it away.
7. **The view switch.** It reads **Godot view** here. Press it and the panel moves to the bottom of the editor, under your level (4.1).

4.1 Turning NodeWorthy on

NodeWorthy is already switched on in every game the NodeWorthy app makes. (Should the **NodeWorthy** panel ever be missing, open **Project > Project Settings > Plugins** and tick **NodeWorthy**. **NodeWorthy Tab** is ticked by NodeWorthy itself the first time you choose the NodeWorthy view. Leave **NodeWorthy (editor parts)** unticked, as it only keeps NodeWorthy's own panels out of Godot's Add Child Node window.)

Where the panel sits depends on the view (2.1). In the **NodeWorthy view** it is the **NodeWorthy** tab beside 2D, 3D, Script and Game, filling the middle of the editor. Selecting a Machine in the Scene dock, or a template in the FileSystem dock, brings that tab up by itself, and **2D** or **3D** takes you back to your level. Opening a scene that has a Machine does the same, and shows that Machine's template, the player's when the scene has a player. The template is fitted to the view, the state it starts in is on show, and that state's first action is open and outlined in the Actions tab. Every tab on the right has one row outlined, so you can always see which row you are on. In the **Godot view** it is a tab in the bottom panel, beside Output and Animation, with your level in sight above the graph. Selecting a Machine or a template opens it by itself, the way selecting an AnimationPlayer opens the Animation panel, and selecting anything else leaves it as it is. Drag its top edge to make it taller, press the bottom panel's expand button to fill the editor with it, or drag its tab out to float it in a separate window, on a second screen if you have one. The **Godot view** and **NodeWorthy view** button at the top right of the editor switches between the two.

Right-clicking a node in the Scene dock offers a **NodeWorthy** entry, holding **Add State Machine...** and the nodes that come with it (10.35).

4.2 Machines and templates

Two words are used everywhere in NodeWorthy, and it helps to have them straight before you start.

- A **template** is a file in your project holding the states, their actions and transitions, the variables and the events. It is what you draw in the NodeWorthy panel. Your own templates are kept in `res://nodeworthy/` (see chapter 11).
- A **Machine** is a node you put in a scene, under the node it should drive. It runs one template while the game plays. In the Add Node window it is called **SCMachine**.

One template can be run by many machines. Ten coins in a level can share one Coin template. Change the template once and every coin changes. Each machine still keeps its own current state and its own variables, and can be given its own starting values (see 6.1).

The quickest way to make both at once is to right-click a node in the Scene dock and choose **NodeWorthy → Add State Machine...**, which chapter 5 walks through. The panel's template list (4.4) shows the templates you make, and selecting a Machine in the Scene dock shows the template it runs, whether it is one of yours or one that came with NodeWorthy.

4.3 The ready-made templates

Ready-made templates are read-only. They live inside NodeWorthy's own folder, which an update replaces. The panel shows them as they came, so press **Make It Mine** and change the copy. Dragging a box in one does the same in one go, and the box stays where you dropped it, in your copy.

Twelve ready-made templates ship in `addons/nodeworthy/templates/`. **Hello World** is the smallest template that does something. It prints Hello to the Output panel, waits a second, prints World, and goes round for ever. Put it on a Machine under any node and run the scene. **AI Enemy** walks its round, chases anybody it spots, attacks when it reaches them, and searches where it last saw them before going back to the round. Its Attacking state sends EnemyAttacked and waits, a placeholder for your own attack. The others are **Guard**, **Chaser**, **Skittish Villager** and **Pack Hunter**, which calls the rest of the pack and then circles at a distance rather than walking straight in. Start one with **Add State Machine...** and it comes with the Perception node it looks with, so it works straight away. Open it in the graph and you can see exactly how it was built and change it. AI Enemy and Guard also come with their round, two markers beside the character in the level, named after it, such as **Guard Round**. Drag **Point1** and **Point2** to where it should walk. Pack Hunter circles and keeps its distance, which only a character body can do, so it is offered only on a CharacterBody2D or CharacterBody3D. Their speeds and distances are set for a 3D game, in metres, and their characters fall. Start one with **Add State Machine...** on a 2D node and your copy is turned into pixels by itself, at 32 pixels a metre, and in a game seen from above **Falls** is unticked on its movement actions too. There is nothing to work out or retype.

To start from one of them, pick it under **Start From** when you choose **NodeWorthy** → **Add State Machine...**, or under **New Template...** in the template list. You get your own copy to change as you like, and the one it came from stays as it is. They are also in **Built-in Templates** in the toolbar (4.4), beside every recipe and the templates of the games, to look through before you pick one. The Perception node the AI templates come with is explained in 6.13.

Six of them are whole players. **Platform Hero (2D)**, **Platform Hero (3D)**, **Side-On Platform Hero (3D)**, **Top-Down Hero (2D)**, **Top-Down Hero (3D)** and **Third Person Hero (3D)** are characters that already walk, and the side-view and 3D ones jump and do everything else a player expects, tuned so they feel right before you change a thing. A player is more than a template, though. It needs a body, a collision shape and a camera. As well as appearing under Start From, the players have their own entry. Right-click the level in the Scene dock and choose **NodeWorthy** → **Add Player**, then one of them. Only the ones that suit the scene, 2D or 3D, are offered. Started from under Start From on a body you made, a hero is given the puff of dust its jumps make, and NodeWorthy says which other node its actions look for beside the Machine, such as **Picture** (2D) or **Model** (3D). Give that name to the node that draws your character. The Top-Down Hero is for a game seen from above. It walks all eight ways with WASD, the arrows or a stick, faces where it walks, and nothing falls. The camera looks a little ahead of it. Space, Shift or a pad's bottom

button dashes the way it is pushed (or the way it faces), with a squash and a puff of dust, and it is tinted blue until the dash comes back after its **Dash Cooldown**. Its settings are under Moving and Dash, with **Can Dash** to switch the dash off. For a player that walks where the mouse clicks, as an action role-playing game is played, add **Click To Move** (10.12) to the Top-Down Hero's walking state. Once you have made a player your own, adding health, a gun or new states, use it in your next level too. Add the same hero there, then right-click it and choose **NodeWorthy** → **Behave Like** and pick your template, which is named with the level it came from, such as **Platform Hero (2D), in Level 1**. That player runs the very same template, so everything you added comes along, and a change made in one level is made in all of them. That builds the body, its shape, something to see it by (the Platform Hero comes as a drawn character with an animation for everything it does, and the three newer 3D heroes as an animated person, and the others as a plain shape until you give them art), a camera at the scene's root and a Machine running your own copy of the template.

The Platform Hero's states are **Playing** (which reads the controls, follows with the camera a little ahead of where it faces, and tints it blue while its dash is spent, over any Tint you chose for it) and, nested inside it, **On The Ground, Jumping, In The Air, Jump Pressed In The Air, Landing, Holding A Wall, Climbing A Ledge, Climbing, Sliding Down, Topping Out, Dashing and Crouching**. It has the feel of the platformer that comes with NodeWorthy (8.8). It gets up to speed at once, a jump forgives a press a moment late off a ledge or a moment early before landing and goes higher the longer it is held, a jump pressed in mid air that comes to nothing shows nothing, holding down falls faster, a dash goes any of eight ways and the ground gives it back, a wall can be slid down, jumped off or climbed over, holding J or the left trigger against a wall climbs it while its grip lasts, and holding down, or C, crouches and crawls. The stick counts as down only when it points mostly down, so a thumb a little off level never crouches it. It squashes and stretches as it jumps and lands, puffs dust, and plays idle, run, jump, fall, wall, climb, dash, crouch and crawl, so a sprite sheet you draw with those names drops straight in. Every number is a variable, tuned in the Inspector. Click the player and its settings are at the top, under Moving, Jumping, Walls And Ledges, Dash and Crouch, each ability with a tick to switch it off (**Can Dash, Can Wall Slide, Can Climb and Can Crouch**). With the wall slide switched off a wall only stops the hero, which no longer slides down it, jumps off it or climbs over its ledge, though holding grab still climbs it while climbing is on. The **Side-On Platform Hero (3D)** is the same hero in a 3D world seen from the side. It has the same states, feel, settings and switches, in metres, and runs and jumps along one line (X) while its depth is held still and the camera watches from the side, a little ahead. It is a box 1.8 m tall wearing the animated person the 3D recipes use (Idle, Sprint, Jump, Crouch and Roll among its clips), glows blue while its dash is spent, and puffs dust. Its moves say **Side On** in Move Character's **Space**. That keeps a stick's up and down, a dash and slipping round a ceiling's corner working along the line, and any 3D character you make can do the same. The **Platform Hero (3D)** is free to run anywhere in a 3D world, the camera behind it turning with the mouse or the right stick (Orbit Camera). It runs where the camera looks and turns to face that way, jumps with the same forgiveness as the 2D hero, dashes the way it is running (or, standing still, the way it faces), glowing blue until it lands, slides down walls and jumps off

them, reaches for a ledge the moment it takes hold of a wall and climbs over when there is one in reach, climbs a wall while grab (J or the left trigger) is held and its grip lasts, and crouches with C or the right face button (B / Circle). Its settings are under Moving, Jumping, Walls And Ledges, Dash, Crouch and Camera, each ability with a tick to switch it off. The **Top-Down Hero (3D)** is for a 3D game seen from above, the camera looking down from above and behind. **Walking** goes all eight ways at once to full speed and snaps round to face the way it goes, walks up steps, and a stick walks as fast as it is pushed. **Dashing** (Space, Shift or a pad's bottom button) goes the way the stick points, or the way it faces when nothing is held, and the hero glows blue until the dash comes back after its **Dash Cooldown**. Walking off an edge, it falls and lands with a puff of dust. Its settings are under Moving, Dash and Camera, with **Can Dash** to switch the dash off. The Third Person Hero has **On The Ground**, **Sprinting**, **Jumping**, **In The Air** and **Crouching**, and walks where the camera is looking. Nothing about either is hidden. Every state is made of ordinary actions with ordinary settings, and the way a character feels is yours to change.

Every ready-made player, thing and node has an entry of its own in **Heroes, Things and Nodes**, the third book that comes with NodeWorthy (NodeWorthy_Heroes_Things_And_Nodes.pdf). Each entry says how to add it, what it puts in the Scene dock, what every setting does, how to change it step by step, and what to do when it does not work. This section is the short tour.

Ready-made things. NodeWorthy → **Add Thing**, on the level, holds the rest. **Ground** is a wide solid floor, in 2D or 3D. **Terrain** is a landscape for a 3D level, with hills to walk over and grass, and takes the place of a flat Ground (6.21). **Model...** puts a model of your own into a 3D level, below. **Enemy** chases the player when it sees them and sends CaughtYou to every machine when it reaches them. The player opens a **Chest** with E (below). A **Villager** is someone to talk to. When the player comes near, "E to talk" shows over its head, and E plays the dialogue picked (or made with **New dialogue...**) at the top of its Inspector, in a Dialogue Box that comes with it, then gives the quest picked in its **Gives Quest**, if any. An **Item Pickup** puts the item picked at the top of its Inspector (or made with **New item...**) into the player's Inventory when they touch it, and disappears (the player is given an Inventory if they have none). Tick **Press To Pick Up** in its settings and the player walks up to it and presses E instead, with "E to pick up" over it, which suits a sword or a key more than a gem. With no item picked, or no room for it, it stays where it is and the Output panel says why. A **Level Exit** takes the player to another scene when they walk in. Pick which in its own settings in the Inspector, under **Goes To**, and the spot they come out at, such as a Marker2D called FromBelow, under **Arrive At**. That way two exits sharing one template can each go somewhere of their own. **Main Menu** is a title with Start and Quit over the whole screen (pick where Start goes in **First Level**, at the top of its Inspector). **Pause Menu** pauses the game on Escape or a pad's Start and shows Resume and Quit, keeping itself running while everything else is paused. Menus can go in any scene, even one with nothing 2D or 3D in it. Each comes as a body, its shape and a plain picture, in 2D or 3D to match the scene, with a template of its own. Clicking it shows its settings at the top of the Inspector, and **Open Its State Machine...** under them shows the template. Each lands to one side of where you right-clicked, standing on

whatever is below. Drag it where you want it. To make one bigger or longer, such as the Ground, click its **Shape** in the Scene dock and drag the orange handles in the view. Its plain picture follows, so what you see is always what is solid. A second Enemy or Chest shares the first one's template, so one change changes them all. **New Template for This Machine...** on one's Machine gives it a separate template. The Enemy's Hit Box hurts the player when it touches them, by its **Damage**, which starts at 1. After a touch it waits its **Time Between Attacks** before its touch hurts again. Set its **How It Fights** to **Swings** or **Shoots** and its touch no longer hurts. In a 3D scene with no light, the first ready-made thing or player added brings a **Sun** and a **Sky** with it, since without them a 3D game runs almost black. Move the Sun to change where the light comes from, or delete both to light the scene your own way. A player with no health is given it when an Enemy is added, and a player added to a scene with an Enemy in it is given it too, as **Give It Health** does, with health 3 on the screen, and at 0 the level starts again. An Enemy set to shoot comes only as close as its **Shooting Distance**, and keeps there while it shoots. Coins, doors, keys, levers, pressure plates, moving platforms and a score on the screen are each an action or two, and the recipes in chapter 9 build every one of them.

Chest. A **Chest** is solid and closed until the player walks up and presses E, or clicks it with a player that walks where it is clicked (Click To Move). Then it opens and spills its loot. The loot lies on the floor with a glow and its name in the colour of how rare it is, and is picked up by walking over it. What it can hold is at the top of its Inspector. Pick the items in **Can Drop**, how many it takes from them in **How Many Items** (rarer ones less often), how likely each is in **Item Chance**, and the gold in **Gold Least** and **Gold Most**. Each chest in a level can drop different things, and takes from its own list every time the level starts. Adding a Chest gives the player an Inventory to carry the items in, if they have none. In 2D its picture has a *closed* and an *open* animation. **Import Sprite Sheet...** with two pictures named for them, such as *Chest - Closed.png* and *Chest - Open.png*, gives it your own art.

The scene the game starts on. Right-click the scene's file in Godot's **FileSystem** dock and choose **Set as Main Scene**. **Run Project** (F5), and the finished game, then open on it. Godot asks this only the first time a project is run, and this is how to change it later.

Set Texture. Most 2D ready-made things and players start as a plain coloured shape. Right-click one and choose **NodeWorthy** → **Looks** → **Set Texture...**, then pick a picture anywhere on your computer. It is copied into the project's `pictures` folder, put in place of the shape, and sized to the thing's height. Say which way the character is drawn facing (the **Drawn facing** choice at the bottom of the window). Art drawn facing left is flipped to face right, the way everything starts. Turning round flips it, as it did the shape. One undo gives the shape back. The Platform Hero (2D) and the 2D Chest start as animated drawings, so **Set Texture...** is greyed out for them. Give those a new look with **Import Sprite Sheet...** in the same menu.

Import Sprite Sheet. A sprite sheet is one picture holding every pose of a character, a row for each animation, such as standing, running and jumping. Right-click the character and choose **NodeWorthy** → **Looks** → **Import Sprite Sheet...**, pick the sheet, and a window shows it with a

grid over it. Say how many pictures **Across** and **Down** (it guesses from the sheet), name each row (idle, run, jump, fall... a row left empty is skipped, as are empty pictures at the end of a short row), how many **Pictures a second**, and which way it is drawn facing, then press **Use It**. The character now plays its animations in place of the plain shape, sized to its height. The animations are kept in a file beside the sheet, so another level's player can use them too. A ready-made player's states are given the one that suits them. On The Ground plays idle, and run while it moves. Jumping plays jump, and In The Air plays fall (or jump). Any state named like a row plays that row. One undo gives the shape back.

A character drawn from several sides. Art for a game seen from above or at an angle (isometric) often comes as a sheet for each animation, idle, walk, attack, with a row for each way the character faces (towards you and to the left, towards you and to the right, away to the left, away to the right). Pick all the sheets at once in **Import Sprite Sheet...** and each is shown in turn. Choose **Each row is one animation, facing another way**, give the **Animation** its name (it is read from the file's name, so "Warrior - Walk.png" is walk) and say which way each row faces. The next sheet starts with the same choices. The character then plays walk, idle or attack from whichever side faces the way it goes, and stays facing that way when it stops. Play Sprite Animation still just plays walk. A side that was never drawn is the opposite side mirrored, so art drawn facing right serves for left too. Enemies given a sheet this way are animated too. Patrolling and searching walk, chasing runs if there is a run, and swinging plays attack.

Standing on its feet. A character cut from art drawn from several sides is seen from above or at an angle, so what is solid about it is where it touches the floor. Its shape becomes a round footprint at its feet, the same whichever way it turns, and its place (the point it sorts by and walls stop) moves down to its feet, with nothing moving on the screen. A wall then stops its feet at the wall, not its middle, and it is drawn in front of whatever it stands in front of. It also gets a soft shadow on the ground under it, and **Shadow**, under **Look** on its page, turns it off. Undo puts its old shape back.

Where the animations are kept. The animations cut from a character's sheets go in one file beside the sheets, named after the character. For example, *Archer - Idle.png*, *Archer - Walk.png* and *Archer - Attack.png* make *Archer_animations.tres*, which every archer in every level can share. A sheet holding one picture takes its animation's name from the file's, so *Chest - Open.png* is open. Choose **As big as it is drawn** under **Size** for art drawn to go with your tiles. The picture stands on the bottom of the thing's shape rather than hanging below it. **Tint** colours every picture, so one set of art makes a skeleton, a ghost and a boss. It is set as a sheet is cut, and changed any time under **Look** at the top of the character's Inspector, and each copy of a character can have a different one.

Your art brings its own animations. NodeWorthy plays the animations your pictures and models already have. It does not draw or make them. For 2D, use a sprite sheet with a row (or a sheet) for each animation. For 3D, use a model that comes with its animations, such as one from Mixamo, Quaternius or Kenney, or give a model the animations of another file with **Use**

Animations From... (Experimental), below. Free packs of both are easy to find, and many are free to use in a game you sell as long as you credit the artist.

Animations from another pack (experimental). A 3D character can also play animations made for a different model. Right-click it and choose **NodeWorthy** → **Looks** → **Use Animations From... (Experimental)**, then pick one file or several (**.glb** , **.gltf** or **.fbx**), such as a pack from Mixamo, Synty or Quaternius. NodeWorthy finds which bone is which by the skeleton's shape, where its legs, arms, neck and fingers branch, whatever the bones are called, or by the Avatar a pack made for another engine keeps beside each file. A file of animations with no character in it, as many packs come, works too. It sets the files up, and gives the character an **AnimationPlayer** of its own holding its model's clips and the new ones. Its states keep the clips they play, NodeWorthy's or your own, and none is swapped by itself. To use a new one, pick it for that state under **Animations** in the Inspector, such as the pack's walk for **Walk Clip**. A state that had no clip at all is given one that suits it. Any humanoid skeleton works, two arms and two legs. The first time, each file takes a few seconds to set up. This is experimental. Fitting animations made for one skeleton onto another does not yet work well for every model, so watch the character play each move before you rely on it.

A file of a whole cast. A character pack often keeps all its characters in one file, on one skeleton. When **Set Model...** brings in such a file, its window also asks which character, and the extras it wears, such as hair, a beard or a hat, ticking those named after the character. Only that one and its extras are taken in from the file, so the others are not in your scene or your game at all, and it stands where your character stands and as tall. Two characters can wear different ones from the same file.

Your own models in the level. A table, a crate, a wall, a tree or a whole building from a pack goes into a 3D level with **NodeWorthy** → **Add Thing** → **Model...** Right-click the node it should stand beside, such as the player, pick the model's file anywhere on your computer, and a window asks **What is it?** **Scenery** is solid and still. It stands on the ground beside the node you clicked, never on top of something already there, at the size its file made it, wearing the textures its pack gives each part. It is solid exactly where it is seen, so a hero walks round the legs of a table and through an open doorway. **Chest**, **Item Pickup** and **Level Exit** make it that ready-made thing, at its own size, its shape fitted to it. **Villager** and **Enemy** make it a character, as tall as a hero, and Set Model's window asks which way it faces and, for a file of a whole cast, which one. Your own templates are listed too, for it to do what one of them does. **Looks** → **Set Model...** on it puts another model in its place.

Characters that come with no animations (experimental). A pack's characters often come with none of their own. When **Set Model...** or **Add Thing** → **Model...** puts such a person on a character, it is given the animations of the person that came with NodeWorthy, made to fit its skeleton, so it stands, walks, fights and falls at once. To give it a pack of your own, such as a zombie's, pick **Animations from files of my own, such as a zombie pack... (Experimental)** in the **What is it?** window, and it plays that pack's moves from the start. Use **Looks** → **Use**

Animations From... (Experimental) later and the pack's moves are added beside the ones it has. None is swapped by itself, so you choose, one pick each under **Animations** in the Inspector. Fitting the animations to its skeleton is experimental, as above, so watch it move before you rely on it.

Models that come in white. A model file names the pictures each of its parts wears, for its colour, its normal map, glow and the rest, often as the artist's own computer kept them, in a folder that is not there or as a Photoshop file, so the model comes in plain white. When **Set Model...** brings in a model, NodeWorthy gives every part what it should wear, by itself. Pictures the file names that its pack holds under those names are set up for Godot to put on each part. Packs bought for other engines say exactly, in their own material files, what each part wears, its colour and tint, normal map, metallic and roughness, occlusion, glow, and whether it is see-through or two-sided, and NodeWorthy reads them. Otherwise it goes by the names the file holds and the names of the pack's pictures. It says what it put on. A model coloured without pictures is left as it is. Godot may say first that it could not open a picture the file names. That is the file pointing at a picture that is not there, and the next line tells you what NodeWorthy put on instead. When a part should wear a picture none can be told for, **Looks** → **Set Model Textures...** opens with that part first. It lists every part of the model and the pictures it wears, and any part's colour picture or normal map can be picked from anywhere on your computer, one part or every part at once.

Gun. Right-click a character and choose **NodeWorthy** → **Give It** → **Gun**. It gets a **Gun**, a second Machine with its own template. **Ready** waits for the **left mouse button**, or the right trigger on a pad, and **Firing** sends one bullet the way the character faces at every enemy, ready again a quarter of a second later. Bullets leave from a **Muzzle** marker at the middle of the body, and the bullet itself is a scene made for you in `bullets/`. A bullet hitting an enemy with a Hurt Box (Give It → Health) takes health off it. The ready-made Enemy is what it hits.

Health. Right-click any character that has a Machine and choose **NodeWorthy** → **Give It** → **Health**. It gets a Hurt Box, a `health` of 3 that never goes below 0, and a **Dead** state that every one of its states leads to when health runs out (the Hurt Box sends `DIED`). The player's Dead waits a second and starts the scene again, and anyone else's removes them. The player's health also shows on the screen, in a Stat Panel. One undo takes all of it back. How much damage, how long a character is safe after a hit, and how far it is knocked back are settings of the Hurt Box and the Hit Box, in the Inspector (6.17).

Bake Navigation Mesh. Adding Navigate To lays ground for it to find its way on over the whole level, baked from its walls and floors, so characters walk round them. After moving walls or floors, right-click the level and choose **NodeWorthy** → **Bake Navigation Mesh** to bake it from the level as it is now. It is offered in a level where something finds its way. One undo puts the last one back (10.21).

Paint With My Own Tiles. To paint a 2D level from a picture of tiles, right-click the level's top node and choose **NodeWorthy** → **Paint With My Own Tiles** → **Create TileSet From Image...**,

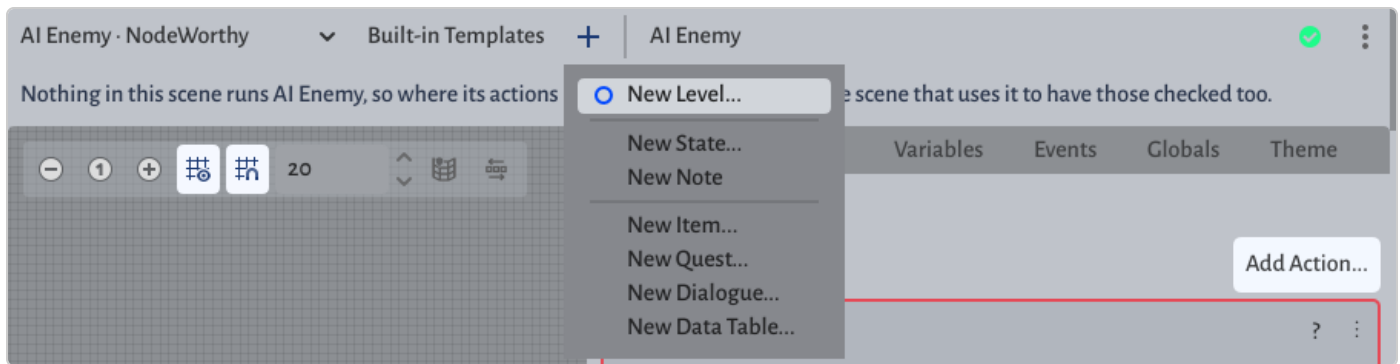
then pick the picture anywhere on this computer. Say how many tiles **Across** and **Down** (it guesses), and whether they are **Seen from above** (squares) or **Seen at an angle (isometric)** (diamonds half as tall as they are wide). Then click each tile nothing can walk through, such as walls, rocks, trees and water. Click one again to let things through. Press **Use It** and the level gets two layers to paint on with Godot's TileMap panel at the bottom of the screen. **Floor** is drawn under everything. **Walls And Things** is where a character walking in front of a wall is drawn over it, and one behind it is drawn under it. The tiles you clicked block the way on the World layer, like any wall, and a torch that casts shadows throws them from those tiles.

Ground, and several pictures. Seen at an angle, the tiles that are ground to walk on are outlined green, such as a block whose top is the floor, or something flat like a bridge. Everything else stands on the ground. NodeWorthy guesses which is which from the pictures. To change one, pick **Clicking marks ground to walk on** and click it. Choose **Create TileSet From Image...** again with the same picture and the window opens as you left it, keeping everything painted. Choose it with another picture and its tiles join the ones the level has, listed down the side of the TileMap panel. The pictures must be cut to tiles of the same size, seen the same way, or NodeWorthy says why not.

Houses, windows and chimneys. Once the level has tiles, the same menu adds layers over them. **On The Walls** is for what hangs on a wall in the same spot as the wall, such as a window, a door or a sign. **A Level Up** is drawn a wall's height higher, for roofs and upper floors. Choose it again for a second level up. What is painted on a level up stands on the ground under it. That ground is solid, enemies finding their way go round it, and a character in front of the house is drawn over it while one behind it is drawn under it. **On The Roof** sits half a wall above the roofs, for a chimney. Paint it on the back of a roof and the front of the roof hides its foot. A layer that cannot be added yet is greyed out, and pointing at it says why. Emberdeep's village, `emberdeep/levels/outdoors.tscn`, is painted this way.

Collision Layer. Right-click a body or an area and choose **NodeWorthy** → **Collision Layer**, then a layer, such as **World**, **Player**, **Enemies**, **Pickups**, or one you named yourself. **Name a new layer...** at the bottom makes a new layer, such as *Breakable*, and puts it there in one go. **Rename a Layer** changes the name of one of yours, and everything on it stays on it. On Enemies, an enemy you made yourself is found by a Raycast that looks for enemies. On one of the four it also bumps into, or notices, what such things do (2.6). One undo puts it back.

4.4 The graph, button by button



The toolbar, with the + menu open. The tick at the right means the template has no problems.

Control	Does
Template list	At the left, it chooses which template is shown. It lists the templates you make , wherever they were saved (two with the same name show their folder after it, as in <i>Player Walks · Coin Counter</i>), and New Template... at the bottom makes another, empty, started from a copy of a ready-made one, or a copy of any other template, your own or one from the recipes and games, to make a variation. Its window says where the template is saved, and Folder... beside it picks another folder for it and the templates after it. The templates that come with NodeWorthy are not in this list, because there are more than a hundred of them and your own would be lost among them. They are one button along, under Built-in Templates. One of theirs being looked at joins the list while it is on screen, with where it came from after its name. Selecting a Machine in the Scene dock, or the node it drives, shows that machine's template either way. A finished build of your game, or another game, kept in a folder inside this one is left out too, because it is a separate game, with its own copy of everything, and its templates belong to it.
Built-in Templates	Beside the list, it shows everything NodeWorthy brings with it in one window. That is the twelve ready-made behaviours (4.3), every recipe (chapter 9) and the templates of the games (the five in chapter 8, and Mossbrook and Emberdeep in chapter 11), grouped by where they come from, with a search box. Pick one and it tells you what it does and what is in it. Show It puts it in the graph to look through. Make A Copy gives you a copy under a name you choose, and leaves the one it came from exactly as it is. That is the safer way in, because a template shown in the graph can be edited like any other, and these are the ones the games and the recipes run on.
+	New State... adds a state at the level on screen. The first one becomes the start. New Note puts a note behind the states. Drag states onto it to make it a group that holds them. At the top, New Level... makes a whole level in one window, blank or ready to play (5.1). Below them, New Item... , New Quest... , New Dialogue... and New Data Table... make a new file for your game and open it, an item or a quest in the Inspector and a dialogue or a table here in the panel. Each window says where the file is saved, at first <code>items</code> , <code>quests</code> , <code>conversations</code> or <code>data</code> , and Folder... beside it picks another folder, which the next file of that kind goes in too (chapter 11).
Breadcrumb	Beside them, it shows which level you are looking at. Click a name, or Up , to go back out of a parent state.

Problems	A tick when the template is sound, or a warning with a count of what is broken. Press it to list them. Broken rows are red in the graph as well. Everything it looks for is in 7.2.
Debug, Pause, Step	Only while the game runs. Debug picks a machine so its state lights up as it changes. Pause and Step hold it still and move it on one frame at a time. How to use them is in 7.1.
– 1 +	At the top left of the graph. – and + zoom out and in. 1 fits the whole template in view. It centres every box on the level you are looking at and zooms until they fill the graph to its edges. A template fits itself to the view when it comes on screen (when you click a Machine in the Scene dock, or pick another template from the list at the top), and you always start by seeing the whole of it.
:	Auto Layout arranges the boxes. It puts the start at the top and each state in a row below the states that lead to it, with every row centred and put in the order that crosses the fewest arrows. It is also the grid button beside the zoom buttons at the top left of the graph. Undo puts them back.
Saving	There is no Save button. The template and its state files are written a moment after every change, and again when the game runs, the scene is saved or the editor closes. Ctrl+Z still undoes every edit. Messages, such as a template being created, appear in the editor's notifications at the bottom right.
Actions / Transitions / Variables / Events / Globals / Theme	The six tabs on the right show what the state you are working on does, the ways out of it, the values the machine carries, the events it sends and waits for, every global variable and global event in the project (6.7), and how the game's menus and panels look (6.14). Clicking a state's box in the graph brings Actions forward, and clicking one of its transition rows brings Transitions forward with that row outlined.

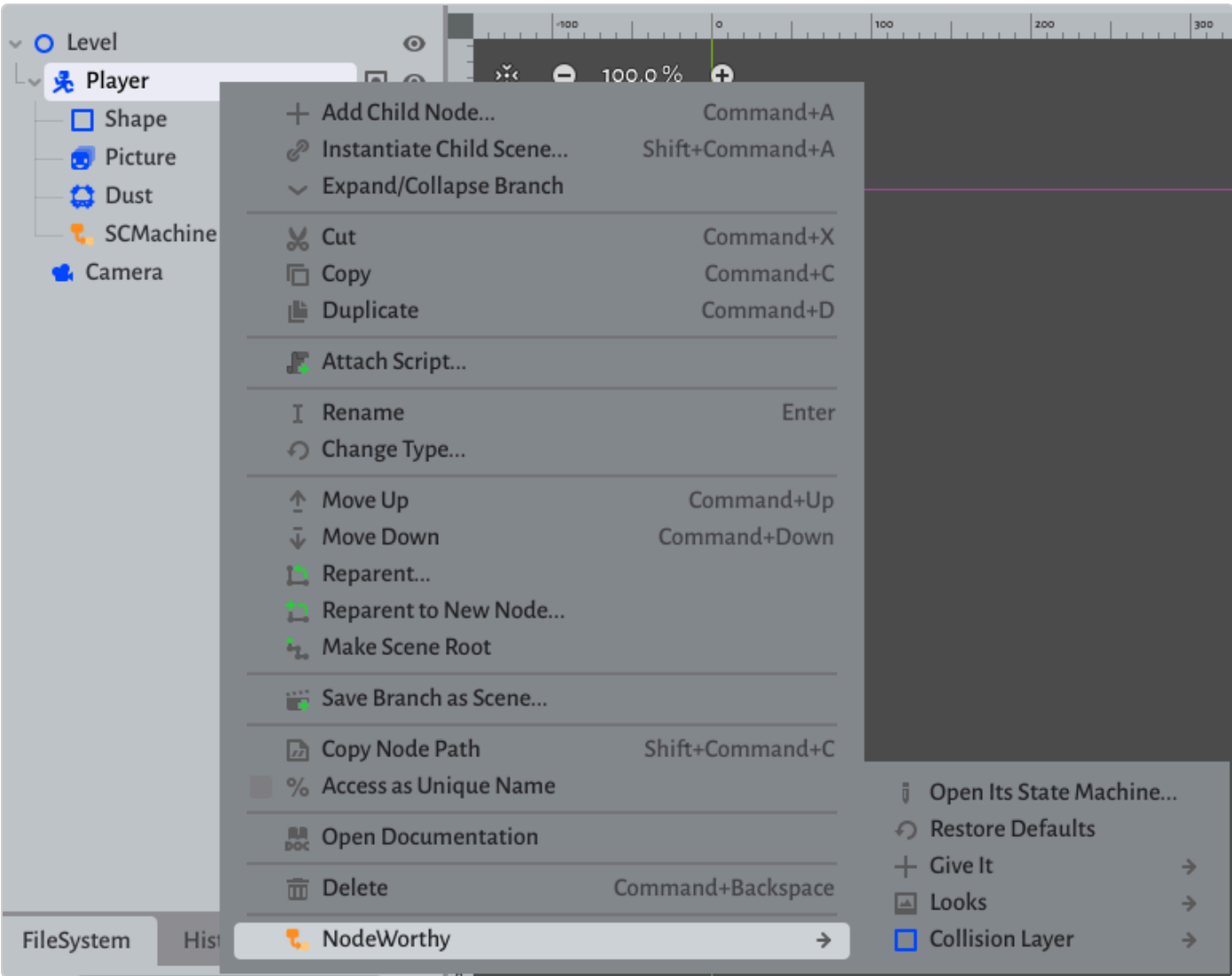
Right-click a state for Set as Start, Add Action..., Add Transition, Add Sub State or Add Parallel State (only the one that fits what the state already holds), Open Sub States, Open Parallel State, Color, Mark as End State and Delete State. Pick several boxes and press Delete to take them all away at once. One Undo brings them all back. Right-click a transition row for Delete Transition. Right-click empty space for New State and New Note or Group, and right-click a note for Empty This Group and Delete Note. Click anything to work on it in the panel and the Inspector.

Right-clicking a state also offers **Copy**, **Duplicate** and **Add Breakpoint**, described in 6.2 and 7.1.

4.5 The right-click menu, item by item

Most of what NodeWorthy puts into a scene comes from one place. Right-click a node in the **Scene dock** and point at **NodeWorthy**. It offers only what suits the kind of node you right-clicked. The level (the top node, or a plain Node2D or Node3D that holds part of it, such as an Enemies node) gets **Add Player**, **Add Thing**, **Add To Screen** and **Add Menu**. A character or a thing (a body, an area, or anything running a template) gets **Open Its State Machine...**, **Give It**, **Looks** and **Collision Layer**. A Machine gets **New Template for This Machine...**. Anything else you can see, hear or place, such as a sprite, a mesh, a light, a camera, a label or the WorldEnvironment, gets **Add State Machine...** and nothing more, because a machine is the one thing it can use. A

screen layer (a CanvasLayer) gets **Add To Screen**. A part of something else, such as a collision shape, an AnimationPlayer, a Timer or one of NodeWorthy's own nodes, gets a single greyed line saying what it is a part of and where to right-click instead. Hover any item to see what it does, and one that is greyed out says why.



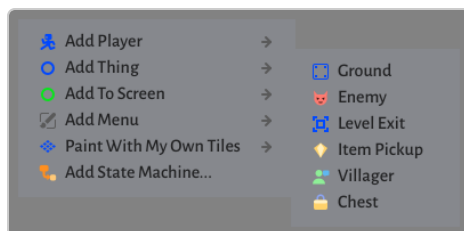
Right-click a node in the Scene dock, here the Player, and point at **NodeWorthy** at the bottom of Godot's menu.

Changing a ready-made thing. Every ready-made player and thing works the moment it is added, and its settings at the top of the Inspector (click it in the Scene dock) change the usual things, such as speed, damage and who it attacks. To go further, open its template in the NodeWorthy panel and add your own states and actions.

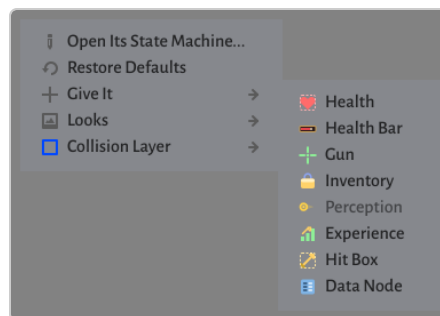
Item	Where it shows	What it does
Start A 2D Level..., Start A 3D Level...	The top of an empty scene	Opens the New Level window (5.1) for this scene, with the kind clicked picked. Ground, camera and light give a wide Ground and a Camera, with a Sun and a Sky in 3D or a soft sky-coloured Background in 2D, and the rest of the window adds a player, things to meet, the screen and the menus. Add

		Player later puts a hero on the Ground and hands it the Camera. Offered only while the scene is empty, and only the kind its top node can hold.
Add Player	The level	Platform Hero (2D), Platform Hero (3D), Side-On Platform Hero (3D), Top-Down Hero (2D), Top-Down Hero (3D) or Third Person Hero (3D), only those that suit the scene. Each comes with a body, a picture, a camera and a template of your own (4.3).
Add Thing	The level	Ground, Terrain, Model..., Enemy, Level Exit, Item Pickup, Villager and Chest, each working the moment it is added (4.3). Terrain and Model... are offered in a 3D level, Terrain once (6.21).
Add To Screen	The level	Stat Panel, Bar, Globe, Icon Meter, Inventory Grid, Quest List, Quest Log, Minimap, Crosshair, Skill Bar, Dialogue Box and Item Collection, put on the HUD and each described in 10.35. A screen needs only one Minimap, Quest List, Quest Log, Dialogue Box or Crosshair. Once the scene has one, it is greyed out and says where the one it has is. The same goes for a Pause Menu under Add Menu.
Add Menu	The level	Main Menu and Pause Menu, over the whole screen (4.3).
Paint With My Own Tiles	A 2D level	Create TileSet From Image... turns a picture of tiles into a Floor and a Walls And Things layer to paint on. On The Walls, A Level Up and On The Roof add layers for windows, roofs and chimneys (4.3).
Add State Machine...	The level, a character without a template, or anything else that can run one	Gives the node a Machine and a new template to build in the panel. Chapter 5 walks through it. Its Start from list greys out a ready-made player that does not suit the node, such as a 2D player on a 3D body, and says why.
Open Its State Machine...	Anything running a template	Picks its Machine and opens its template in the NodeWorthy panel.
New Template for This Machine...	A Machine	A separate template for this Machine, in place of the one it runs.
Behave Like	Anything that already has a template, when there is another of its kind	Makes the node act like something already in your game, by giving it the same template to think with, so five guards can share one template that is changed once for all of them. A ready-made character is offered only templates of its own kind, enemies for an enemy and players for a player, 2D or 3D as it is, each named with the level it is used in, and the one it uses now ticked. With nothing else of its kind, it is not offered, and something with no template yet, such as the HUD or a sprite, is given one with Add State Machine... instead. A template of your own is offered your other templates of your own. A template that cannot work on the node is greyed out, and resting the pointer on it says why, such as no AnimationPlayer to play clips

		from, or a walking enemy's template offered to a camera or a marker. Give the node what it needs and right-click it again. A template that only needs a Perception node or a NavigationAgent is not greyed, as Behave Like adds those to a 2D or 3D node by itself. Afterwards a message says what changed. Undo puts back what it did before.
Restore Defaults	A ready-made player, thing or menu	Puts its template back as it came with NodeWorthy, keeping its name. Health it was given stays. Undo brings your changes back.
Give It	A character or a thing, or the level once it runs a template	Health (a Hurt Box, a health of 3 and a Dead state, and a state machine first if it has none), Health Bar (a small bar over its head that shows its health, with health given too if it has none), Gun (for a character, the left mouse button or the right trigger on a pad fires the way it faces), and the Inventory, Perception, Experience, Hit Box and Data Node that a character carries (4.3, 10.35). Health, Perception and a Hit Box are offered only to what can fight, a body, an area or another thing in the 2D or 3D world. A camera, a light or a label running a template is offered none of them.
Looks	A character or a thing with a plain shape	Set Texture... and Import Sprite Sheet... in 2D, Set Model... , Set Model Textures... and Use Animations From... (Experimental) in 3D. They put your own picture, animations or model in place of the plain shape, give each part of a model the textures from its pack, or give a 3D character the animations of another pack, which is experimental (4.3).
Collision Layer	A body or an area	Sets which named layer it is on (World, Player, Enemies, Pickups or one of your own), and names or renames layers (2.6).
Bake Navigation Mesh	The level, where something finds its way	Bakes the ground Navigate To finds its way on again from the level's walls and floors (10.21).



All of it, for a 2D level.



A character gets its own items at the top, for its picture, sprite sheet, a gun, health and its layer.

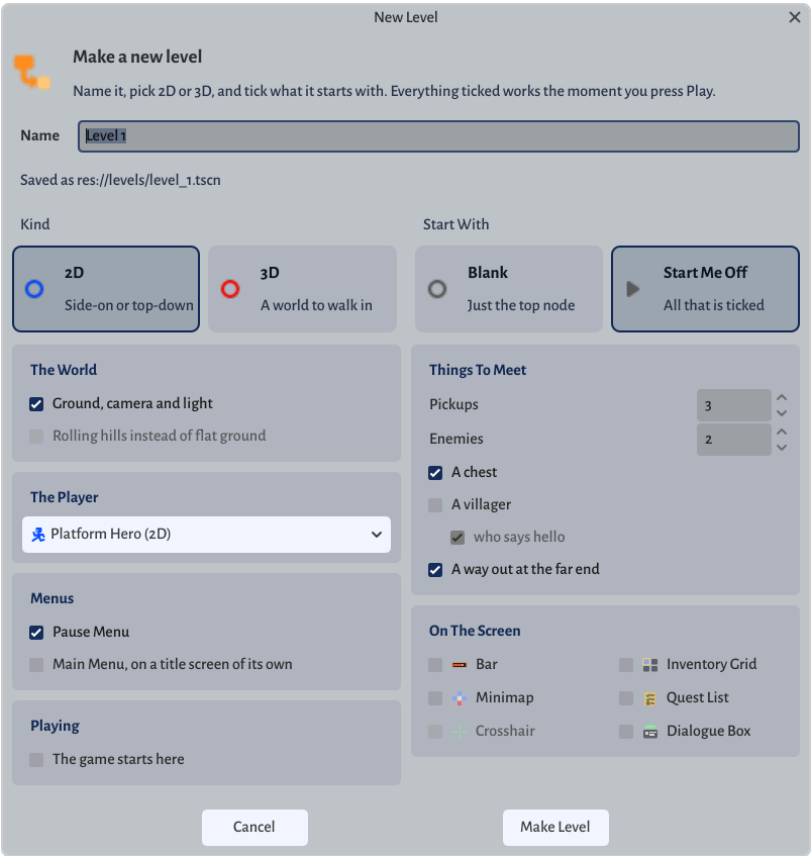
5. Your first machine

This chapter starts with a level made of ready-made things, then changes how its chest opens. If you have the NodeWorthy app and would rather start from something that already moves, 5.4 does the same with the ball in your new game. It takes about five minutes and uses every part of chapter 3 except sub states and parallel states.

5.1 A level you can play, in five minutes

NodeWorthy comes with ready-made things, each working the moment it is added, such as a player, the ground, a chest and an enemy. Start with those, then look inside them.

The quickest way is one window. In the NodeWorthy panel press **+** and choose **New Level...**, or right-click a folder in the FileSystem dock and choose **Create New → NodeWorthy Level...** Give the level a name, pick **2D** or **3D**, and choose **Blank** for nothing but its top node or **Start Me Off** for everything ticked below it.



The New Level window. Everything ticked works the moment you press Play.

Part	What it puts in the level
The World	Ground, camera and light gives a wide Ground, a Camera, and a Sun and Sky in 3D or a soft sky in 2D. In 3D, Rolling hills puts a Terrain of hills in place of the flat Ground (6.21). With the

Part	What it puts in the level
	Top-Down Hero picked, it is a room seen from above instead, a tiled floor with stone walls round it, and everything is put about the room.
The Player	One of the ready-made players that suit 2D or 3D, standing at the start of the ground with the camera following them, or No player.
Things To Meet	Pickups ahead of the player, each a blue gem handing out a Gem item made once in the folder new items go in, at first <code>items</code> . Enemies further along, and the player is given health so they can be hurt. A chest and a villager behind the player. who says hello gives the villager a dialogue of its own in <code>conversations</code> , greeting the player by the level's name with two answers to pick from (6.15). A way out at the far end is a Level Exit that leads to the title screen when there is one, or else starts the level again.
On The Screen	Any of a Bar, an Inventory Grid, a Minimap, a Quest List, a Crosshair (3D) and a Dialogue Box, put on the screen as Add To Screen does. A Minimap comes with its picture of the level already taken, so the ground shows under its marks from the first Play. The player's health and gold show by themselves once there are enemies to hurt them.
Menus	A Pause Menu , and a Main Menu on a title screen of its own, saved beside the level, whose Start leads to the level.
Playing	The game starts here makes Play open on this level, or on its title screen when there is one.

Making a level takes a few seconds. Meanwhile a small window says each step as it is done, laying the ground, adding the player, each enemy and so on, with a bar that fills as it goes, and it goes away when the level is open.

Press **Make Level** and the level is saved where the window says, at first your `levels` folder or the folder you right-clicked, and opened. **Folder...** beside it picks another folder, and the next level goes there too. Everything is laid out along the ground as a first walk through it. The window keeps what you ticked, so the next level is one click. Everything in it is an ordinary ready-made thing, so you can move it, change it in the Inspector, or add more with the right-click menu. The steps below build the same kind of level piece by piece, which shows where each part comes from.

1. Make sure NodeWorthy is turned on (see 4.1).
2. Choose **Scene** → **New Scene**. In the **Scene** dock press **2D Scene**. Save it with **Scene** → **Save Scene**, as `level.tscn`.
3. Right-click the top node in the Scene dock and choose **NodeWorthy** → **Start A 2D Level...**. The New Level window opens for this scene. Untick everything but **Ground, camera and light**, set Pickups and Enemies to 0, pick **No player** and press **Start Level**. A long green floor, a camera and a soft sky appear.
4. Right-click the top node again and choose **NodeWorthy** → **Add Player** → **Platform Hero (2D)**. A little helmeted hero lands on the ground, and the camera follows it.

5. Right-click the top node and choose **NodeWorthy** → **Add Thing** → **Chest**, then **NodeWorthy** → **Add Thing** → **Enemy**. Each lands to the right of the player. Drag them wherever you like in the 2D view.
6. Press **F6**, or the **Run Current Scene** button at the top right. Walk with **A** and **D**, jump with **Space**. Walk up to the chest and "E to open" shows over it. Press **E** and it opens and spills its loot. The enemy, a little blue blob, comes after you when it sees you.

Nothing was typed and nothing needed the Inspector. Click any of them in the Scene dock and the Inspector shows its settings at the top, above Godot's own, in sections that fold, such as how likely the chest is to hold an item and how much gold. Press **Open Its State Machine...** under them and the NodeWorthy panel shows its template. The Chest's is four states, **Closed**, **Player Near**, **Opening** and **Open**. When the player comes near, Closed moves to Player Near, which shows the words and waits for E. Its **Event On Input** action sends the event `Open` when E is pressed, and the arrow that reads "when E is pressed → Opening" takes it.

5.2 Your first change, a chest that opens when you walk into it

1. Click the **Chest** in the Scene dock, and under its settings at the top of the Inspector press **Open Its State Machine....** The NodeWorthy panel shows its template.
2. Click the **Player Near** box. The **Actions** tab on the right lists its actions. On **Event On Input**, press `:` and choose **Remove**, then do the same on **Set Visible**, the one that shows the words. Now nothing opens the chest, and "E to open" no longer shows.
3. Click the arrow's row at the bottom of the Player Near box, "on Open → Opening". The **Transitions** tab comes forward, and under **When** it says *Nothing in this scene sends Open yet. Press When and pick what makes it happen.*
4. Press **When**, the button above it, and pick **When the player touches it**. An **Event On Collision** action is added to Player Near, sending the event `PlayerTouched`, and the arrow now reads "when the player touches it → Opening".
5. Run the scene again and walk into the chest.

That is the whole idea of NodeWorthy. It comes down to **states** (Closed, Player Near, Open), **actions** that run in them (Event On Collision), **events** (PlayerTouched) and the **transitions** that wait for them. Build your own from scratch with **NodeWorthy** → **Add State Machine...** on any node. It asks for a name, then gives the node a Machine and an empty template to draw. **Start From** in that window copies a ready-made behaviour (4.3) for you to change as you like.

5.3 Where to go next

You now know enough to try the recipes in chapter 9. Open one, run it, and click its Machine to see how it is built. When something in a recipe is new to you, chapter 6 explains it. When a machine you built does not do what you expect, chapter 7 shows how to watch it run and find out why. Chapter 8 then builds bigger things, one step at a time.

5.4 With the NodeWorthy app, a hop on the Space bar in five minutes

If you made an empty game with the NodeWorthy app (1.1), you already have a machine to practise on. It is the ball's template, **Ball**, with its one state, **Hop**. This makes the ball wait for the Space bar before each hop. It uses the same moves as the chest, with a second state, a way to it and a way back.

1. Look at the **NodeWorthy** tab. It shows the **Start here** note and the Hop box. If it shows anything else, click **SCMachine** under **Ball** in the Scene dock.
2. Press **+** in the panel's toolbar and choose **New State....** Type **Waiting** and press **Create**. A **Waiting** box appears on the graph. If it lands on top of another box, drag it clear.
3. Now the question that leads from one state to the other. Click the **Waiting** box, press **Add Action...**, open the **Input** group, pick **Input** (marked *check*) and press **Add**. Beside **Input Action** press **▼** and pick **Key: Space**. Set **When** to **Just Pressed**. In **True Event** type **Go**. The new action's line in the Actions tab ends "Key: Space just pressed → Go".
4. The problems button in the toolbar now shows a warning. Press it. It says that nothing in this template waits for the event Go, and that nothing leads to Waiting. Both are true, and the next two steps mend them.
5. Right-click **Waiting**, choose **Add Transition**, and drag the new row's port onto **Hop**. Click the row, then on the Transitions tab press **When** and pick **Go**, under *This template's events*. The row reads "on Go → Hop".
6. The way back is already half there. Hop has one row, "when finished → Hop", which is what starts it again. Drag that row's port onto **Waiting** instead. It now reads "when finished → Waiting".
7. The problems button shows a tick again. Press **Ctrl+S** (**Cmd+S** on a Mac) to save, and run the game. The ball hops once and then waits. Press Space and it hops again. Under **Debug**, pick the machine and you can see Hop and Waiting light up in turn.

Hop still counts a second and a half before it goes back to Waiting, so a press during that time is missed. Click the Hop box, open **Wait** in the Actions tab and change **Seconds** to 0.5 for a livelier ball.

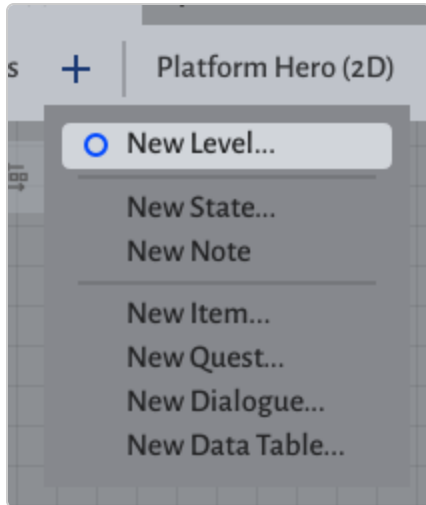
Good to know. That is the shape of nearly every character you will make. It has one state for each thing it can be doing, a Check in each state asking whether it is time to leave, and a transition for each answer. Jumping, crouching, being hurt and being dead are more boxes of the same kind.

5.5 An item the player picks up, step by step

This and the sections after it build one small level, a step at a time, with nothing typed into a template. You need a level with a player and ground. In the Scene dock (top left), right-click the

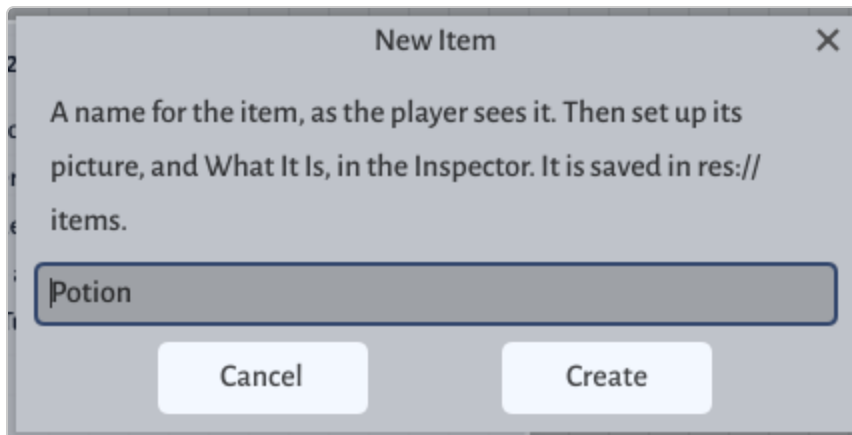
top node and choose **NodeWorthy > Add Player > Platform Hero (2D)**, then **NodeWorthy > Add Thing > Ground** (5.1 shows it). Things in your game that are files (items, quests, dialogues, tables) are made from the NodeWorthy panel, and set up in the Inspector on the right, like any Godot node or file.

1. In the NodeWorthy panel, press **+** in its toolbar and choose **New Item...**



The + menu makes states and notes for the template on screen, and new items, quests, dialogues and tables for your game.

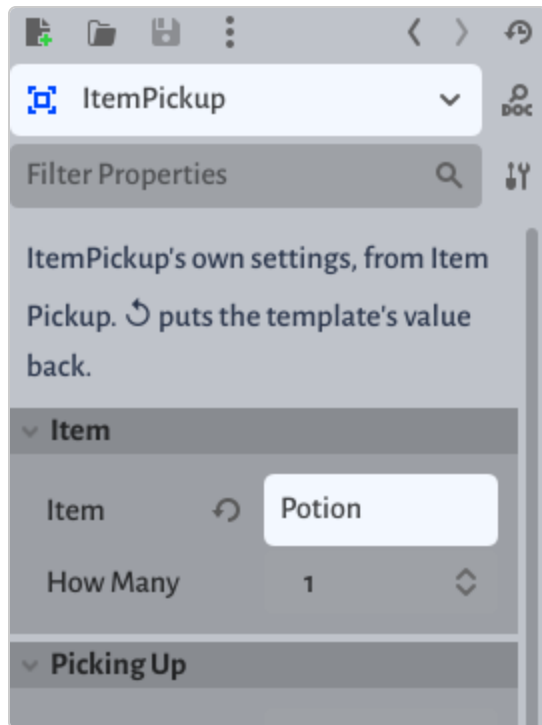
2. Type a name, **Potion**, and press **Create**. The item is made in your project's `items` folder, as the window says (you can see it in the FileSystem dock, bottom left), and opens in the Inspector. **Folder...** in the window picks another folder, and the next item goes there too.



Every new file asks for its name first.

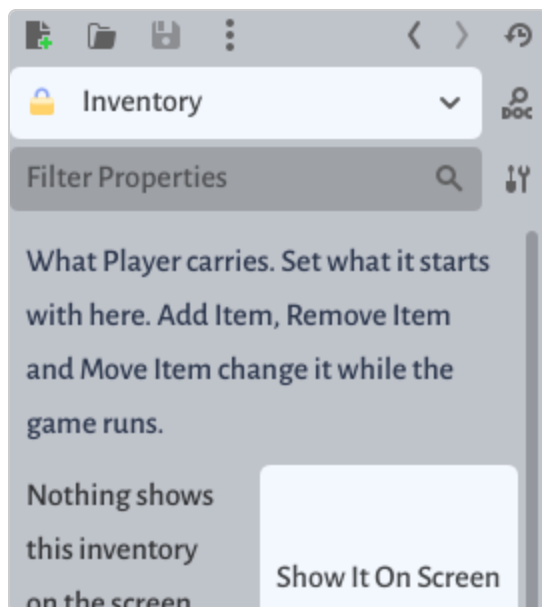
3. In the Inspector, give it a **Description** (what the player reads about it) and an **Icon** (press the Icon box and choose **Pick a picture...** for one already in your project, or **From my computer...** for one anywhere on this computer, which is copied into your project). Leave **What It Is** at **Carried** for now, because 5.6 makes it a potion the player drinks.
4. Put it in the level. Right-click the top node in the Scene dock and choose **NodeWorthy > Add Thing > Item Pickup**. It lands beside the player. Drag it where you want it in the 2D view.

5. Click **ItemPickup** in the Scene dock. At the top of the Inspector, under **Item**, press the box that says **Pick an item....** Choose **Pick an item...** in the menu it opens, then **potion.tres** in the list of your project's items.



A ready-made thing's own settings sit at the top of the Inspector, above Godot's.

6. The player needs something to carry it in, and already has it. Adding the Item Pickup gave the player an **Inventory** (look under **Player** in the Scene dock). Click it. The top of the Inspector says nothing shows it on the screen yet, so press **Show It On Screen**.

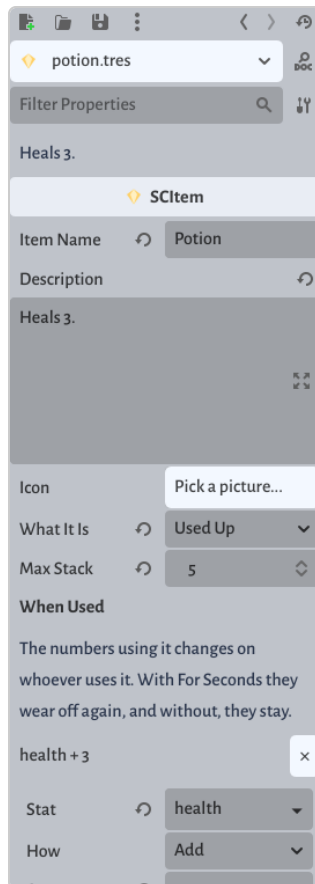


An Inventory says what it is for, and offers to show itself.

7. An **Inventory Grid** goes on the screen, in the bottom left corner. Press **Run Current Scene** (top right, or F6) and walk onto the pickup. The potion appears in the grid.

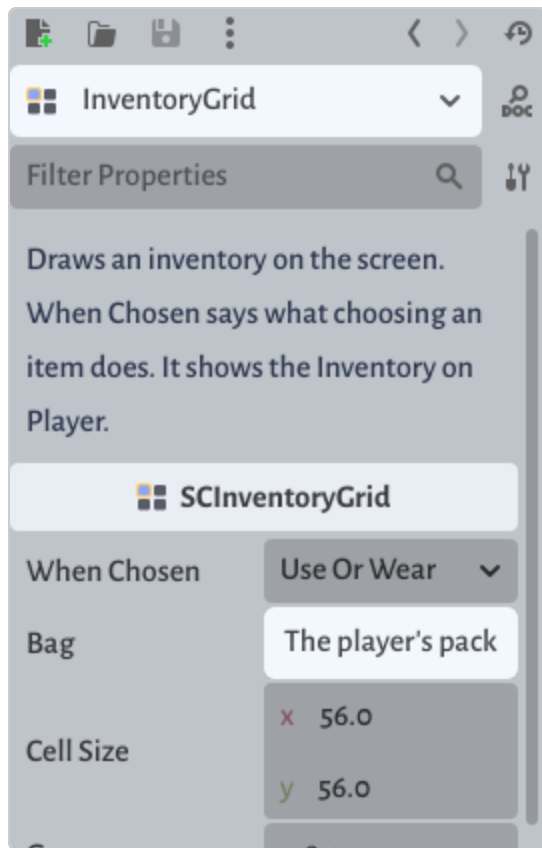
5.6 A potion the player drinks, step by step

1. The potion will heal, so the player needs health. Right-click **Player** in the Scene dock and choose **NodeWorthy > Give It > Health**. The player gets a health of 3, shown on the screen. (A player that already has health, such as one given it by an Enemy, skips this step.)
2. Click **potion.tres** in the FileSystem dock. In the Inspector set **What It Is** to **Used Up**, and the settings of something used once and gone appear. Under **When Used** press **Add Stat Change**, pick **health** in its **Stat** list, and set **Amount** to 3. Set **Max Stack** to 5 so five potions share one slot.



What It Is decides which settings the item has. A potion has When Used.

3. Click **InventoryGrid** (under **HUD** in the Scene dock). Its **When Chosen** is **Use Or Wear**, which means choosing an item in the grid uses it.



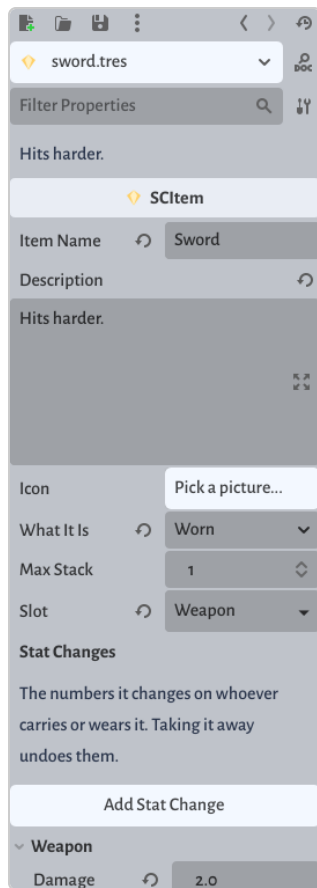
With Use Or Wear, the grid drinks potions and puts on swords by itself.

4. Run the scene and pick up the potion. A potion only heals what has been lost, because health never goes above max_health, so let something hurt the player first, such as an Enemy (**Add Thing > Enemy**). Then click the potion in the grid twice (the first click moves the highlight to it, the second chooses it), or move to it with the arrow keys and press Enter. The player's health goes up by 3, as far as max_health, and the potion is gone from the grid.

Good to know. To have a key drink a potion instead, use the **Use Item** action in the player's template (10.32). Nothing in this section needed a template at all.

5.7 A sword the player wears, step by step

1. In the NodeWorthy panel press **+ > New Item...** and call it **Sword**.
2. In the Inspector set **What It Is** to **Worn**. **Slot** appears, already set to **Weapon**, the slot it is worn in (the list offers Armour, Trinket and the slots your game uses, or type your own). Under **Weapon**, set **Damage** to 2. This is how hard it hits with the **Attack** action (6.17). Anything under **Stat Changes** is added to the wearer's numbers while it is worn, and taken off again when it comes off.



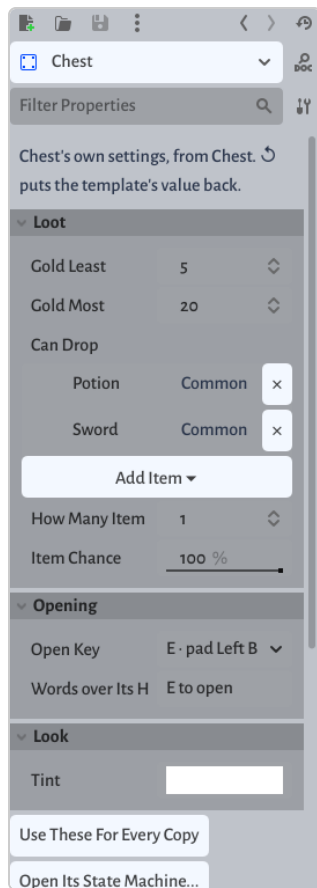
A worn item has a slot, and a weapon's numbers.

3. Add a second **Item Pickup** and pick the sword as its Item, as in 5.5.
4. Run the scene, pick up the sword and choose it in the grid. It leaves the pack and is worn. The first thing the player wears makes them a **Worn** Inventory to wear it in.

Good to know. To see what is worn, and take it off again, give the player an Inventory for it before the game starts. Right-click **Player > NodeWorthy > Give It > Inventory**, and in the Inspector, under **Worn**, press **Add Slot** under **Slot Names** and pick **Weapon** in its list. Press **Show It On Screen** on it too, and its grid goes just above the pack's. Choosing the sword in that grid takes it off, back into the pack.

5.8 A chest with loot, step by step

1. Right-click the top node in the Scene dock and choose **NodeWorthy > Add Thing > Chest**.
2. Click the Chest. At the top of the Inspector, under **Loot**, press **Add Item** under **Can Drop** and pick **Potion**, which your own items list first. Add the sword the same way. Each item is a row with its picture, its name and how rare it is, and **X** takes it out. **How Many Items** is how many it takes from the list as the level starts (rarer ones less often, and you set an item's **Rarity** under its **Extras**), and **Item Chance** how likely each of them is really there (100 is always). **Gold Least** and **Gold Most** are the gold.



Each chest in a level can drop different things.

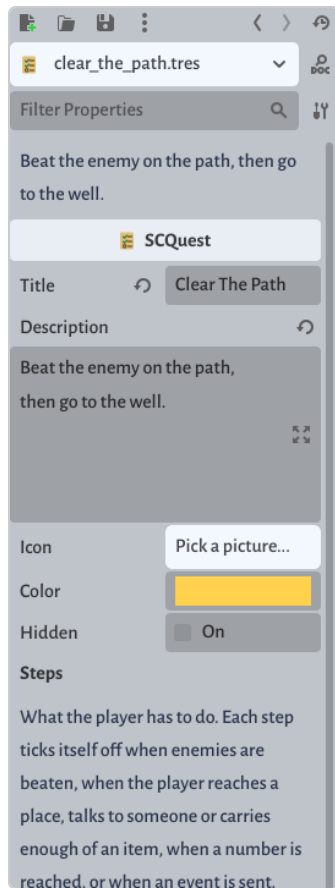
3. Run the scene, walk up to the chest and press **E**. It opens and spills its loot and gold. Walk over them to pick them up. A ready-made **Enemy** has the same Loot settings, for what it drops when beaten.

5.9 A quest from a villager, step by step

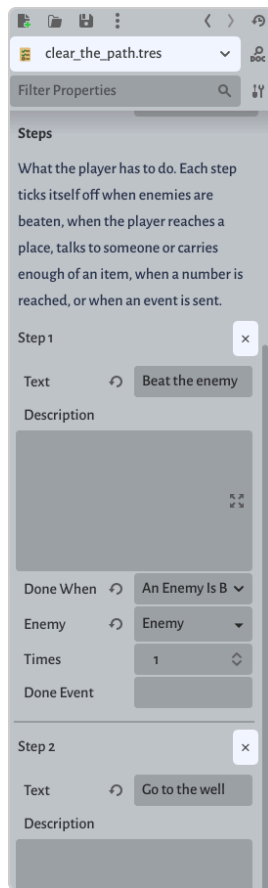
A quest the farmer gives. The player beats the enemy on the path, then goes to the well, and gets a potion as the reward.

1. Something to beat and somewhere to go. Right-click the top node and choose **NodeWorthy > Add Thing > Enemy**. Then right-click the top node again, choose **Add Child Node**, type *Marker2D*, press **Create**, double-click the new node's name and call it **Well**, and drag it in the 2D view to where the well is. To beat the enemy the player needs a way to hurt it, so right-click **Player > NodeWorthy > Give It > Gun** (the left mouse button fires).
2. In the NodeWorthy panel press **+ > New Quest...** and call it **Clear The Path**. It opens in the Inspector with a first step.
3. Give it a **Description**. In the first step, set **Text** to *Beat the enemy* (what the player reads), leave **Done When** at **An Enemy Is Beaten**, and pick **Enemy** in its **Enemy** list (the list offers the kinds of enemy in your level, and empty means any). **Times** is how many.
4. Press **Add Step**. Set its Text to *Go to the well*, its Done When to **The Player Reaches A Place**, and pick **Well** in its **Place** list. Tick **In Order** so the well only counts once the enemy

is beaten. Under **Reward**, add the potion to **Reward Items**.

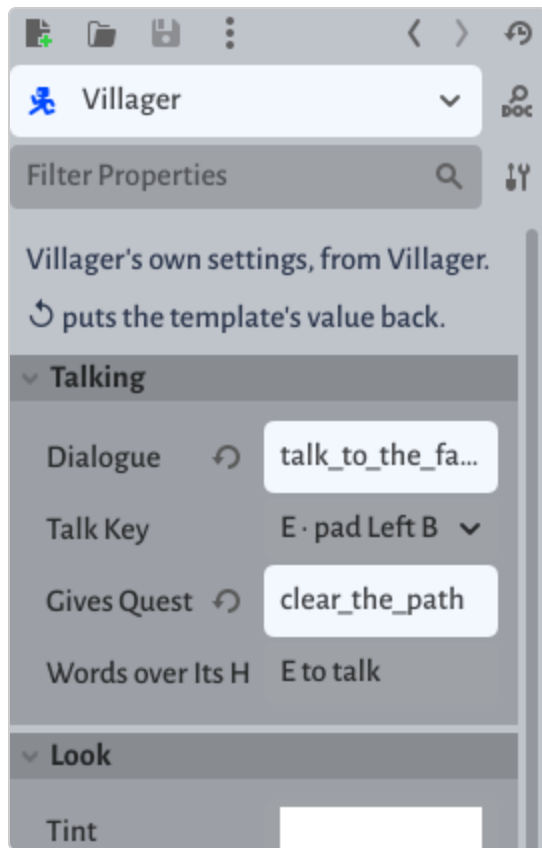


A quest opens in the Inspector with its title and description first and its steps below.



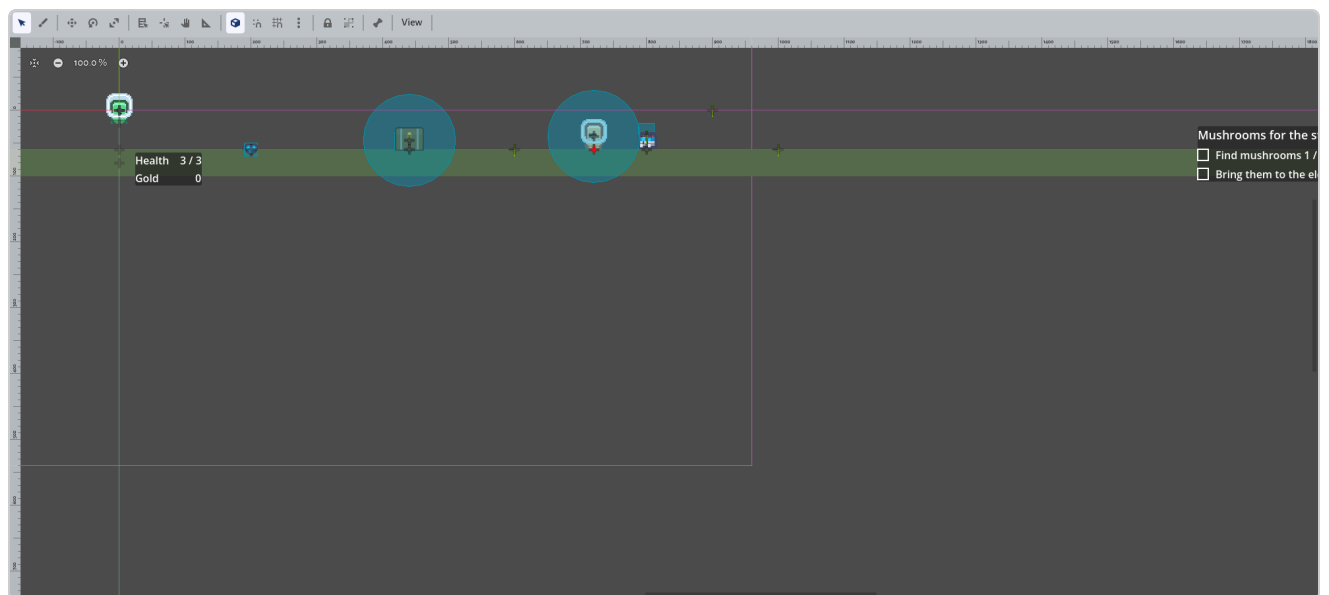
Each step says how the game knows it is done.

5. What the farmer says. Press **+ > New Dialogue...** and call it **Talk To The Farmer**. It opens as a graph in the NodeWorthy panel, with one line said by Somebody. Click that line, and at the side type *Farmer* in **Speaker** and what the farmer says in **Text**, such as *Wolves on the path again. Clear them, and meet me at the well.* (6.15 says more about dialogues.)
6. Right-click the top node and choose **NodeWorthy > Add Thing > Villager**. Click the Villager. At the top of the Inspector, under **Talking**, pick **talk_to_the_farmer.tres** in **Dialogue** and **clear_the_path.tres** in **Gives Quest**. A **Quest List** goes on the screen by itself, in the top right corner.



A villager says its dialogue, then gives its quest.

- Run the scene. Walk up to the farmer and press **E**, then Enter to read on. When the talk ends, *Clear The Path* appears in the top right with its steps. Shoot the enemy and the first step is ticked. Walk to the well. That was the last step, so the quest is done. It leaves the list, and the potion goes into the pack.



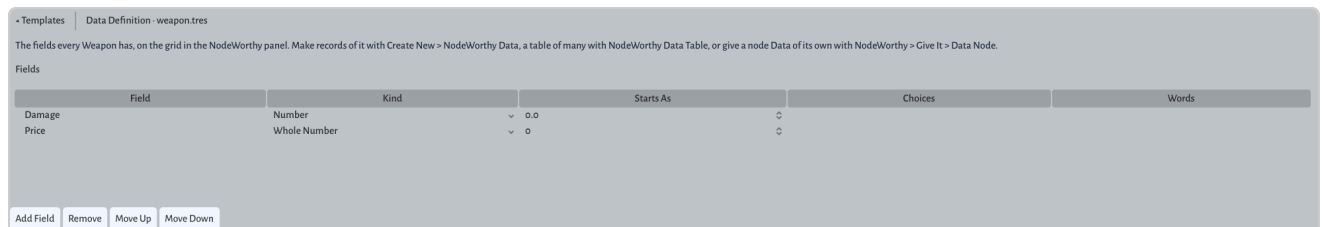
The whole level, with nothing typed into a template. The pack and the quests show on the screen while the game runs.

Good to know. A step can also be done when the player talks to someone (**The Player Talks To Someone**, where you pick the dialogue they play), carries enough of an item, or a number reaches a value. **An Event Is Sent** is for anything else a template can say. 6.16 has the whole of quests.

5.10 A table of weapons, step by step

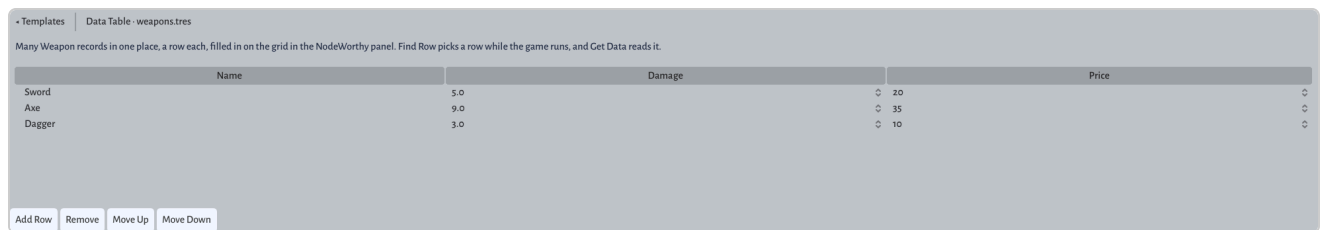
A Data Table keeps many things of one kind in one grid, like a spreadsheet. For example, it can hold every weapon with its damage and its price. It is for games with a lot of them, and a game with three swords can skip it.

1. First say what a weapon has. In the FileSystem dock, right-click a folder of your own and choose **Create New > NodeWorthy Data Definition...**, and call it **Weapon**. Its grid opens in the NodeWorthy panel. Press **Add Field**, name it **Damage** and leave its **Kind** at Number. Add another, **Price**, a Whole Number. Every weapon already has a **Name**.



A Data Definition is the list of what every one of its kind has.

2. Press **+ > New Data Table...** in the NodeWorthy panel and call it **Weapons**. In the Inspector, pick **weapon.tres** as its **Data Definition**. The grid in the panel gets a column for each field.
3. Press **Add Row** for each weapon and type into the cells, for example a Sword with Damage 5 and Price 20 and an Axe with 9 and 35. Every change is saved as you make it, and Undo takes it back.



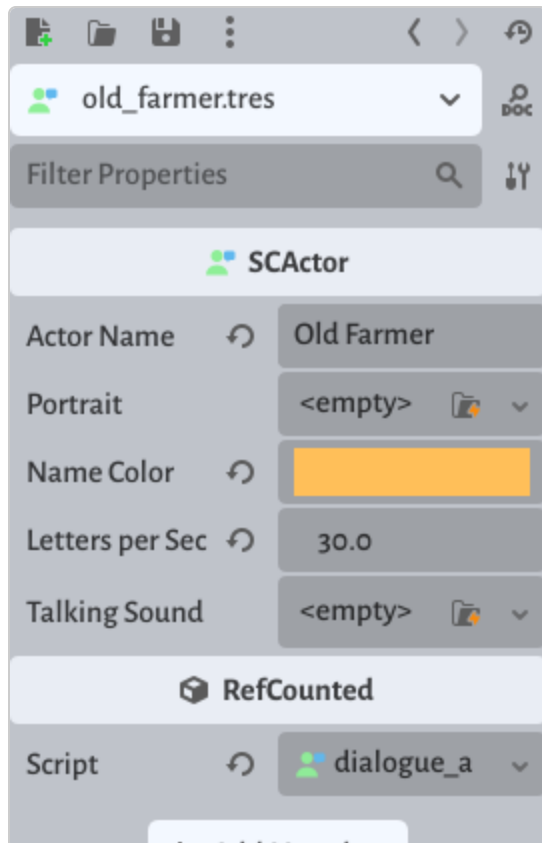
The Weapons table, with one row per weapon.

4. To use it, a template reads it with the **Find Row** and **Get Data** actions. They can pick a row at random and read its Damage into a variable.
`addons/nodeworthy/examples/data/find_row` shows it working, and 6.20 has the whole of Data.

5.11 Someone to talk to, step by step

A farmer the player can talk to, who greets them, answers a question once, and gives a quest. It carries on from 5.9's villager.

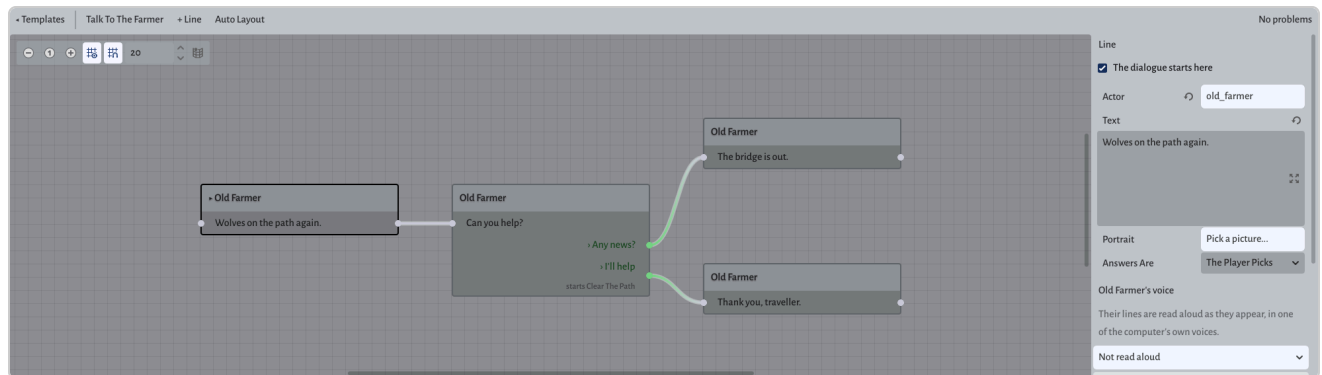
1. Click the Villager. At the top of the Inspector, press **Gives Quest** and choose **None**, because from now on one of the farmer's answers gives the quest. Then press **Dialogue** and choose **Open Talk To The Farmer**. The dialogue 5.9 made opens as a graph in the NodeWorthy panel, with its one line.
2. Who says it. Click the first line's box. At the side, press **Actor** and choose **New actor...**, and call them **Old Farmer**. The Actor opens in the Inspector. Give them a **Portrait**, set **Letters Per Second** to 30 for a slow, steady voice, and pick a **Talking Sound** (any short sound, and **From my computer...** copies one in). Every line you pick them on now shows their name, picture and colour, types at their speed and blips as it goes.



An Actor is made once and picked on every line they say.

3. The first line still has the farmer's words from 5.9 in its **Text**. Press **+ Line** and a second line appears already joined on after the first, said by the same Actor. Type *Can you help?* in its **Text**.
4. An answer. With the second line picked, press **Add Answer** and type *Any news?*. Press the ⚙ beside it and tick **Only Once** so that, once asked, it is not offered again. Add a second answer, *I'll help*, and in its ⚙ pick the quest in **Start Quest**.

5. Where the answers lead. Pick the second line again and press **+ Line**. The new line is joined on from *Any news?*, the first answer leading nowhere yet. Type *The bridge is out*. Pick the second line once more and press **+ Line**, and this time the new line is joined on from *I'll help*. Type *Thank you, traveller*. An answer or a line whose arrow leads nowhere ends the dialogue.



Each answer's arrow leads to the line it brings. + Line joins each new line on for you.

6. Run the scene and talk to the farmer. Ask *Any news?*, then talk again, and the question is gone. Answer *I'll help* and the quest begins.

Good to know. For a greeting that changes, set a line's **Answers Are** to **Picked At Random** and give it three answers, each leading to a different greeting. One is picked each time. The **Dialogue Line Said** check asks whether a line has been heard, for a villager who greets the player differently the second time (6.15).

6. The pieces in depth

This chapter takes each part of a template in turn and says everything about it. It follows the order of chapter 3, with variables and checks added where they are first needed.

6.1 The Machine node

In the Add Node dialog it is **SCMachine**. Every class NodeWorthy declares starts with SC, so none of them can clash with a class of the same name in your game. Add it under the thing it drives. It carries a bright orange icon, so a glance down the Scene dock tells you which nodes in your level are driven by a machine. Its settings are listed below.

Setting	What it does
Template	The template it runs. States and variables are taken from it every time the machine starts, so editing the file changes every machine using it.
Target	The node the actions work on, which they call Self . Empty means the node the Machine sits under. Fill it in only when the actions should work on another node.
Autostart	On by default. Off leaves the machine waiting until something calls <code>start()</code> .
Physics Process	Run the machine in the physics step instead of the frame step. Turn it on when most of what it does moves a body. You rarely need to (see 6.11).
Debug Log	Print every state change and event to the Output panel.
States, Initial State, Variables	Used only when no template is set, for a machine built by hand in the Inspector, and shown only then.
Starting Values	At the bottom, and in the Variables tab for the Machine picked in the Scene dock (see 6.5). It lists each of the template's variables, at the template's starting value. Change one to give this machine its own, so one template can start differently on different nodes, such as a Target template worth 10 points on a small target and 50 on a big one. The arrow beside a changed value puts the template's back. float, int, bool, String, Vector2, Vector3, Color, Array, Dictionary, Node and Resource are offered. A global variable is not, because it is one value for the whole game.

A machine tidies up after itself when its node leaves the scene. Delete the node, or change to another scene, and the state it was in is left properly first. Its listeners stop listening, a slowed-down game gets its speed back and a shaking camera settles. Take the node out and put it back later, and the machine carries on in the state it was in, starting that state's actions again, with its variables as they were.

6.2 States

A box in the graph. Its title bar is green when it is the state the machine starts in, and you can give it any colour from the right-click **Color** menu. Its arrows take that colour too, and a few colours make a big template far easier to read.

Every state in a template needs its own name, because transitions find their destination by name. NodeWorthy numbers a repeat for you and refuses a rename to a name already taken.

Copy, paste and duplicate. Right-click a state for **Copy** or **Duplicate**, or select several and press Ctrl+C, Ctrl+V or Ctrl+D. Everything in them comes along (actions, transitions, and the states inside them) as separate copies, and changing one never changes the other. Pasted states get a number if their names are taken, and the transitions between them point at each other's new names. Copied states can be pasted into another template too.

Add Breakpoint, also on a state's right-click menu, is described in 7.1.

6.3 Actions

Click a state and its actions appear in the **Actions** tab on the right, in the order they run, a card each. Along the top of a card are a tick that switches it on and off without losing its settings, and its name. Click the name to open the action's settings. The one that was open folds away, clicking the open one again folds it, and double-clicking the name gives the action a name of your own. After the name come  on an action that can run once or every frame, lit while it runs every frame (press it to change), , which opens its page in the Action Reference, and , a menu to move it up or down, duplicate it, add another action above it, change it into a related action (Convert To), reset its settings, open its page, or remove it. Under the name, a line says what it does, such as *Wait 1 s*, with the variables it reads in colour. A card with a problem says so on a line of its own, in the same words as the problems button. While the game runs, the card of the machine picked beside Debug shows a live line. It says whether it is running, done, or waiting for the action above, and what it is doing now. Drag a card by its name to put it somewhere else in the list. Where a setting reads a variable, the line names it in square brackets, such as **Turn by [Spin] over 0.5 s**. A tick box that reads a variable changes what the line says rather than appearing in it, so the line stays as it is and the variable is named after it, as in **Move to (0, 0, 0) over 1 s (Relative: [back])**. A choice from a list has no Var, because it is picked. An action you have just added opens. The boxes in the graph name each state and its ways out and nothing more, to keep a big template readable. Its actions are always here. Every tab works this way. Ways out, events and variables all open by pressing their name and fold when another opens. The state's name is the heading at the top, and you can type into it to rename the state. Its End State tick sits beside it. Its ways out are on the **Transitions** tab. An action added to a machine on a 2D node starts with its distances and speeds in pixels, not metres. That way a character walks at a speed that shows, a raycast looks down the screen and a following camera sits on what it follows.

Actions
Transitions
Variables
Events
Globals
Theme

Chasing

Actions

Add Action...

☒
1. Chase

?

⋮

what it senses, stopping 1.5 short

Body

Self

Var

Target Variable

None

▼

Speed

↺

4.5

Var

Stop Within

1.5

Var

Reached Event

↺

InReach

▼

Finish on Arrival

☐ On

Var

Turn to Face

☒ On

Var

Turn Speed

10.0

Var

Falls

☒ On

Var

Update Interval

0.0

Wait for Previous

☐ On

Disabled

☐ On

Display Name

☒
> 2. Check: Target Lost

↺

?

⋮

Lost sight for 0.6 s → LostSight

Parallel states: none

Add Parallel State

Chasing's actions. Chase is open, and the Check below it, asking whether sight of the target has been lost, is folded until you click it.

Add Action... opens the **browser**. It holds every action there is, grouped by what it is for, with a search box. Type what you want to do, "move", "wait", "sound", "key", and the list narrows. It searches the descriptions too, so "colour" finds Set Property. Picking one shows what it does and every setting it has, in plain English, taken from the action itself. Double-click, or press Add, to put it on the state. Right-click a state in the graph and choose **Add Action...** for the same window.

Questions such as Distance or Timeout are in it too, each **in the group of what it asks about and marked Check**.

Three settings appear on every action.

- **Run Mode:** *Once* does the work the moment the state is entered and no more. *Every Frame* keeps doing it for as long as the state is active. Setting a property, doing a sum or sending an event is usually *Once*. Reading the keys being held, keeping a value in range or feeding a blend is *Every Frame*. Actions that take time of their own, such as *Wait*, *Move To* or *Play Animation*, do not show this setting, because they always run until they are done. **Watch out.** An action on *Every Frame* is never finished, so a state holding one never sends *FINISHED*, and a transition on *FINISHED* from that state never fires. That includes a *Check* left on *Every Frame*. Leave such a state on another event, such as the one its *Check* sends. On each frame the *Every Frame* actions of a state take their turn from the top of the list down, so a *Set Variable* above a *Rotate To* has added to the angle before the turn reads it. Put the action that works a value out above the one that uses it.
- **Update Interval:** seconds between repeats. On an *Every Frame* action it is how often the work is done again. On an action that takes time, such as *Move To* or *Rotate To*, it is how often the action moves on, so a large value makes the movement jump in steps. Zero repeats every frame. Raise it only for expensive work. For a distance check, 0.25 is plenty.
- **Wait For Previous:** hold off until the action above has finished. This is how you build a sequence, for example move, then play a sound, then wait, then send an event.

Most actions have a setting that says which node they work on. It is called **Node**, or a name for what it points at, such as **Body** or **Camera**. Left empty, it works on **Self**, the node the Machine works on, and the field says which one, such as **Self (Player)** for a Machine under Player. Hover over it to see where that node sits in the scene. Press it to pick another node. It is remembered as the way there from the Machine, so the same template works on every node that has the same nodes under it. A few settings mean something else when empty, such as **Current camera**, and the field says that instead.

Only what can work can be picked. Pressing a node setting opens **Select a Node**, which shows your scene as the Scene dock does, with **Filter Nodes** at the top to narrow it by name. A node the setting cannot work on is greyed and cannot be picked. Rest the pointer on it and it says what the setting needs and what that node is, such as *Body needs a CharacterBody2D or a CharacterBody3D. Sprite is a Sprite2D*. A setting that wants an Inventory or a Machine also takes a node with one under it, such as your player. A setting that can work on any node greys nothing. If a node already picked cannot work, or is no longer in the scene, the field turns the warning colour and its hover says why. The settings that say who an action is about, such as **Look For**, **Hurts**, **Cover** or Send Event's **To**, work the same way. Something the action cannot work with is greyed, and its hover says what to do about it. Something with no Hurt Box says to give it Health, a node with no Machine says to add one, and a marker in 3D offered to a 2D character says the two are never in the same world.

Numbers stay where they work. A number setting takes only the numbers that make sense for it. A speed, a distance, a number of seconds or a count cannot go below zero, and a share such as an alpha stays between 0 and 1. Type a number outside that and the field keeps it at the nearest one that works. Settings where any number means something, such as an angle, a value in a sum or a seed, take any number. A number read from a variable is whatever the variable holds while the game runs.

The Machine's own **Target**, in the Inspector, is the node it works on, and empty it reads **Parent**, the node it sits under. A node the template cannot work on is refused, in a window saying what that node lacks, such as an AnimationPlayer to play from or a Perception node to see with, and the Target stays as it was. A template put on a Machine in the Inspector that cannot work on its node is taken off again the same way, and one that only needs a Perception node or a way to find its path is given it. One node has one Machine. A second one added under it is taken away again, because both would run and fight over the node. For two jobs at once, add **Parallel States** to the template, or put the second job on a child node with a Machine of its own.

Wherever a setting names something that belongs to another part of your game (a property, a method on your own script, an animation clip, a state or a blend parameter in an AnimationTree, an input action), there is a ▼ beside the box listing what is really there, grouped by where it comes from, with a line under each saying what it is in plain words. Nothing has to be spelled from memory, which was the commonest way to make an action quietly do nothing. You can still type into the box, for a name the game only makes while it runs. A setting holding a file is picked the same way. A sound, a picture, an item, a quest or a dialogue shows its name, and pressing it offers **Pick a sound...** (the project's own), **From my computer...** for a sound or a picture (copied into the project's `sounds` or `pictures` folder and picked, with nothing to drag in first), **New item...**, **New quest...** or **New dialogue...** (a new file in the project's `items`, `quests` or `conversations` folder, picked and opened to set up), and **None**.

A few settings take any kind of value, such as Set Property's **Value**, Set Variable's **Value** and Tween Property's **To**. They can hold text, a number, true or false, a colour or a Vector. The field shows the kind it holds now, and the button beside it changes the kind, so the same Value can set a label's text in one action and a light's colour in the next. Where the value goes with a property, picking that property from its list sets the kind for you. Pick **Visible** and Value becomes a tick box, pick **Scale** and To becomes a size.

An action reports itself finished when its work is done. A state whose actions have all finished sends itself the event FINISHED, and a transition on FINISHED fires.

What an action needs comes with it. Some actions need a node beside them. Adding the action adds it too, when it is missing. Look For Target, Chase, Search Last Known, Flee, Keep Distance, Strafe Around and the Can See and Target Lost checks get a **Perception** node. Navigate To gets a **NavigationAgent**, Start Dialogue a **Dialogue Box** on the screen and Start Quest a **Quest List**. Add Item and Use Item on the player get an **Inventory** under the player. Starting a template from

one of NodeWorthy's own behaviours does the same, and on a 2D character its speeds and distances are turned from metres into pixels.

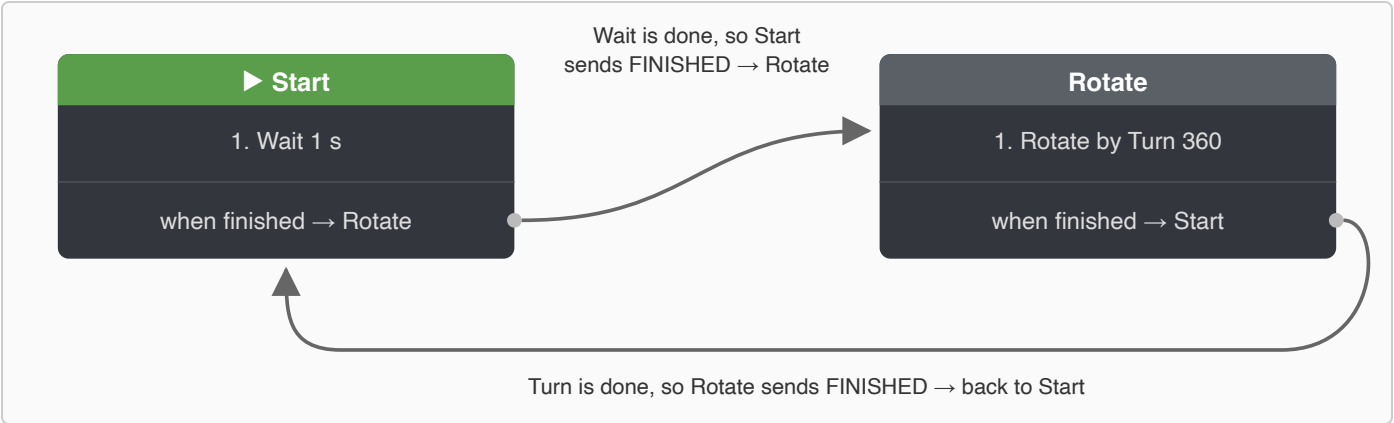
6.4 Events and transitions

The difference in one sentence. An **event** is something that happens, and a **transition** is what a state does when it happens.

The two always work as a pair, and they are easy to mix up because a transition's row shows the name of the event it waits for. They are different things, as the table shows.

	Event	Transition
What it is	A message with a name, such as <code>FINISHED</code> , <code>ButtonPressed</code> or <code>Opened</code> .	A rule on one state that says "while I am the current state, if this event arrives, go to that state".
Where you see it	On a transition's When button, in the settings of actions that send one, and in the Events tab.	As a row at the bottom of a state's box, such as "when finished → Rotate" for a transition on <code>FINISHED</code> , and in the Transitions tab.
Where it comes from	The state itself (<code>FINISHED</code>), a Check , Send Event , Event On Signal , or another machine.	You make it, with Add Transition on a state.
What it does by itself	Nothing. It is only an announcement. If no transition is waiting for it, nothing happens.	Nothing, until its event arrives. Then it moves the machine to another state.

Think of a doorbell. The bell ringing is the event. "If the bell rings while I am in the kitchen, go to the front door" is the transition. The bell does not move you, but your rule does. If you are in the garden and have no rule about the bell, it rings and you stay where you are.



Each state sends `FINISHED` when all its actions are done. Only that state's own transition, the row at the bottom of its box, hears it and moves the machine on.

Step by step, a box that keeps spinning. It has two states, `Start` and `Rotate`.

1. **Start** runs its one action, **Wait 1 s**. When the wait is over, every action in Start is done, so Start sends itself **FINISHED**.
2. Start has a transition on **FINISHED**, whose row reads **when finished** → **Rotate**. It was waiting for that name, and the machine leaves Start and enters Rotate.
3. **Rotate** runs a **Rotate To** action with **Turn** set to 360 and **Relative** ticked, so the box turns once all the way round. When the turn is done, Rotate sends itself **FINISHED**.
4. Rotate's transition on **FINISHED**, **when finished** → **Start**, takes the machine back to Start, and it all repeats.

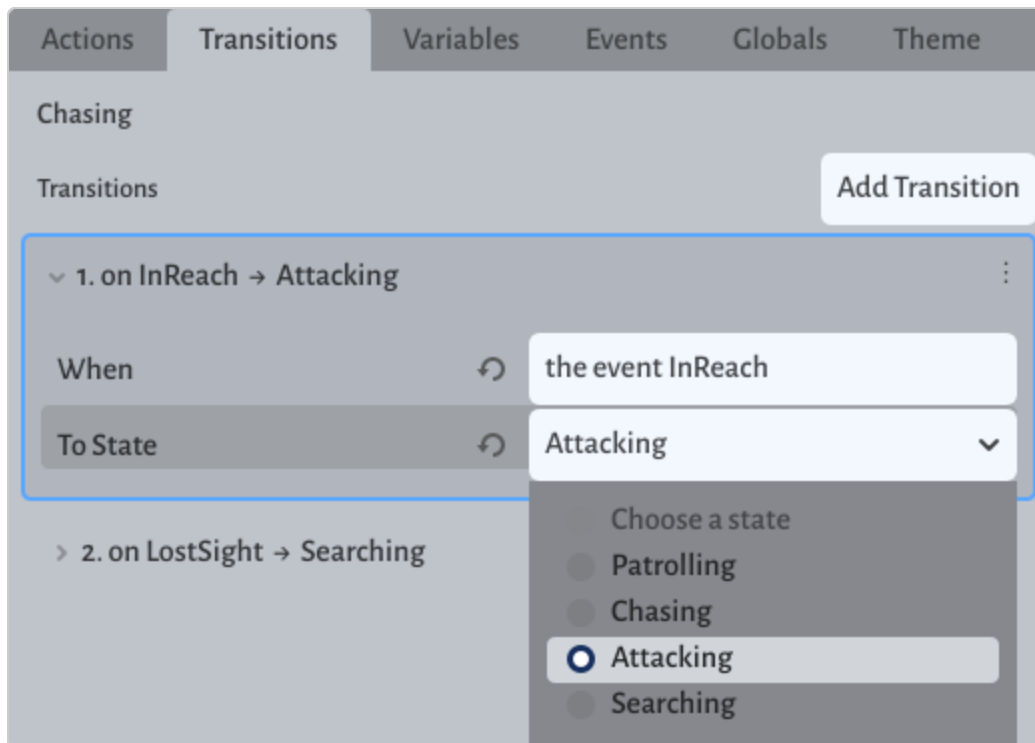
Both states send the same event, **FINISHED**, but the machine only listens with the transitions of the state it is in. So **FINISHED** from Start leads to Rotate, and **FINISHED** from Rotate leads back to Start.

Rules worth remembering

- **Only the current state is listening.** An event matters only if the state the machine is in, or a state that one sits inside (see 6.8), has a transition waiting for that name. A global event reaches every machine (see 6.7), but each machine still reacts only through its own current state's transitions.
- **An event nobody is waiting for is thrown away**, not saved for later. If **ButtonPressed** arrives while the machine is in a state with no transition for it, it is gone. Moving a moment later to a state that does wait for it will not bring it back. **FINISHED** is sent as soon as a state's work is done, and a transition on **FINISHED** from that state is taken right then.
- **Names must match exactly.** **Opened** and **opened** are two different events. Pick names from a list instead of typing them. A transition's **When** list, and the ▼ beside every event field on an action, offer every event this template already knows, so a name cannot be misspelt. You can still type anything. An event is only a name, and **Name an event of your own...** at the bottom of the **When** list takes a new one. Picking from the list, though, removes the commonest reason a machine "does nothing".
- **One way out at a time.** If two transitions on the same state could both fire, the higher row wins. Rows are checked from the top down, and the first one that fires wins, so put the important ways out first. Click the state and, in the **Transitions** tab, use **Move Up** and **Move Down** on a transition's : menu to change that order. A state inside another gets an event first, and the state it sits inside takes the event only when nothing inside it did.

Making a transition. A transition is a row at the bottom of its state's box, reading what it waits for and where it goes, such as "on **Opened** → **Open**", or "when finished → **Rotate**" for **FINISHED**, or "when the player touches it → **Opening**" when its own state has the action that makes it happen. It has a port on the right that you drag onto the state it should lead to, or you pick the state from **To State** in the **Transitions** tab.

- A transition is taken the moment its event is sent, before anything else happens. When an action in a state sends it, the actions listed below that one in the state never run. Use **FINISHED** for "when this state is done, move on".
- Anything can send the event, whether an action such as Event On Signal or Send Event, another machine, or a **Check**.



Chasing's ways out on the Transitions tab. To State lists every state in the template.

Events Godot sends by itself. Near the bottom of a transition's **When** list, **Signals** holds the things that happen to the node the machine works on, with no action needed in the state. They carry the names Godot itself gives those moments, the same names as the signals in Godot's Node dock. **body_entered** and **body_exited** come when a body walks into or out of an area, or when a rigid body bumps into something or stops touching it. **area_entered** and **area_exited** come when another area comes in or leaves, and **mouse_entered** and **mouse_exited** when the mouse moves onto or off a node. **button_down** comes when the mouse button is pressed on it and **pressed** when it is clicked, whether it is a button, a control or a physics body. **right_pressed** and **middle_pressed** come when it is clicked with the right or the middle button, and **text_submitted** when Enter is pressed in a text box. **hidden** and **shown** come when the node is hidden or shown, and **application_focus_out** when the player switches away from the game. Hover over one in the list to see what it is and what it needs. For example, **body_entered** needs the machine to sit under an Area2D or Area3D or a rigid body, and a click needs a physics body, an area or a control. A signal the machine's node can never send is greyed out in the list and says what would send it, and the problems button names one a transition already waits for. A machine listens only for the ones its transitions wait for. Because these names are taken, give your own events names that say what they mean, such as `SwitchPressed`, not `pressed`. To pick out only the player, or to listen to an area somewhere else, use **Event On Collision** in the state instead.

A transition may point back at its own state, and on an event that starts the state again. It is left properly and entered afresh, so its actions run from the top. That is how a doorbell pressed again while it is still ringing starts its ring over. A Check (see 6.6) sends its event when its answer changes, unless you untick **Only When It Changes**, so a question that stays true does not restart the state every frame.

Who sends it? A transition's **When** list also offers, under **Sent by others in this scene**, the events other machines and nodes send to this one, each saying who sends it, such as *When Enemy sends CaughtYou* or *When Hurt Box sends HURT*. Pick one rather than typing it. When the event is sent from elsewhere, the line under When says so, *Sent by Enemy*. When nothing in the scene sends it, it says *Nothing in this scene sends Opened yet. Press When and pick what makes it happen*. A machine in another scene may send it, and then all is well. If not, press **When** and pick how it should happen. The top of the list offers **When the player touches it**, **When the player comes near**, **When E is pressed beside it**, **When it sees the player** or **After a few seconds**. The first four happen round a place in the world, so on a machine whose node has none, such as a Label or a plain Node, they are greyed out and only the last can be picked. The action that makes it happen is added to the state the arrow leaves, already set up, and the arrow waits for the event it sends. That action is Event On Collision, Event On Input (E, while the player stands near), Look For Target (with a Perception node given to the character if it has none) or Send Event with a two-second Delay. **When everything in this state has finished**, just under them, waits for FINISHED. Only an area notices what touches it, so a body, a picture or anything else is given a **Trigger** area a little bigger than itself, or a wider **Near** area for "comes near", and the action listens to that.

6.5 Variables

A template carries named values, such as a score, a count, a flag or the node a raycast found. Actions read and write them, checks test them, and every machine using the template gets its own copy, starting at the value you set. A variable with **Is Global** ticked is the exception. It is one value for the whole game, as described below.

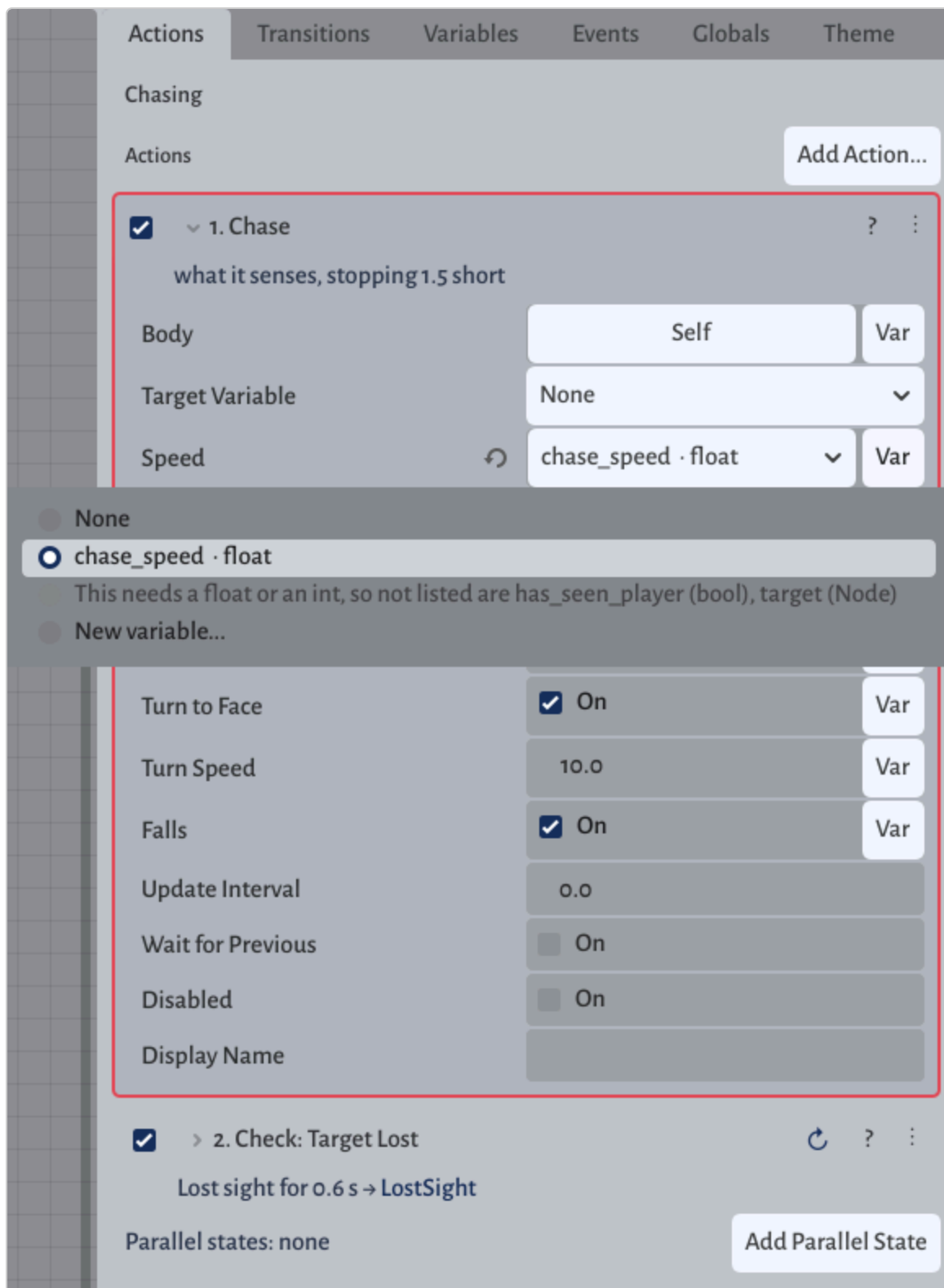
The **Variables** tab sits beside the Actions and Transitions tabs on the right of the graph.

The screenshot shows the 'Variables' tab of a game engine. At the top, there are tabs for 'Actions', 'Transitions', 'Variables', 'Events', 'Globals', and 'Theme'. Below these, there's a 'Variables' section with a dropdown menu set to 'float' and an 'Add Variable' button. A list of variables is shown: 'chase_speed', 'has_seen_player', and 'target'. The 'target' variable is expanded, showing its details in a form. The form has the following fields: 'Name' (target), 'Type' (Node), 'Starting Value' (Nothing yet), 'Description' (A Node this machine remembers.), 'Is Global' (On), and 'Exported' (checkbox). The 'target' variable is highlighted with a red border.

The Variables tab, with a number, a true-or-false and a node variable.

- Pick what sort of value you want and press **Add Variable**. A **float** is a number that can have a fraction, such as a speed or an angle, and an **int** is a **whole number**, such as a count of lives. There is also a **bool**, **true or false**, and a **String**, **text**. A **Vector2** is two numbers, X and Y, for a place, a direction, a speed or a size in a 2D game, and a **Vector3** is the same with three numbers, X, Y and Z, for a 3D game. Last come a **Color**, a **Node**, an **Array**, a **Dictionary** and a **Resource**. The sort decides what it starts as, and which settings offer it. Every variable names its sort on its **Type** row, straight under its name. Pick another there and the variable starts again as one of those, and Undo gives back what it held. An angle is a float.
- Give it a name and a starting value. The **Description** becomes its hover text, so write one if anybody else will read this template. Renaming a variable follows it through. Every action and every check that named it, and every setting with **Var** pressed on it, names the new one, in one step you can undo.

- An action uses it by picking it. Every setting that names a variable, such as **Store In** or **Target Variable**, is a list of the template's variables. It lists first the ones holding the right sort of value, so a speed lists the floats, a Check's answer the true-or-false ones and a target the Nodes. Under **Converts** come the ones that can be used by converting them, each saying how, such as an int read as a float, a Vector2 in a 3D setting with its Z as 0, or a Node standing for the place it stands. The last line names the variables left out and the kind the setting takes, to show why yours is not there. **New variable...** at the bottom of the list makes one of the right sort and picks it. A name that is not a variable, or holds the wrong sort of value, is shown as such in the list, with what it holds and what the setting takes, such as *flag (holds true or false, but this needs a float)*. The Problems button lists it, and while the game runs the action says so in the Output panel rather than quietly using the value typed in. Most settings that hold a value offer both, in one field. Type a number in, or press **Var** beside it and pick a variable, which is read while the game runs. Press **Var** again to go back to the typed value. A list of choices, such as Set Variable's Operation, and a row of tick boxes are picked in the panel and have no Var. To do one thing or another as the game goes, use a Check or a second state. A place or a direction is two number boxes in a 2D game and three in a 3D one, with its Var beside them. The node an action works on is one of those settings. Press **Var** beside **Node**, **Body** or **Camera** and pick a Node variable, and the action works on whatever node that variable holds, such as the enemy a Raycast hit, the coin that was touched or the label that was just spawned. While the variable holds no node, the action does nothing. It never falls back to Self.
- The **problems** button in the toolbar tells you when an action reads a name the template does not have. A missing name is the usual reason a check never says yes. The settings only offer the variables the template has, and **New variable...** makes one it does not.
- **While the game runs**, each row shows what the watched machine's variable holds, in green. Pick the machine under Debug in the toolbar first (see 7.1). This is the fastest way to see why something is not happening.



Chase's Speed with **Var** pressed. It reads chase_speed while the game runs, and the list offers only the number variables.

Nodes. A Node variable holds something in the scene, such as the player to chase, the door to open or the enemy a Raycast hit. Its **Starting Value** says *Nothing yet* until you press it and pick a node from the scene, and **X** beside it clears it again. The node is kept as the way to it from the Machine, so every machine running the template looks the same way from where it sits. Ten coins whose template picks the Sparkle node under each coin each find their own Sparkle. Leave it at

Nothing yet for the game to fill, as a Raycast or a collision does. If the node is not there when the machine starts, the variable holds nothing and the Output panel says so. Picking the starting node greys the nodes that the actions reading the variable cannot work on, so a variable that Jump reads as its Body offers only character bodies. If nothing reads it yet, any node can be picked. The problems button checks the starting node against every setting that reads the variable, and names one that is no longer in the scene.

Warning signs on NodeWorthy's nodes. A Hurt Box, Hit Box, Perception or Minimap that cannot work where it is shows Godot's warning sign beside it in the Scene dock. Rest the pointer on the sign to read why and what to do. It may sit under a plain Node instead of a 2D or 3D character, its character may have no collision shape or no health, nobody in the level may be in the group it watches for, or a Minimap may sit in the 2D world instead of on the screen. Fix it and the sign goes.

Lists. A list is how a machine remembers several things, for example the doors it has opened, the spawn points to choose between or the lines of a dialogue. Make one the same way as any other variable (pick **Array** and press Add Variable), and its starting items are written out one to a line, with **Add item** to put another on the end and **X** to take one out. Its Starting Value line says how many it starts with. **Element Type** says what goes in it. **Values** are typed, and each is read as what it plainly is. `12` is a number, `true` is a tick, `1, 2, 3` is a place, and anything else is words. **Nodes** are picked from the scene, each row a button to press, such as the markers a guard walks between. Like a Node variable, each is found from the Machine as it starts. **Files** are picked from the FileSystem, such as the items a chest can hold.

Most lists start empty and fill up while the game runs. The **Array** action does that work. It can add an item, put one in at a place, change or take out the one at a place, read the one at a given place, find where an item is, count them, pick one at random, ask whether it holds something, sort them smallest or largest first, or empty it. It also makes the variable for you if you name one that does not exist yet. **For Each** then walks through a list one item at a time, sending **Next** while items remain and **Done** when they run out. Use it to do something to every enemy in a room or every point on a patrol. An event can carry a whole list too. Set its **Argument** to Array in the Events tab, and the transition that waits for it can drop what it carries straight into a list variable.

Tables. A table keeps a value for each of several names. Examples are the player's inventory as *coins* 12 and *keys* 2, the best time on each level and the price of each thing in a shop. Pick **Dictionary** and press Add Variable, and its starting entries are written out one to a row, a name beside its value, with **Add entry** to put another in and **X** to take one out, and its Starting Value line says how many it starts with. Values are read the same way as a list's typed items. The **Dictionary** action sets an entry, reads one into a variable, asks whether one is there, takes one out, counts them, or makes a list of their names for **For Each**, and its **Missing Event** is sent when the entry asked for is not there. The entry's name can come from a variable, so one action can add to whichever item the player just picked up.

Resources. A Resource variable holds something from the FileSystem dock, like an item, a scene, a picture or a sound. Pick **Resource** and press Add Variable, and its starting value is a picker. Drag a file onto it, or leave it empty for the game to fill, as **Pick Item** does when it chooses an item for a pedestal. Settings that want a file, such as Pick Item's **Store In** or Give Item's **Item** with **Var** pressed, list only Resource variables. An event can carry a Resource too. Set its **Argument** to Resource, and the transition that waits for it can put it into a Resource variable. Written into text with `{name}` , a file shows its own name, so an item shows as *Spire Heart*.

Variables an action brings. An action that keeps an answer, such as Get Input Vector keeping the direction held in `move` , or Stamina keeping its meter in `stamina` (starting full), brings that variable with it when you add it, so the next action can read it straight away. Point the answer at another variable instead and the one it brought is taken away again, as long as nothing else uses it and you have not written a description for it.

One template, different starting values. Every machine using a template starts its variables where the template says, unless the Machine is given its own. Select the Machine in the Scene dock. Its template opens, and the top of the **Variables** tab says that this machine runs it. Open a variable and, under its own settings, a box called, say, **Door starts at** holds this machine's starting value. Change it and only that machine starts there. Two doors can share one template and each lead to a different level. **Same As The Template** puts it back. A list of values or a table can be given its own starting value there too, written out a row at a time. In a racing game, for example, each stage carries its own rivals' times on one template. A Node variable is picked from the scene the same way, so each switch in a level picks the door it opens. A variable holding a file (a Resource) is picked from the project's files, and each villager can have a different dialogue. A text variable that a **Change Scene** reads as the scene to go to is picked from the project's scene files, and each level's exit leads to its own next level. A Machine inside a scene placed in the level, such as a switch dropped in twice, can only keep values of its own when that placed scene has **Editable Children** on, and giving it one turns that on for you. When several machines in the scene run the template and none is picked, the tab says so. The same values are under **Starting Values** at the bottom of the Inspector, for anybody who works there. Renaming the variable in the Variables tab brings the value along. If the template loses the variable, the problems button says so, and a **Forget** button under Starting Values in the Inspector clears the value that is no longer used.

Limits. A float or int variable has **Min** and **Max**. Pick **A number** and type it, such as `0` , or pick another number variable of the template from the same list, such as `max_health` , and the variable can never go past it, whatever changes it, whether that is Set Variable, a potion, gear or a hit. Gold never goes below 0, and health never goes above `max_health`. When the variable a limit names goes down, the one it limits comes down with it, so taking off an amulet that raised `max_health` takes extra health off too. **No limit** takes a limit off. If a variable a limit names is renamed or removed, the list shows it as not in this template, and the problems button says so too.

Attributes such as health, mana and stamina. A number kept up to another number, as health is up to max_health, is an *attribute*. Once its Max is set, **Starts** appears, with three choices. **At Its Starting Value** uses the Starting Value, kept within the limits. **Full** starts it as high as its Max lets it, so health starts at max_health, whatever max_health is. **At A Share** starts it part of the way up, with **Share** saying how far (50 starts half full), and the Starting Value is then put away, since it is not used. max_health is then a separate stat that levelling up, gear and potions change, and health always fits inside it. Give It → Health and the ready-made Enemy make health an attribute that starts Full. An attribute carried over from the last scene, or handed in by Run Template, goes on where it was.

Stats. A character's stats are ordinary number variables, such as strength, defence, speed and health. **Modify Stat** adds to one or multiplies it under a name, and taking that name away later takes exactly that change back off, whatever else changed meanwhile. Items and gear do the same through their stat changes. A stat name the character does not have, usually a typo, is reported in the Output panel. To show the stats, add a **Stat Panel** (see 10.35).

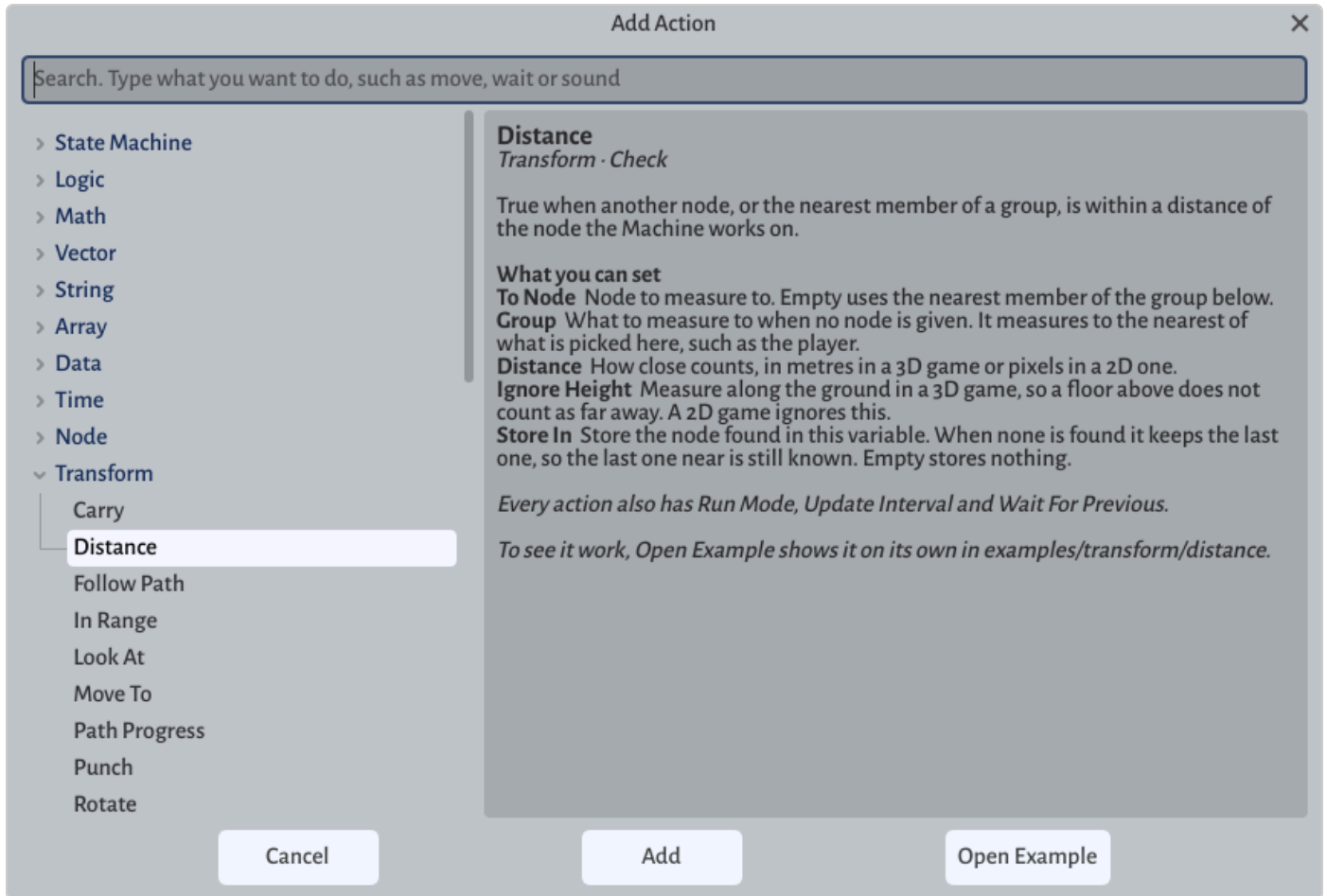
Global variables. Tick **Is Global** on a variable for a value the whole game shares. That might be the score, the lives, the coins or what the player carries. Every machine with a global variable of that name sees the same value, so the player's machine adds to the score and the HUD's machine shows it, and a check waiting for the score to change hears about it straight away. The value carries on when the scene changes, so the next level starts with the score the last one ended on. It starts at its starting value the first time any machine uses it. You do not have to make it again by hand in each template. Every variable list offers the game's globals under **Globals in your game**, such as a `score` another template keeps, and picking one adds it to this template as well, global, and both share the one value. **New variable...** has a tick, **Shared by the whole game (global)**, for making a new one. Its row in the Variables tab says (*global*).

{name} in words. Set Text, Format String, Text To Speech and Print put a variable's value wherever the words say `{name}`, as in `Score: {score}`. The template needs a variable of that name, or the words show `{score}` itself. The problems button says so, naming the setting.

6.6 Asking a question with Checks

Checks ask a question. "Is the ball within 4 metres?", "has this state run for two seconds?", "is the fuel below 30?" are each asked by a Check action in the state. **Add Action...** lists every question in **the group of what it asks about, marked Check**. The question's settings sit at the top of the action, and below them you choose what each answer sends, in **True Event**, **False Event**, or both. A check set to Every Frame keeps asking while its state runs. A question about variables alone, such as "is the fuel below 30?", is asked again the moment one of its variables is written rather than every frame, as long as Only When It Changes is ticked, so a machine waiting on one costs nothing until something changes. Unticked, it is asked, and sends, every frame. Give a question about the world an **Update Interval** to ask it less often. In the Actions tab its line reads

as the question and the event, "Ball within 4 m → BALL NEAR", and on the state's box the transition waiting for BALL NEAR shows where it leads.



Add Action... with the Transform group open, where Distance sits. Every question is here, with what it asks and what you can set on it.

Actions
Transitions
Variables
Events
Globals
Theme

Chasing

Actions

Add Action...

☒ > 1. Chase

?

:

what it senses, stopping 1.5 short

☒ 2. Check: Target Lost

↺

?

:

Lost sight for 0.6 s → LostSight

Body

Self

Var

Out of Sight For

↺

0.6

Var

Only When Forgotten

☐

On

Var

Ask The Opposite

☐

On

Wait for Actions

☐

On

Store In

None

▼

True Event

↺

LostSight

▼

False Event

event name

▼

Only When It Changes

☒

On

Var

Run Mode

↺

Every Frame

▼

Update Interval

0.0

Wait for Previous

☐

On

Disabled

☐

On

Display Name

Parallel states: none

Add Parallel State

A Check in the Guard template, with the question's settings at the top and the event each answer sends below them.

The **Check** action itself is described in full in 10.2, and every question it can ask is listed in chapter 10, in the group of what it asks about.

6.7 The Events tab, with events written down, what they carry, and global events

An event is only a name. Any action can send one, any transition can wait for one, and nothing has to be set up first. That is deliberate, so you can build a whole machine without ever opening this tab.

The **Events** tab, beside Actions, Transitions and Variables, is where you write the important ones down, and it buys you three things.

- **You can say what an event means**, in a line the next person reads as hover text. Renaming an event here follows it through as well. Every transition waiting for it and every action sending it says the new name, in one step you can undo.
- **You can give it a value to carry**, under **Argument**, with a name for the value. Its heading, and the event lists, then say what it carries, and the transitions waiting for it can put the value in a variable (see **What an event carries**, below).
- **You can make one global**. A global event is heard by *every* machine in the game, not only the one that sent it, with nothing wiring them together. It suits the handful of things everything cares about, such as the game pausing, the player dying or a level ending. One machine sends `LightsOut` and every lamp in the level hears it.
- **The problems button stops warning about it**. An action that sends a name nothing in this template waits for is nearly always a typo (`Open` against `Opened`), and the problems button says so. Writing the event down says "this one is meant for somebody else", and the warning goes. An action set to reach a group or every machine is left alone for the same reason, as is an event an action still sends under the name it came with, such as Change Scene's LoadFailed. That one is there for you to use if you want it.

What an event carries. An event can carry a value with it, for example the points a coin is worth, what came into an area or the text somebody typed. Write the event down in the Events tab and choose its **Argument**, and give the value a name. A transition waiting for that event then shows a setting named after it, such as **Store Points In**, listing only the variables that can hold it. The value is put there as the transition is taken, before the next state's actions start. System events that are about something carry it already. For example, **body_entered** carries the body that came in, as **Body. Send Event** says what to send in its **Data**, typed in or read from a variable.

The lower half of the tab lists every event name the template actually uses, and whether something sends it and something waits for it. It looks at every machine in the open scene, not only this one, so a button that sends `OPEN_DOOR` to a door reads *sent, waited for by Door*, and the door reads *waited for, sent by OpenButton*. **Green** means both ends are there. **FINISHED** and system events always count as sent, because NodeWorthy sends FINISHED when a state's actions are done, and Godot sends the system events. **Orange** means something is missing. Perhaps nothing anywhere waits for a name that is sent, nothing sends a name that is waited for, or an action sends a name only to its own machine while another machine is waiting for it. Hover the line to see what to do. **Grey** means nothing is wrong, but nothing in this scene is at the other

end. It may be a name you wrote down because a machine in another scene sends or hears it, such as an enemy a bullet hits, or an event an action sends by itself that you have not used, such as Chase's *Reached*. An active item's Use Event counts as sent by **Use Item**.

Under each name it says where, such as *sent in Standing: Check: Input or waited for in Standing* → *Crouching*. Click one to go straight to that state, even inside another state. Following an event from the action that sends it to the transition that waits for it takes two clicks. **Add Every Used Event** writes them all down in one go so you can describe them. While the game runs, the tab also lists the events the watched machine has just sent, global ones marked as such.

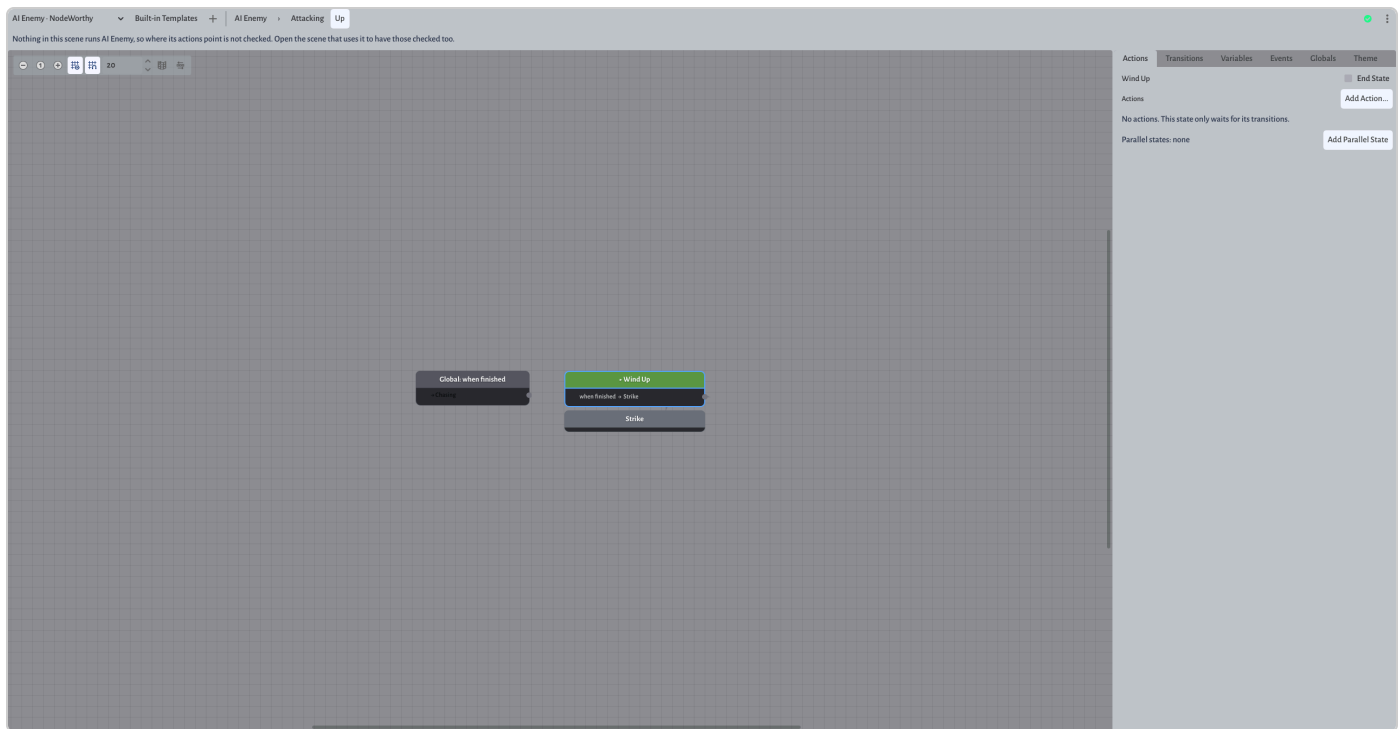
A transition waiting for an event that nothing sends is never reported by the problems button, because it can come from another scene. The tab shows it in orange so you can check.

Every global in one place, on the Globals tab. The Events tab shows one template. The **Globals** tab, beside the Theme tab on the right, shows the whole project. Its first half lists every global variable, with its name, what it starts as and every template that carries it. Its second half lists every global event, showing which templates send it and which wait for it, including templates that wait for it without having written it down themselves. Click a template's name to open it. It warns you in orange when nothing sends a global event, when nothing waits for one, when two names differ only by a space, an underscore or a capital letter (`DoorOpened` and `Door Opened` are two different events), and when two templates disagree about what a global variable starts as.

6.8 Sub states

A state can hold other states. While the parent runs, one child runs inside it. **The child hears an event first**, and the parent takes the event only when the child has no transition for it. That is the point. A rule shared by every child is written once on the parent, and a child that needs a different answer to the same event has its own transition for it. Section 3.5 walks through an example.

- Right-click a state and choose **Add Sub State**. The graph opens that state's level, with a breadcrumb above it. Double-click a box marked "2 sub states" to go in, and press **Up** or click a breadcrumb to come out.
- Inside a parent, **Set as Start** chooses the child that runs first.
- **Mark as End State** says "reaching this child, with its actions done, means the parent's work is finished". The parent can then leave with a transition on FINISHED, which also waits for the parent's own actions, or with a **Sub State Finished** check sending an event of its own.
- Inside a parent, the parent's own transitions float as small **global** boxes beside their targets, since they apply from every child that does not take the event itself.



Inside Attacking, which holds Wind Up and Strike. The breadcrumb in the toolbar shows where you are, and Attacking's own way out floats beside them.

6.9 Parallel states, for several things at once

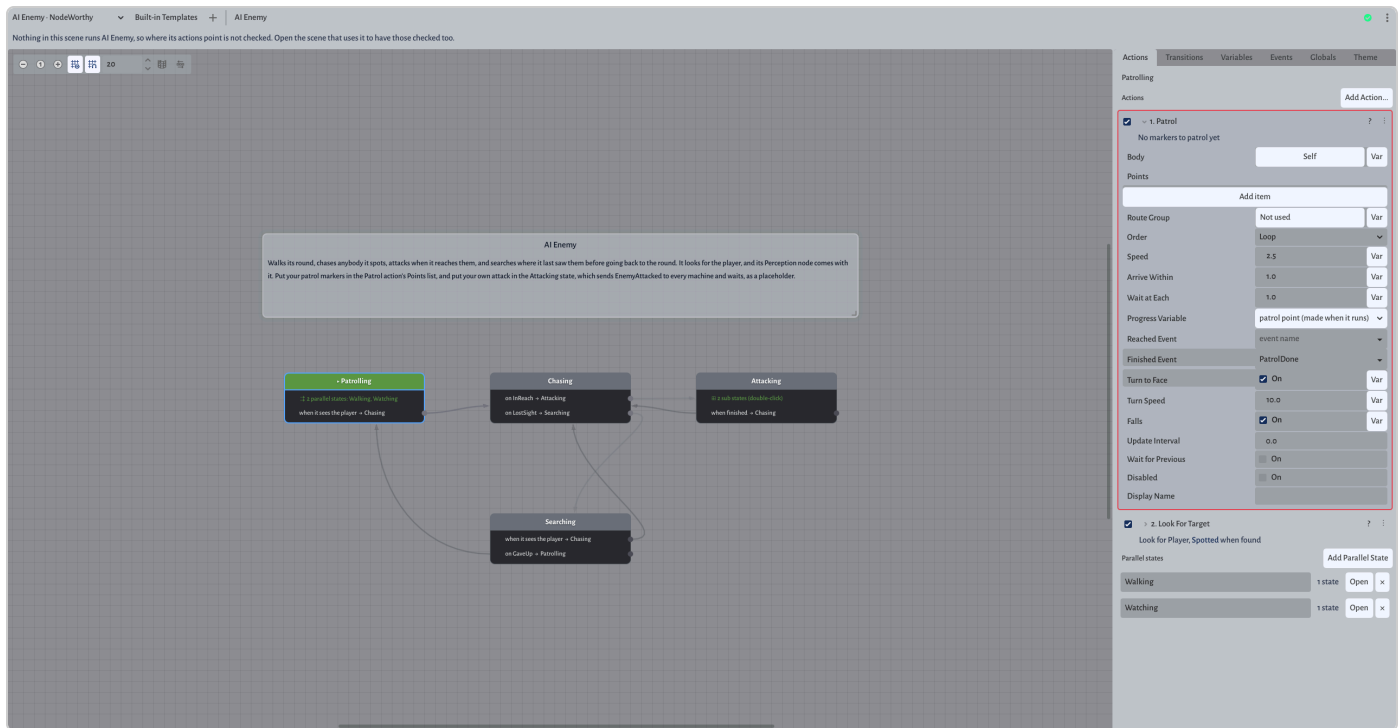
A sub state is one thing running inside another. **Parallel states** are several, at the same time. A ride whose lights flash through their patterns while its music plays through its tracks. A level that counts down while its doors work on their own. A menu that animates while it listens for a key.

Each parallel state is a little machine of its own, with its own states, its own start state and its own current state. They all run while the state holding them is active, and they all stop the moment it is left.

- Click a state, then **Add Parallel State** at the bottom of its Actions tab (or right-click the box, **Add Parallel State**). Name it for what it does, such as Lights, Music or Timer. Two parallel states of one state never share a name, so a name already taken gets a number after it.
- The graph opens the new parallel state, empty. Press **+**, **New State...** to build inside it, exactly as at any other level, and the breadcrumb takes you back out.
- A box that holds parallel states shows **2 parallel states: Lights, Music**. Double-click opens the first, and right-click, **Open Parallel State**, picks any of them.
- An event is offered to every parallel state, and each one waiting for it moves on. The state holding them takes the event only when none of them did.
- A transition inside a parallel state may lead anywhere, including out of the parent altogether. When it does, every parallel state stops, because the state holding them was left.
- To wait for *all* of them, mark the last state of each parallel state as an **End State** and put a transition on **FINISHED**, or a **Sub State Finished** check, on the parent. The parent counts

as finished only when every parallel state has ended.

- A state runs parallel states *or* sub states, not both, so a state that already has one is not offered the other. The problems button says so if a template has both.



The AI Enemy template's Patrolling state with two parallel states added, Walking and Watching. The line on its box names them.

The machine reports the first parallel state's current state as its own, and the graph lights that state up.

6.10 Notes and groups

New Note puts a box on the graph, in the middle of what you can see or just below whatever is there. On its own it is a comment, with a heading and a few lines saying why this corner of the machine works the way it does. Double-click its heading to type a new one straight onto the box. Enter or clicking away keeps it, and Esc leaves it as it was. Click the box and its heading, its words and its colour are in the Actions tab on the right, where a state's actions would be. Drag its bottom right corner to make it any size. A box too small for its words grows downwards until they all show.

A note stands on its own, as a state does, so nothing covers it and it covers nothing. When it grows to show its words, the states below it move down to make room, keeping their places beside each other. Drop a note onto a state, or another note, and it slides straight down until it is clear. **Auto Layout** moves any note the states land on. Each of these is one step to undo.

Drag states onto it and it becomes a **group**. It remembers which states it holds, and from then on they move together when you drag the box. Right-click it for **Empty This Group**, which lets the states go without deleting the note, or **Delete Note**. Tick **Fits its states** and the box shrinks to

whatever it holds instead of staying the size you dragged. Its words then sit in a strip just above the states it holds, so no box covers them.



A group called Hunting holding Chasing and Attacking, with Fits its states ticked, so its words sit just above the states.

A note belongs to the level it was made on, so a group drawn inside a parent state stays inside it and does not follow you back out. Notes change nothing about how the machine runs. They are there for whoever reads it next, which is usually you in three months.

6.11 Which step things run in

Godot runs your game in two steps. The **frame step** comes once per drawn frame, and the **physics step** at a fixed rate. Some work has to happen in the physics step, and some is better on frames. NodeWorthy sorts this out for you.

- **Actions that must be in the physics step say so themselves.** Apply Force, Set Velocity and Raycast run in the physics step whatever the machine is set to, because that is where pushing a body and asking the physics world a question actually work. Their first push or question happens as the state is entered. You do not choose, and you cannot get it wrong.
- **Everything else follows the machine's own setting.** That is the frame step, or the physics step if you tick **Physics Process**. Timers run on the machine's own step, and stand still whenever the machine cannot run. That is while the game is paused (unless the machine runs through a pause), and while its Process Mode, or that of a node above it, is Disabled. A machine that is switched off like that when the game begins does not start at all. It starts,

from its first state, the moment the node is switched on again, for example by a **Set Property** that sets that node's Process Mode back to Inherit. That is how enemies placed in a room wait, doing nothing at all, until the player walks in. Events, and the transitions they take, happen the moment they are sent, whichever step sent them. An event from a signal during the physics step, such as a body entering an area, arrives as soon as that step is over, so the state it leads to can change the physics world safely. A Check is asked where its question belongs, so a question for the physics world is asked in the physics step.

- **Turn on Physics Process** when most of what a machine does moves a body. Its states then change in step with the movement.
- **For "after everything else"**, the way a camera should follow only once the things it watches have moved, raise the Machine node's **Process Priority** in the Inspector. That is Godot's own ordering, and it works for machines just as for any node. A machine on Physics Process follows it in the physics step too, unless you set its **Physics Priority** yourself.

6.12 Making things move, and animation

A lift or a door needs no animation at all. Its movement can be the **Move To** action easing the node's position over a couple of seconds, and a traffic light can be a **Set Property** action changing a colour. For a door, a lift, a platform, a panel or a camera move, that is usually the whole answer. There is nothing to author, and the distance can come from a variable.

Smooth on any screen. Characters are moved 60 times a second, but many screens show 75, 120 or 144 pictures a second, where a character moved only 60 times would judder. So NodeWorthy draws what **Move Character**, **Drive**, **Camera Follow** and **Orbit Camera** move smoothly in between, and everything carried by it too, such as a character's picture and a camera under it. You set nothing for this. The one exception is **Mouse Look**, because a first-person view turns as soon as the mouse moves. What it turns is never smoothed, and aiming is never a frame behind.

There are four ways to move something, and it is worth knowing which you are reaching for.

- **Actions that move it directly** are Move To, Rotate To, Rotate, Look At and Tween Property. Best for mechanical motion on props.
- **A clip on an AnimationPlayer**, keyframed by hand, started with **Play Animation** and stopped, paused or resumed with Stop Animation. Best when the motion is hand-authored, such as a chest lid that bounces or a machine with pistons.
- **A sprite's own animations**, for a 2D game. These are named sets of frames on an AnimatedSprite2D, played with **Play Sprite Animation** and stopped, paused or held on one frame with Stop Sprite Animation. **Import Sprite Sheet...** (4.3) makes them from a sheet.
- **An AnimationTree**, for a character. The machine does not play clips here. It tells the tree what is happening with **Travel To** and **Set Tree Parameter** (a blend, a condition that lets the

tree move on, or a OneShot such as an attack played over the walk), and the tree does the blending. Two state machines each do their own job. Yours decides the behaviour, and the tree decides what that looks like. The **Playing** check asks the tree or player what it is doing, so a state can wait for an animation to end with no guessed number of seconds.

6.13 AI and what a character notices

Walking to a place is not AI. What makes an enemy feel alive is what it *knows*, and that lives in a **Perception** node (**SCPerception** in the Add Node dialog) you drop under the character, the same way a NavigationAgent goes under anything that walks. Adding Look For Target, Chase, Search Last Known, Flee, Keep Distance, Strafe Around, or a Can See or Target Lost check adds one for you, and so does picking what Look For Target looks for. Navigate To likewise brings the NavigationAgent it walks with. The AI actions and checks read it, so the range, the view cone and the memory are set once, on its settings in the Inspector (pick it in the Scene dock), not on every action.

It does three jobs.

- **Sight.** Is what it looks for within range, inside its view cone, and not behind a wall. Another character standing in the way does not hide anything, but a wall, a crate or any other body does. Facing is the character's own forward, so a guard really does have a blind spot behind it.
- **Memory.** Where it last saw what it was watching, and how long ago. This is the difference between an enemy that walks at you and one that *saw* you, lost you, went to where you were, and gave up.
- **A target.** One thing every state agrees it is dealing with, so the template reasons about *who*, not *where*.

Hearing is there too. Anything within Hearing Range is noticed whatever direction the character faces and whatever is in the way. Leave it at zero for sight alone.

The actions and checks that read a Perception node are in 10.24. Ready-made AI templates are in 4.3.

6.14 The Theme tab and how the game looks

Every menu, panel and button in your game can have one look, covering its font, its colours, how round its corners are, and your own pictures for buttons and panels. Set it once, in plain choices, in the **Theme** tab, the last tab on the right of the NodeWorthy panel, and everything follows it, from Godot's own buttons, labels, panels and bars to NodeWorthy's **Stat Panel** and **Inventory Grid**.

- **Start from a ready-made look.** Press **Clean**, **Parchment**, **Neon** or **Pixel**, and every choice below is filled in from it. Then change anything you like. The preview at the top of the tab is drawn in the look and changes as you do.
- **Plain choices.** A font (drag a .ttf or .otf file onto it), how big text is, the colours of text, panels, edges, buttons and the highlight a button takes under the pointer or a gamepad, how round corners are and how thick edges are.
- **Your own pictures.** Set **Button Style** to Pictures and give it your own picture of a button, and if you have them, one for under the pointer and one for pressed. A picture stretches to any size with its corners kept exactly as drawn. Under each picture is a small editor. Drag its four red lines to where the picture's corners end. Beside it the picture is drawn stretched wide, to show whether the corners keep their shape. **Panel Texture** works the same way for panels.
- **A pointer and pixel art.** **Pointer Texture** replaces the mouse pointer in your game. **Pixel Art** keeps pixel pictures crisp when they are drawn bigger.
- **Use For The Whole Game** makes it the look of the whole game. From then on, every change in the tab changes the game too. The choices are kept in `res://nodeworthy/game_look.tres` and the theme made from them beside it.

One screen, or one part of one, can look different. The look is an ordinary Godot **Theme**, and any Control can be given a Theme of its own in the Inspector. Everything under that Control takes it, and nothing else does. The recipe `ui/presentation/your_own_buttons` gives only its menu a look made of its own pictures, and Mossbrook gives only its screen the Pixel look. (Mossbrook is a small village adventure that comes with NodeWorthy, in `addons/nodeworthy/games/mossbrook/`, with a hero, a shop, slimes, two quests and a cave. Open `village.tscn` and run it. This chapter uses it as an example several times.)

6.15 Dialogues and the dialogue graph

What a character says, and what the player can answer, is drawn as a separate graph, the same way a machine is, with a box for each line and an arrow from each answer to the line it leads to.

- **Make one.** In the FileSystem dock, right-click a folder of your own and choose **Create New > NodeWorthy Dialogue...**, and name it. The window says it is saved in the folder you clicked, and **Folder...** picks another. It opens in the NodeWorthy panel. Open it again later by double-clicking the file. ◀ **Templates** goes back to the machine you were working on.
- **Lines.** **+ Line** adds a box, already joined on after the line picked, either as its next line or from its first answer that leads nowhere yet. Click a box to pick it, and at the side pick who says it in **Actor**, fill in the words, and, if you like, a **Portrait** to show beside them. The line with **The dialogue starts here** ticked is the first one said.
- **Actors.** An **Actor** is a file for someone who speaks. It holds their name, their **Portrait**, the colour of their name, how fast their words appear (**Letters Per Second**), and a **Talking**

Sound, a short blip played every few letters as their words appear, the way older games give a character a voice with no one reading it, with its **Sound Pitch** and how much it wanders. Make one with **New actor...** in a line's Actor list, or **Create New > NodeWorthy Actor...**, and pick it on every line they say. Then their name is never spelt two ways, and changing their picture changes it everywhere. A passer-by with one line needs no Actor. Leave Actor empty and type their name in **Speaker**.

- **Answers. Add Answer** at the side gives the picked line an answer for the player to choose, with its own green arrow on the box. Drag that arrow to the line the answer leads to, or drop it on empty space to make that line at once. A line with no answers has one arrow instead, showing where the dialogue goes when the player presses on. An arrow that leads nowhere ends the dialogue.
- **Tidy. Auto Layout** sets the lines out from left to right in the order they are said. The toolbar says how many problems the dialogue has, such as a line that nothing leads to, so it is never said. Point at it to read them. Everything is kept in the file as you go, and Undo works as it does in a machine.
- **Picked at random.** Set a line's **Answers Are to Picked At Random** and its answers are never shown. When the player presses on, one of them is picked, and the dialogue goes where its arrow leads. In the graph each such answer says *picked at random* under it, so it is not mistaken for a button. A guard greets the player a different way each time, one line picked from three.
- **Words that change.** A `{variable}` in the words or the answers is filled in when the line is said, as in Set Text, for example *You have {coins} coins*.
- **What an answer does.** Press the ⚙ beside an answer to open its settings. **Only When** offers the answer only when a check is true. Pick any check by name, such as Compare with gold at least 10, and its settings appear underneath, the same as in a Check action. An answer left out closes up, and a line whose answers are all left out moves on as if it had none. **Only Once** offers the answer only until it has been picked, so a question the player asks is gone once it is answered. **Send Event** sends an event when the answer is picked, and **Change Variable** sets a variable, or with **Add** adds to it (-10 takes 10 away), so *Pay 10 gold* is one answer. The box in the graph says what each answer does under its words.
- **Voices.** Pick a line, and under its words is the speaker's voice. Pick one of the computer's own voices, a pitch and a speed, and press **Hear It**. From then on every line that speaker says in the dialogue is read aloud as it appears. Voices differ from one computer to another. A voice the player's computer has not got is replaced by its own voice, at the same pitch and speed, so the speakers still sound different.

Every line said and every answer picked is remembered for the rest of the game, and Save Game keeps it. The **Dialogue Line Said** check asks about it, such as whether the guard has told the player the password yet. Put it in an answer's Only When, or in a Check action, and a villager can greet the player differently the second time.

Checks, events and variable changes belong to the machine having the dialogue, as its actions do, so they use its own variables and the game's global ones. The variable lists in an answer offer the variables of every template that plays the dialogue, and the global ones.

An answer that sends the machine elsewhere ends the dialogue. That is how a shop sells. In Mossbrook, *I'll take one* takes 10 gold and sends **BOUGHT**. The shopkeeper's machine goes to a **Selling** state that puts a potion in the hero's pack, and on to **After A Sale**, which has a short dialogue of its own, *Anything else?*

To have it said in your game, two things are needed. The first is a **Dialogue Box**. Right-click the level in the Scene dock and choose **NodeWorthy** → **Add To Screen** → **Dialogue Box**, and it goes along the bottom of the screen on the HUD, where the game's look reaches it. The second is the **Start Dialogue** action in the state the dialogue belongs to. Give it the dialogue, and the state waits until the dialogue ends, so a transition on **FINISHED** goes on afterwards. While it lasts the player is held still, as **Hold The Player Still** says, and a second dialogue begun while one is going takes the box over. The player reads a line a letter at a time. Enter, Space, a click or a gamepad's bottom button (A / Cross) shows the rest at once and then moves on, and the answers are buttons chosen with the arrows, the D-pad or a stick, which go round from the last answer to the first and never leave them. While a dialogue goes on the box has the focus, so a button or an Inventory Grid elsewhere on the screen cannot take those presses, and whatever had the focus has it back when the dialogue ends.

Talking to someone. The guard in Mossbrook is built this way. Its machine waits in **Alone** until the player walks near, then shows "E to talk". E or Enter sends **TALK**, and the **Talking** state has Start Dialogue in it, going back on FINISHED. The Enter that ends a dialogue is not taken as a new press and does not start the talk again. *I have a sword. Let me pass* is only offered once the hero has the sword, and sets the guard's `let_pass`. After the talk, a Check on `let_pass` sends him out of the road. One dialogue file can be used by any number of characters. Keep yours outside the addons folder, with your own templates.

6.16 Quests, for things to do, ticked off as they are done

A quest, a mission or a challenge is a file with a title, a line saying what it is about, the steps to do, and a reward. Its steps tick themselves off as the game is played, so nothing in your templates has to count.

- **Make one.** Choose **New Quest...** on the NodeWorthy panel's + menu, or right-click a folder of your own in the FileSystem dock and choose **Create New > NodeWorthy Quest...**, and name it. The window says where it is saved, and **Folder...** picks another folder. It opens in the Inspector, as an item does, with a first step to fill in. Its steps are rows there. **Add Step** adds one, and the **X** beside a step takes it away.

- **Steps.** Give each step the **Text** the Quest List shows, and pick in **Done When** how the game knows it is done. With **An Enemy Is Beaten**, the step waits for enemies to be beaten, as many **Times** as you say. Pick which **Enemy** from the kinds in your level (such as Slime), or leave it empty for any. With **The Player Reaches A Place**, pick the **Place** from what stands in the level, such as a Marker2D called Well or a door, and the step is done when the player comes **Within** reach of it (0 means close by). With **The Player Talks To Someone**, pick the dialogue they play in **Talks To**. The step is done when the player has heard it through. **The Player Carries An Item** wants at least **How Many** of an item, such as three mushrooms. **A Variable Reaches A Number** waits for a variable of the player's, or a global one, to reach **At Least**, such as 100 gold. **An Event Is Sent** is for anything else, and counts an event sent by any machine, as many Times as you say. Tick **In Order** on the quest and only the first step not done counts, so *Bring them to the elder* waits for *Find three mushrooms*. A step missing what it needs, such as a Place or a dialogue in Talks To, is named above the steps in the warning colour, saying what to pick, and the words go once it is picked.
- **Reward.** When every step is done, the quest is done and its reward given. The **Reward Items** go into the player's pack, **Reward Amount** is added to **Reward Variable** (20 to gold), and the **Reward Event** is sent to every machine in the scene, for a gate to open or the next quest to begin.
- **Beginning one.** The easiest giver is a ready-made **Villager**. Pick the quest in **Gives Quest** at the top of its Inspector, and it gives it once the player has talked to it (a Quest List goes on the HUD by itself). The **Start Quest** action begins a quest from any template, and so does an answer in a dialogue. Open its settings with the ⚙ and pick the quest in **Start Quest**, and *I'll help* begins it with nothing else in between. **Complete Quest** finishes one whatever its steps say, and **Fail Quest** fails one. A failed quest can be begun again.
- **Asking about one.** The **Quest Is** check asks whether a quest is not started, under way, done, failed or abandoned. In a dialogue, put it in an answer's **Only When**, and whoever gives the quest then offers it only once and says thank you once it is done. The **Quest Step Is** check asks about one step, picked by its words. It asks whether the step is done, not done, or the one to do now (with In Order, the first one not done), so the farmer can say "you found the mushrooms, now bring them to me".
- **When a step is done.** A step's **Done Event** is sent to every machine the moment it is ticked off, so the next door opens or someone says well done. A step's **Description** says more about it, under its words in a Quest List that shows descriptions.
- **Giving up, and failing.** **Abandon Quest** lets the player give a quest up, from a journal or by answering "never mind". It then shows as abandoned, its steps stop counting, and no reward is given. **Fail Quest** fails one, when the prisoner was not saved in time. Each sends the quest's own event, **Abandoned Event** or **Failed Event** (under **When It Ends Otherwise**), and either can be begun again with Start Quest.
- **How it looks.** A quest's **Icon** shows beside its title in the Quest List, and its **Color** marks it on the Minimap (and colours its title once changed from the usual gold). Tick **Hidden** for

something the player is not told about, such as an achievement or a secret. It is never listed or marked, but its steps count and its reward is given.

- **Tracking.** A quest is tracked from the moment it begins. **Track Quest** stops tracking one, or starts again. A Quest List with **Only Tracked** ticked shows only tracked quests, so a player with many quests sees the ones they follow, and only tracked quests are marked on the Minimap.
- **The quest log.** A **Quest Log** node (**NodeWorthy** → **Add To Screen** on the level) is the player's journal. Q opens it over the game, which stands still, with the quests down one side and the one picked set out beside them, step by step, with its reward. From it the player can track a quest, stop tracking it, or give it up.
- **On the map.** A **Minimap** (**NodeWorthy** → **Add To Screen** on the level) is a radar in any level, with the player in the middle. Besides the enemies, people and items near the player and the way out (10.35), it marks, in the quest's colour, what each tracked quest wants now. That means the place a step sends the player to, whoever plays the dialogue a step waits for, and each enemy a step wants beaten. One further off than the map reaches sits at its edge, pointing the way. A step done, or with In Order a step not reached yet, is not marked. Untick the Minimap's **Shows Quests** to leave the quests' marks off, and press its **Take Map Picture** to show the level itself under the marks.
- **Showing them.** A **Quest List** node, from **NodeWorthy** → **Add To Screen** on the level, draws the quests under way with a tick against each step done, for example *Find red mushrooms 2 / 3*. It hides while there are none.

Quests last the whole game, across scenes, and **Save Game** keeps them with everything else.

The elder's stew. In Mossbrook, the elder's answer *I'll find them* starts *Mushrooms for the stew*. The hero finds three red mushrooms, then brings them to her, in order. *Here are your mushrooms* is offered only while the hero carries three (an Inventory Has check), and sends GAVE_MUSHROOMS, which ticks the last step off. The quest pays the 20 gold itself, and her template takes the mushrooms and says thank you. Asked how the stew was, she offers a second quest, *The elder's old key*. The hero beats the great slime by the old cave (its Hurt Box sends BIG_SLIME_DIED, which the first step counts), picks up the key it sat on, and brings it back, for 100 XP.

6.17 Fighting with Hurt Box, Hit Box and weapons

Every game where things fight uses the same two nodes, in 2D and in 3D, whatever does the hurting, be it a sword, a fist, a slime, spikes or a bullet.

- **Hurt Box** makes something able to be hurt. Right-click the player, an enemy or a barrel that has a Machine and choose **NodeWorthy** → **Give It** → **Health**, and it gets a Hurt Box, a health and a Dead state (4.3). A Hurt Box on its own is under **Nodes To Add** in the Add Action window (10.35). When a Hit Box touches it, the damage, less its **Defence Variable**,

comes off its **Health Variable** (at least 1), and its machine is sent **HURT**, carrying how much was taken. At 0 health it is sent **DIED** instead. Give health a **Min** of 0 and it stops there. Health kept as a whole number takes whole hits. A hit flashes it in its **Flash Color**. Tick **Shows Damage** and each hit's number rises and fades over it, in its **Damage Color**. Pick a **Hurt Sound** and a **Died Sound** and they play where it is. All of these are in its Inspector. Shows Damage and the two sounds are also among the settings at the top of the Inspector of the character it is under, beside the rest of that enemy's settings.

- **Hit Box** is the part that does the hurting. **Damage** is how much, or **Damage Variable** reads it from a variable. **Hurts Group** keeps it to one side. Pick it in the scene, so an enemy's hurts the player and the player's sword hurts every copy of the enemy. It never hurts its own character.
- **Knockback and safety**. A hit knocks the one hurt back, away from whatever hit it, as hard as **Knockback** says, and it is safe from another hit for as long as its **Invincibility Time** says, blinking. Given health again, as a respawn does, it can be hurt again after two seconds' safety, so it is not beaten again where it gets up.
- **Always on, or a swing**. A slime's body always hurts. Set to **When Safe**, it hits again whenever its victim is no longer safe. A sword's Hit Box is switched on only for the swing, with **On**, and set to **Next Swing** it hits each thing once until it is switched off and on again.

Weapons. An item becomes a weapon when its **Damage** is set, under **Weapon** in its settings, with its **Reach**, **Swing Seconds** and **Knockback**. Worn in the Weapon slot with Equip Item, it is swung by the **Attack** action. Put Attack in a Swinging state the attack button leads to, going back on FINISHED. A swing reaches in front of the character, the way they last walked in a top-down game or the way they face otherwise. Whatever it touches that is in **Hurts Group** takes the weapon's damage plus the character's **Strength Variable**, once a swing. A stronger sword, or more strength, beats an enemy in fewer hits. With nothing worn, Attack is a punch. In 2D the weapon's picture is swung out in front, where the player can see it.

Bullets need nothing new. Give the bullet's scene a Hit Box, and a Hit Event that its own machine answers with Queue Free. An **Explosion** (10.11) hurts through a Hurt Box too. One swing of a great axe, a fireball or an explosion takes health off everything it reaches that has one.

Loot. The ready-made Enemy comes with a **Loot** Inventory, and its Loot settings are at the top of its Inspector. **Can Drop** lists the items it may carry, **How Many Items** how many it takes from them as the level starts (rarer ones less often), **Item Chance** how likely each is really there (35 to begin with, about one time in three), and **Gold Least** and **Gold Most** the gold. Beaten, it drops whatever it carries (**Drop Loot** in its Dead state, 10.32). Fill in the Loot Inventory's **Starts With** for something one enemy always drops, such as a boss's crown.

Both take their shape from their character's own collision shape, a little bigger, and need nothing setting up. For a sword's reach in front of a character, set **Shape** to Circle with a **Radius** and an **Offset**. With **Debug > Visible Collision Shapes** on, Hurt Boxes are drawn red and Hit Boxes gold while the game runs.

The slimes of Mossbrook. Each slime has a Hurt Box and a Hit Box that hurts the player group for 7. The hero's Hurt Box takes off their Defence of 4, so a slime takes 3 health at a time. J, a click or a gamepad's X swings the sword worn. The iron sword on the ground does 6, plus the hero's Strength, and the steel sword from the shop 16. In the hero's template, DIED takes **Walking** to **Fallen**, where they lie still for two seconds, then get up at the scene's `GetUpHere` marker with their health back. **Get Property** reads the marker's Global Position into a variable, and **Move To** reads its To from that variable.

Going inside. Mossbrook's old cave is a separate scene, `cave.tscn`, with its own hero, camera and HUD. The arch is an Area2D whose machine, when the player walks in, runs **Change Scene** to the cave with **Arrive At** `FromVillage`, a Marker2D just inside the cave's west wall. A gap in that wall goes back the same way, arriving at `FromCave` in the clearing. The hero's template has **Carries Over Between Scenes** ticked, so the pack, the sword, health and level go in and out with them, and the village remembers the slimes already beaten (see 6.19).

6.18 Levelling up with experience and levels

Beating enemies and finishing quests earns experience, or XP. Enough XP takes the character up a level, and each level makes them stronger. One node does all of it.

- **Experience.** Put an **Experience** node under the player. Right-click it and choose **NodeWorthy** → **Give It** → **Experience**, which also gives their template a number variable called `xp` and one called `level` starting at 1, where it has none of those names. Whatever adds to `xp` counts, whether a beaten enemy, a quest's reward (Reward Variable `xp`) or Set Variable with Add.
- **How much each level needs. First Level XP** for level 2, 100 as it comes, and each level after needs **Each Level More** times as much again, making 100, 150, 225. XP left over carries into the next level, and enough for several levels counts each one.
- **What a level gives. Gains Each Level** lists what each level adds. Given from the menu it comes with +5 to `max_health` and +1 to `strength`, each where the character keeps that number, and with every row taken away a level adds nothing. **Heal on Level Up** fills health up to its Max.
- **Level up!** Every machine in the scene is sent **LEVEL_UP**, carrying the new level, and a message on the HUD, a sound or a flash can each wait for it.
- **XP from enemies.** A **Hurt Box's XP Worth** is given to whoever beats it. It is among the enemy's own settings in the Inspector, under **Worth**. A skeleton can be worth 25 and its captain 120.

The **Stat Panel** shows the level, and the XP gathered against the total for the next level, as in *XP 40 / 150*. A variable name of two letters, such as `xp` or `hp`, shows in capitals.

Mossbrook. Each slime is worth 35 XP and the big slime 150, so three slimes take the hero to level 2. The **Level up!** message on the HUD has its own machine. It stays **Hidden** until LEVEL_UP, then goes to **Showing** for a moment and a half.

6.19 Carrying the player on from scene to scene

A game with more than one place, such as a village and a cave, a dungeon's floors or a platformer's levels, keeps each place in a separate scene and goes between them with **Change Scene**. Each scene has its own copy of the player, so the player can be placed and tried in every scene on its own. Something has to take the player's health and belongings from one copy to the next.

- **Carries Over Between Scenes.** Tick it at the top of the player's **Variables** tab. It belongs to the template and covers every scene whose player runs that template. When Change Scene goes somewhere else, the player's variables go with them to the player in the new scene before anything there starts. So do the changes gear and potions have made to their stats, and a potion still wearing off keeps the time it had left. So do the **Inventories** under the player, such as their pack and what they wear. The new scene's Starts With is not added on top, so the player never gets a second starting sword.
- **Arrive At.** Give each doorway a **Marker2D** or **Marker3D**, such as `CaveDoor`, and type its name into Change Scene's **Arrive At**. The player appears there, and in 3D faces the way the marker faces. A smoothed camera jumps there with them instead of sliding across the level.
- **Places remember how they were left.** The scene the player leaves is remembered while the game runs, so beaten enemies stay beaten, an opened chest stays open and a key once picked up stays gone. The machine whose Change Scene it was comes back in its first state, so a doorway does not send the player straight out again. Global variables are not put back, because a score or gold that is Global belongs to the whole game and stays as it is now, not as it was when the player left.
- **Try Again. Reload Current Scene** goes back to the moment the player walked in, with the health and gear they came in with, standing where they came in, and the scene as it was then. A player who dies is never carried into the retry with no health left. The game's global variables go back too, as they were when the scene began, so a score is what it was at the start of the level and its coins cannot be taken twice. List a global under **Keep As They Are** to leave it as it is, such as `Lives`, so losing one is not undone. Untick **Put Globals Back** to keep them all, as a game does whose settings (volume, sensitivity) are globals changed from its pause menu.
- **A new game starts fresh.** Going to a scene with no player in it to carry on, such as a title screen or a game over screen, ends the journey. Nothing is carried any further, and every remembered place is forgotten.

Global variables and quests belong to the whole game and go on regardless. A variable holding a node is left behind, because that node belonged to the old scene. If the new scene's player does not take what was carried, for example because its template is a different copy or does not have the tick, the Output panel says so and names the reason.

6.20 Data, for records and tables of your game's things

Many games keep lists of things of one kind, such as weapons with their damage and price, enemy types with their health and speed, or levels with their time limits. Data keeps them in separate files, so a number is changed in one place and every template reading it follows. Four kinds of file do it, each made in the FileSystem dock. Right-click a folder of your own and choose **Create New**, then the one you want.

- **Data Definition (NodeWorthy Data Definition...)** says what a thing of one kind has. A Weapon, for example, has a Damage, a Price and a Rarity. It opens as a grid in the NodeWorthy panel, a line for each field, and its name and words are in the Inspector. **Add Field** adds one. Give it a name, pick its **Kind** (a number, a whole number, text, On/Off, a colour, a place in 2D or 3D, a choice, or a file such as a picture or a sound), what it **Starts As**, and for a Choice the Enum its **Choices** come from. Every record already has a **Name**, such as Sword (the first column of a table, and a field Get Data can read). A new definition therefore starts with no fields, and no field can be called Name. Renaming a field keeps the values records already hold for it.
- **Enum (NodeWorthy Enum...)** is a list of named choices, such as Common, Rare and Legendary. A field whose Kind is Choice picks one, so a rarity can never be spelled two ways.
- **Data (NodeWorthy Data...)** is one record. Pick its Data Definition and its fields appear in the Inspector to fill in, such as a Sword with Damage 5. A field left alone follows the value its definition starts it at, so changing the definition changes every record that never set it.
- **Data Table (NodeWorthy Data Table...)** is many records of one kind in one file. Click it and its grid opens in the NodeWorthy panel. Pick its Data Definition in the Inspector and the fields become the columns. **Add Row** adds a row, a cell is changed where it is, pressing a column's heading sorts the rows by it (again for the other way), and **Move Up**, **Move Down** and **Remove** work on the picked row. Every change is saved at once and taken back with one undo. Each row needs a name no other row has.

Data of a node's own. Two goblins of one kind may each need their own health. Right-click the goblin in the Scene dock and choose **NodeWorthy > Give It > Data Node**, and a Data node goes under it. Pick a Data Definition and its fields show at the top of the goblin's Inspector, with values of the goblin's own. Pick a record in **Starts As** to start from its values instead.

Using it while the game runs. Four pieces in the Add Action window's **Data** group work with records. Each one's **From** says where the record is. It can be **A Data File**, **A Variable** holding

one, or **Data On A Node**, the Data node under a node (Self, the one the machine works on, unless another is picked). Their **Field** list offers the fields that record really has.

- **Get Data** reads a field into a variable, such as the sword's Damage into `damage`.
- **Set Data** changes a field, or adds to a number, for example taking 1 off this goblin's own Health. A Data file changed while playing is not saved back into the file, so values that change belong on a Data node.
- **Find Row** picks a row of a table and keeps it in a variable. It can pick by name, by place, at random, the first whose field holds a value, or the one with the highest or lowest number. Get Data and Set Data then read it with From set to A Variable.
- The **Data** check asks about a field, such as whether this goblin's Health is 0 or less, or whether the weapon is Legendary.

A value that does not suit its field, such as words in a number field or a choice its Enum has not got, is refused, in the grid and while the game runs, and the Output panel says why.

Loot from a table. A chest's Opening state runs Find Row on a Weapons table, How set to Random, keeping the row in `loot`. Get Data then reads the loot's Name into a text variable for the pickup message. To add a weapon to the game, add a row to the table.

6.21 3D Terrain, a landscape to walk over

A 3D level starts on a flat Ground. A **Terrain** gives it hills instead, with solid ground to walk on and grass that stands round the player and sways. Right-click the top node of a 3D level and choose **NodeWorthy** → **Add Thing** → **Terrain**. Rolling hills 256 metres across appear. Where the level had the flat Ground, the Terrain takes its place with its level middle where the Ground's top was, and the player, the enemies and their patrol points each stand on the new ground where they were. One undo puts it all back.



A Terrain as it comes, rolling hills with grass round the player.

Click **Terrain** on the **3D** tab and the top of the Inspector changes the ground. **Kind** picks what it is shaped as. **Hills** roll gently, a **Valley** has a floor between two sides, an **Island** has sea all round it, and **Mountains** ring a meadow with peaks too steep to walk. **Try Another** makes different ground of the same kind, as often as you like, and Undo goes back one at a time. **Use My Own Height Map...** shapes the ground from a grey picture of your own, where lighter is higher, and **Save Height Map As A Picture...** saves the heights to paint on. When the level has a **Minimap**, **Take Map Picture** is there too, to picture the level again for the map once the ground is right. Whichever you press, the middle stays where it was and everything standing on the ground stays standing on it.

Below that are its settings. **Size** is Small, Medium or Large, which is 128, 256 or 512 metres across, and **Highest Point** changes with it so the slopes stay as steep as they were. **Highest Point** sets how tall the hills are. **Grass** is None, Thin or Thick. **Grass Cover** is how much of the gentle ground the grass covers. **Everywhere** is a meadow, and **Mostly**, **In Patches** and **Here And There** leave more and more bare ground with the grass in patches across it. Turn on **Dry Patches** for some patches of yellow grass among the green. Where the ground has water, as an Island does, **Waves** is **Still**, **Gentle** or **Rough**, and the sea rolls up the shore with foam where it laps and goes on to the horizon. **Walls at the Edge** stops anything walking off the side of the world, and **Colours** holds the colours of the grass, the dry grass, the bare ground, the rock, the steep bare earth, the sand and the water. Each has its own colour, so yellow grass leaves sandy ground sandy.

Trees, boulders and flowers. **Trees**, **Boulders** and **Flowers** are each None, Mild, Medium or Heavy, and the Terrain puts them where they can stand, the same places every time for the same

ground. Trees are pines and leafy trees, never in water, on rock or on the level middle where the player starts. Boulders sit on slopes and by the shore. Flowers grow in clusters in many colours and sway with the grass. Trees and boulders are solid, so characters bump into them, and nothing bumps into a flower. **Leaf Colour** and **Boulder Colour** under **Colours** pick their colours. A Terrain from Add Thing starts with Mild of each.

A lake. Turn on **Lake** and a lake is carved where the open ground lies lowest, away from the level middle, with a shore that wanders and banks sloping down to it. **Lake Size** is Small, Medium or Large, and **Lake Depth** is how deep it is in the middle, in metres. Its edge is shallow whatever the depth, so a character can walk out of it. Nothing else floods, and turning Lake off gives the ground back exactly as it was. Try Another carves a lake into the new ground too. An Island has the sea round it instead, so it has no lake, and Lake is greyed out there. A character swims in it with **Swim**, and the **In Water** check tells when its feet, its waist or its head are under, as the recipe Swim And Hold Your Breath shows. When the camera goes under the water, everything fades into a blue green haze and the surface shines overhead, and the moment it comes up the haze is gone.

Lock it before you build on it. Once the ground is the way you want it, tick **Locked**. Kind, Try Another, Use My Own Height Map..., Size, Highest Point and the lake are greyed out, so a slip of the mouse cannot move the ground from under what you place on it. Colours, grass, waves, trees, boulders and flowers can still be changed. Untick Locked to change the ground again.

A player walks up any slope gentler than 45 degrees and stops at anything steeper as it would at a wall. That ground shows as rock. The **Platform Hero (3D)** climbs it by holding J, as it climbs any wall, quickly up steep rock and more slowly where the ground is only a little too steep to walk. An **Enemy** that sees the player comes straight over the hills. A machine using **Navigate To** walks on the walkable ground baked over the level, which follows the hills. After changing the ground, right-click the level and choose **NodeWorthy** → **Bake Navigation Mesh** again so it matches.

For one level, not a world without end. All of a Terrain is there at once, so it comes in three sizes and no bigger, and one is all a level has. For a bigger game, give each part of the world a scene and a Terrain of its own and join them with a **Level Exit**. What the player carries goes with them (6.19). Every setting, and what to do when something goes wrong, is in the Terrain's entry in Heroes, Things and Nodes.

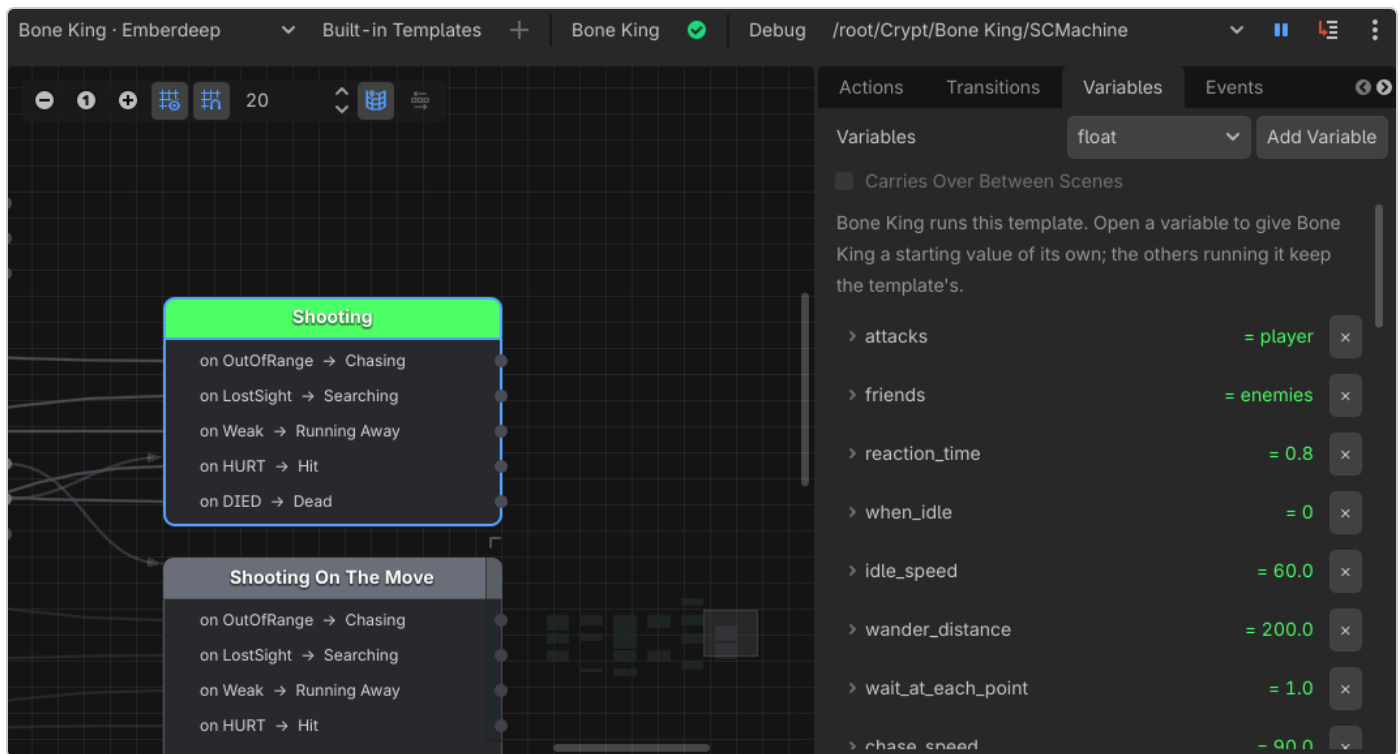
7. Testing and fixing problems

Every template has a moment when it does not do what you meant. NodeWorthy gives you three ways to find out why. It can show you a machine working while the game runs, it points out mistakes while you build, and it stops a runaway template with a message rather than freezing the game.

7.1 Watching a machine while the game runs

Seeing the game and the graph together. In the NodeWorthy view the game plays in a window of its own over the editor, and the NodeWorthy tab stays behind it, so you watch the game and its machines together. Move the game window aside to see more of the graph. In the Godot view the game can play in the **Game** view in the middle of the editor, with the NodeWorthy panel along the bottom under it. For more room, drag the NodeWorthy tab out into a separate window and put it beside the game, or on a second screen.

Debug, Pause and Step. These controls sit in the graph's toolbar and appear only while the game runs. Every machine reports itself under **Debug**. Pick one and its state lights up as it changes. **Pause** holds that machine exactly where it is, actions, timers and all, without pausing the game around it. Press it again to let it go. **Step** runs the paused machine for one frame and stops again, the fastest way to see which of two transitions is winning. Events sent to a paused machine wait until it is let go or stepped, and a timer or movement a state starts during a Step waits for the next one.



The debugger watching Emberdeep's Bone King. Debug names the machine being watched, and its Shooting state is lit because that is where it is right now. Pause, beside it, is pressed, so the king holds still while the game carries on. His variables show their live values in green in the Variables tab.

Add Breakpoint. Right-click a state and pick it, and a **II** appears in its title. While the game runs from the editor, a machine arriving at that state holds still there, as if you had pressed **Pause**, and the graph watches it. Look at its variables, then press Pause again or Step. It does nothing in the finished game.

What else the panel shows while it watches. The Variables tab shows what each of the watched machine's variables holds, in green (see 6.5), and the Events tab lists the events it has just sent (see 6.7). The Machine's **Debug Log** setting prints every state change and event to the Output panel (see 6.1).

7.2 The problems button

The problems button in the toolbar shows a tick when the template is sound, or a warning with a count of what is broken. That covers no start state, a transition with no event or nowhere to go, a name that matches no state, two states with one name, an action without a script, a state waiting for FINISHED beside an action that runs every frame and so never finishes, or beside a movement that never finishes as it is set (a Chase or Navigate To without Finish On Arrival, a Flee without Finish When Safe, a Patrol going round and round, a Wander), an action set to Wait for Previous under one that runs every frame, a Check set to Once above an action that changes the variable it asks about (it would see the value from before), states that lead straight round to one another with nothing to wait for, a Change Scene or Instantiate holding a scene that holds the same machine again, so that neither could load. It also names what would otherwise fail without a word while the game runs. That means a setting still showing (*pick one*), a node field pointing at a

node that is not in the scene, or a key, mixer bus, signal or method that nothing answers to. It means a variable of the wrong kind for the setting that reads it or keeps something in it, such as a float picked where a Vector3 is needed (both kinds are named), and an Expression formula that cannot be read. It also means a node an action cannot work without, such as the Perception node Look For Target sees with, the navigation agent Navigate To finds its way with, or the AnimationPlayer Play Animation plays from. It lists a node or a Who setting holding something that cannot work for it, and a number outside what its setting takes, as one saved before the setting had its range. When a setting reads its node from a Node variable, it checks the node the variable starts as and says where to change it, and it names a starting node that is no longer in the scene. Last, it spots a background picture or coloured panel left on Stop in front of a node waiting to be clicked, because that would take every click for itself. The count follows the scene as you change it, so a node deleted or a Target changed shows at once. Those about the scene are checked against the Machine selected in the Scene dock, or the first one running the template, so open the scene to see them. Press it to list them. Broken rows are red in the graph as well.

7.3 Can a machine get stuck in a loop?

Can a machine get stuck in a loop? Events are acted on the moment they are sent, so states can lead straight on to one another within a single frame. That is what makes a machine quick, but two states that send each other straight back, such as two states with nothing to do whose FINISHED transitions point at each other, would go round for ever. The machine counts the states it enters in one run of moves, and past 50 it stops, printing a message naming the state it was entering, so a template mistake turns into something you can read instead of a game that locks up. Each event arriving from outside starts a count of its own. A bomb that takes a hundred enemies at once sends the score a hundred kills in one frame, and taking each of them is not a circle. You should rarely get that far, because the graph spots such a circle while you build it, paints its states red and names them under the problems button. A **Wait** or a **Next Frame Event** in one of the states breaks such a circle. The limit is the template's **Loop Increment Limit**, in the Inspector when the template file is selected (click it in the FileSystem dock). Raise it only for a template that really does pass through that many states in one run of moves.

7.4 A checklist for when a machine does nothing

Most "it does nothing" moments come from a short list of causes. Go down it in order.

1. **Look at the problems button.** If it shows a warning, press it and fix what it lists (7.2).
2. **Watch the machine.** Run the game and pick the machine under **Debug** in the toolbar (7.1). If no state lights up, the machine is not running. Check that the Machine has a Template and that **Autostart** is on (6.1), and that its Process Mode, or that of a node above it, is not Disabled (6.11).
3. **If it sits in one state for ever,** look at what that state is waiting for. A transition on FINISHED never fires while any action in the state is on **Every Frame**, including a Check

(6.3), or is a movement that never finishes as it is set, such as a Chase without Finish On Arrival. Send an event after a Delay to move on instead. An event name that is spelt differently, even by a capital letter, is a different event (6.4). And an event that arrives while the machine is in a state with no transition for it is thrown away, not saved for later (6.4).

4. **If a Check never says yes**, look at the variable it reads in the Variables tab while the game runs (6.5), and at the question's settings (6.6).
5. **If the states change but nothing moves on screen**, the problem is usually in the scene rather than the graph. Check the action's **Node** (6.3). For a physics body, check that the body is the top node with its shape and mesh under it (8.1). A walking character needs a **Move Character** action to move at all (10.12).
6. **If a click or a touch is ignored**, check that no picture or panel set to **Stop** sits in front of the node (7.2), and that the body has a collision shape if it needs one (2.6).

8. How-tos, from small to big

These worked examples start with one thing and end with a whole game. Each one uses only pieces explained in the chapters before, and names the actions it uses so that you can look them up in chapter 10. In the games that ship with NodeWorthy (8.4 to 8.8), click any **Machine** in the Scene dock to see how it is built.

8.1 How to make a ball that hops (something physics moves)

A ball, a crate, a barrel, or anything else that falls, rolls, bounces or gets pushed is moved by Godot's physics, not by you. Setting one up has one trap that catches nearly everybody the first time. Here it is step by step.

The idea in plain words. In the Scene dock, a node placed under another one is carried along by it, the way a sticker goes wherever the box it is stuck on goes. It never works the other way round, so moving the sticker does not move the box. Physics only ever moves one node, the **RigidBody3D**. Everything you want to move with it has to be *under* it.

- **RigidBody3D** is the real thing, the part with weight that falls and gets pushed. On its own you cannot see it.
- **MeshInstance3D** is its skin, the shape you see. It has no weight and does nothing by itself.
- **CollisionShape3D** is how big the body is to physics. Without one, the body falls straight through the floor.
- **SCMachine** is the brain, and its jobs happen to the node it sits under.

The right way round

Ball	RigidBody3D	← the real thing; physics moves this
└ MeshInstance3D		← the skin, carried along
└ CollisionShape3D		← its size, carried along
└ SCMachine		← the brain, pushing its parent

The trap is starting from a shape you can already see and adding the body under it.

Ball	MeshInstance3D	← the shape you see; nothing moves it
└ RigidBody3D		← physics moves this, and it is invisible
└ CollisionShape3D		
└ SCMachine		

Here everything works, but you cannot see it working. The machine changes state and pushes the body, and the body flies up, but it is invisible. The shape you are watching stays still, because

it sits *above* the body and nothing carries it. If a physics body "does nothing" when you run the game, look at this first.

Building it

1. Your floor needs a body too, or the ball falls through it. Under the floor's mesh, add a **StaticBody3D**, and under that a **CollisionShape3D**. In the Inspector, set its **Shape** to **New BoxShape3D** and make it the size of the floor. A static body never moves, so here it is fine for it to sit under the mesh.
2. Select the scene's top node, press **+** in the Scene dock and add a **RigidBody3D**. Rename it "Ball". In the Inspector, under **Transform**, set its **Position** to where the ball should start, a little above the floor.
3. With Ball selected, add a **MeshInstance3D**, and set its **Mesh** to **New SphereMesh**. Leave its Position at 0, 0, 0 so the skin sits exactly on the body.
4. With Ball selected again, add a **CollisionShape3D** and set its **Shape** to **New SphereShape3D**. Its size should match the mesh.
5. Right-click **Ball** (the body, not the mesh) and choose **NodeWorthy → Add State Machine....** Call it "Ball". The new SCMachine sits under the body, and its **Target** reads Parent (RigidBody3D).
6. Add a state called "Hop". Give it an **Apply Force** action with Kind **Impulse** and Force **0, 5, 0**, which is straight up. Then add a **Wait** of 1.5 seconds.
7. Right-click Hop and choose **Add Transition**. On the Transitions tab press **When** and pick **When everything in this state has finished**, which is FINISHED, then drop the row's port back on Hop itself (or pick Hop in the transition's **To State** list), so that when the wait is over the state starts again.
8. Run the game. The ball jumps about a metre, lands, and jumps again.

Already built it upside down? You do not need to start again. In the Scene dock, drag the RigidBody3D onto the scene's top node, then drag the mesh onto the RigidBody3D. Give the body the position the mesh had, set the mesh's Position back to 0, 0, 0, and rename them so the body carries the thing's name.

Good to know. A push is added to whatever speed the body already has, and gravity takes away about 10 each second. An impulse of 5 on a normal-weight body is a hop. Much bigger pushes, or pushes that come faster than the body can fall back, add up, and it keeps climbing until it is out of sight. If your body vanishes upwards, use a smaller push or a longer Wait. And always put something that takes time, such as a Wait, between pushes. A state that pushes and starts again with nothing to wait for goes round for ever, and the graph paints it red to tell you.

If it still does not move

- Is the **RigidBody3D** the top node of the thing, with the mesh under it?
- Does the body have a **CollisionShape3D** with a Shape set? The Scene dock shows a warning triangle when it does not.
- Is **Freeze** ticked on the **RigidBody3D**? A frozen body ignores every push.
- Run the game and pick the machine under **Debug**. If the states light up in turn, the machine is working, and the problem is in the scene, not the graph.

Everything here works the same way in a flat game with **RigidBody2D**, **Sprite2D** as the skin and **CollisionShape2D**. In 2D, up is a negative Y, so the push is `0, -300`, because pixels are much smaller than metres.

8.2 Small patterns worth knowing

These short patterns come up again and again. Each is a handful of actions.

Checkpoints. Put the checkpoints under one node, and give the player an **Event On Collision** watching areas, with **Only** set to Everything under that node and **Store Node In** set to a `checkpoint` variable, so the variable always holds the last one touched. On dying, a **Move To** with Duration 0 and Var pressed on To, reading `checkpoint`, puts the player back there, because a node in a variable counts as the place it stands.

Dropping through a platform. Give the platforms a layer of their own. Make a state for the drop, which the player's machine goes to when Down and Jump are pressed together, and add a **Set Collision** action to it. Set the action's Change to **Remove Collision Mask**, which shows its Layers. In Layers, choose **Name a new layer...** and call it *Platforms*, which ticks it. World is ticked there to begin with, so untick it and leave Platforms the only layer ticked, or the player falls through the ground as well. Then right-click each platform and choose **NodeWorthy** → **Collision Layer** → **Platforms**. The ready-made player still stands on them, as it bumps into every layer but Pickups. Leave Node empty, so the action works on the player itself, and set For Seconds to `0.3`. Give the state a transition on FINISHED to **In The Air**, where the ready-made player falls, because a state with no movement action leaves the player hanging where it is (see 10.11). The player stops bumping into Platforms and falls through, and it lands on the next platform down because by then it bumps into Platforms again. A 2D game's layers and a 3D game's are named apart, and NodeWorthy shows the ones the game uses.

A first-person player. A **CharacterBody3D** with a collision shape and a **Camera3D** under it, lifted to head height. Its machine has one state with **Mouse Look** (with Head set to the camera), **Get Input Vector** reading **Key: A**, **Key: D**, **Key: W** and **Key: S** into a variable, and **Move Character** following that variable with **Space** set to **Local**. The player walks the way they look.

A third-person player. The same body without the camera under it. The camera sits in the scene behind the player, with a machine running **Orbit Camera** around the player. The player's **Move**

Character has **Space** set to **Camera** and **Turn to Face** ticked, so pushing forward walks away from the camera and the character turns the way it walks.

A button that sends an event. A button, a slider or any other node becomes an event with **Event On Signal** listening to one of its signals, such as a button's `pressed`. The events it sends are what the rest of the template reacts to (see 10.9). For the node the machine itself sits on, a transition can wait for **pressed** under Signals with no action at all (see 6.4).

An Options menu. A volume slider sends `value_changed` through **Event On Signal**, with **Store Argument In** set to a variable. **Set Volume**, with **Measure** set to **Volume Linear** (0 to 1) and its **Volume Linear** reading that variable, sets the Music or Effects bus. Music, Effects and Voice need nothing set up. A game that uses one the Audio tab does not have makes it as it runs, going into Master, so a game shipped to another project keeps its sliders working. Play Sound, Play Random Sound and Play Sound At Position start on Effects, and Play Music on Music, so every sound and tune answers to the sliders with nothing set. A fullscreen tick box runs **Screen Settings**, a language button **Set Language** with To The Next One, and each row of a Controls page **Rebind Input**. Keep the choices in variables and **Save Game** them, and load them as the game starts.

8.3 How to make a twin stick shooter

Walk with one stick, aim with the other, hold the trigger to fire, and clear waves of enemies. Every part is an action, and there is nothing to write.

- **The player** is a **CharacterBody2D** with a **Marker2D** called Muzzle a little in front of it. Its machine has an **Aiming** state with two **Get Input Vector** actions (the left stick into `move`, the right stick into `aim`), a **Move Character** reading `move` with Falls off, and a **Look At** with Face **A Direction** and Var pressed on Direction, reading `aim`. For mouse aiming, use Face **The Mouse** instead.
- Add an **Event On Input** for `Pad: Right Trigger Pressed`, sending FireDown, and a transition to a **Firing** state. Firing has the same walking and aiming actions, a **Instantiate** of the bullet scene At A Node, the Muzzle, with **Every Seconds** `0.25`, and an Event On Input for the trigger Released that goes back to Aiming.
- **The bullet** is an **Area2D** scene. Its machine has **Translate** forward, a **Queue Free** with a Delay of a second so misses do not stack up, and **Event On Collision** with Only set to every copy of the enemy, storing what it hit in `hit`. On that event a state uses **Machine Variable**, Add To Theirs, Amount `-1` on their `health` with Var reading `hit`, a **Send Event** Hurt with Reach A Node reading `hit`, and a Queue Free.
- **An enemy** is a **CharacterBody2D** in the `enemies` group with a `health` variable. It uses **Find Child** to put the player in a variable, **Move Character** Toward A Node to chase, and a **Check** on health at or below 0 that goes to a state adding 1 to a global `score` with **Math**

and destroying itself. On Hurt, a **Vector** Direction from the player to itself, scaled up, fed to **Apply Force** Impulse, knocks it back while it keeps chasing.

- **Waves** come from a spawner with spawn points as its children. A **Repeat** reading the `wave` variable rings round a state that **Spawns** an enemy At One Of The Nodes Under the spawn points. When it is done, a **Group** action counts `enemies` every frame and its Empty Event leads to a state that adds 1 to `wave` and starts again.

8.4 How to make a first person shooter

The games are for looking at, not for changing. The games that ship live inside NodeWorthy's own folder, and an update replaces that folder whole, so anything changed in them is lost. Play them and take them apart in the panel as much as you like. To build on one, start a new game from it (see 1.1), or make your own copy of a template with **Make It Mine**.

NodeWorthy ships a first person shooter made the same way, with no scripts at all, in `addons/nodeworthy/games/first_person_shooter/`. Open `menus/title.tscn` and press **Run Current Scene** to play it from its title screen, or run `arena.tscn` to go straight into the level. It is a whole level. You fight through the courtyard, find the red keycard, open the vault, and reach the exit, with pickups, a score, a checkpoint and lives, and the menus a player expects around it, such as options, key choices, a pause menu and best scores kept between runs.

- **The player** is a CharacterBody3D with a Head node, a View node under it and the Camera under that. Its **Playing** state runs **Mouse Look** on the camera, **Get Input Vector** reading `Key: W` or `Pad: Left Stick Up` and the other three ways, **Move Character** with Space set to **Local** and Speed read from a `speed` variable, and **Head Bob** on the View. Three parallel states run inside it. **Movement** has Standing, Walking, Sprinting, Jumping, Falling and Crouching. Each sets `speed` and the crosshair's Spread, and its Checks move between them. **Aim** zooms the camera in with **Camera Zoom** while the right mouse button is held, halves the look speed and tightens the crosshair. **Target** casts a **Raycast** from the camera and turns the crosshair red over anything in the `enemies` group.
- **Crouching** eases the Head down and the collision shape shorter with **Tween Property** (the property `shape:height` reaches inside the shape). Standing up first casts a short **Raycast** upward, so the player cannot stand up under a low roof.
- **The player** has a Health parallel state. A hit takes the damage off, flashes the screen red, and at none the game shows YOU DIED and puts the player back at the start.
- **The rifle** is a few boxes under the camera with Muzzle and Ejection markers, and its own machine. **Ready** fires while the button is held. **Firing** takes a round, spawns a muzzle flash and a casing, plays the shot with **Play Sound At Position**, kicks the view with **Recoil**, and blooms the crosshair. It reads the crosshair's **Opening** into a variable and uses it as the **Raycast**'s Spread, so shots scatter as widely as the crosshair shows. A hit spawns a tracer,

sparks and a bullet hole standing on the surface (**Stand On**, reading the Raycast's normal), and a hit on an enemy sends HIT carrying the damage, more for the `head` group.

Reloading tips the rifle down, fills the magazine from the reserve with **Math**, and is cancelled by firing.

- **The targets** share one template. HIT keeps the damage in a variable, **Math** takes it off their health, and at none they fall over with **Tween Property**, wait, and stand up again.
- **The arsenal.** The pistol, shotgun and rocket launcher each began as a copy of the rifle's template (**Start From**, *A copy of Assault Rifle*) and were changed from there. The pistol fires when the button is *just pressed*, once a pull. The shotgun's **Firing** leads into a **Repeat** of nine. Each go is a **Raycast** with Spread 6, whose hit sparks, marks and hurts, so a close shot lands many pellets. Its machine has **Physics Process** ticked, so all nine go out in the same step, and **Pump** slides the fore-end back and forth with two **Tween Property** actions. The launcher spawns `effects/rocket.tscn` at the camera. The rocket's own machine runs **Translate** and a short **Raycast** ahead, and on a hit spawns `effects/explosion.tscn` where it struck, stood off the surface with **Stand On**. The explosion starts its three particle bursts with **Emit Particles**, hurts with two **Explosion** actions (enemies, and the player less), shakes the camera and fades its light.
- **Switching weapons** is one more machine on the Camera, called **Arsenal**. Its **Armed** state has two parallel states. **Weapon** holds **Holding**, whose Checks read 1 to 4, the mouse wheel and the pad's Y button. Each leads through a small state that sets the `slot` variable, then **Lowering**, which pauses every weapon with **Enable Machine** and tips them down, and back to Holding, whose sub states pick the one in `slot`. They show it, hide the others, pause all four and resume that one, name it on the screen and bring it up. Holding the pad's left bumper opens **Wheel**. It starts `picked` at the weapon in hand with **Set Variable** (Value read from `slot`), shows the weapon wheel, slows time with **Set Time Scale**, and sets the player's stick look speed to nothing with **Machine Variable**. **Get Input Vector** reads the right stick into `wheel`, an **Expression** turns it into the quarter it points at, pushed at least half way, and **Switch** sends that quarter's event, moving between four sub states that light up a slot and set `picked`. Letting go takes `picked`, so a tap with the stick left alone keeps the weapon in hand. **Grenades** throws one on G or the right bumper. Instantiate stores the grenade in a variable, **Next Frame Event** lets it join the scene, and **Apply Force** (Local Space) throws it the way the camera faces. The grenade's own machine has a Fuse parallel state that waits and explodes, and a Bounces one that plays a knock on each **Event On Collision**. Dying sends HOLSTER, which puts every weapon away, and playing again sends DRAW. While the wheel is open it writes the player's `stick_scale`, not its look speed itself, so the player's own sensitivity still applies when it closes.
- **The effects** are small scenes, each with a machine that removes it again. The flash goes after a moment, the tracer as it fades, sparks and casings after a second or two, and bullet holes after they fade. So firing for as long as you like leaves nothing behind.
- **The soldiers** are one scene, `enemies/soldier.tscn`, with a Perception node, a NavigationAgent3D, a gun with a Muzzle marker, and a head on a collision layer of its own,

so shots can tell a head from a body. They share one **Soldier** template. **Patrolling** walks the route named in the `route` variable and watches with **Look For Target**. Spotting the player, or hearing a shot (the rifle's **Alert Allies** tells soldiers within earshot), leads to **Combat**, which tells the others and holds the fight in sub states. These are **Take Cover**, a pause in cover, stepping out with **Keep Distance** until **Can See** says yes, then a burst of shots, each a **Look At** on the gun, a **Raycast** with Spread and a **Send Event** to the player when it hits. After a burst **Random Event** picks between cover, flanking with **Strafe Around**, and firing again. **Target Lost** leads to **Search Last Known**. The runner is the same template with `melee` ticked under Starting Values, which sends it to **Chase** and strike instead, and the heavy has more health and longer bursts. A dying soldier turns its collision shapes off, falls, drops an ammunition box and is removed a few seconds later.

- **Pickups** are small scenes in `pickups/`, each an Area3D with its own machine. **Event On Collision**, only for the `player` group, stores who walked in. A health pack reads the player's health with **Machine Variable**, leaves itself where it is at full health (trying again each second while the player stands on it), and otherwise adds 35, caps it at 100 and disappears. Armour works the same way. The player's **Taking Damage** lets armour soak up most of each hit with a few **Math** actions. The keycard writes `has_red_key` on the player. A weapon pickup finds the camera with **Find Child**, tells its **Arsenal** the weapon is owned, adds ammunition to the weapon and sends it TAKE_3 or TAKE_4 so it comes straight into the hands. Until then the Arsenal skips that slot.
- **The vault door** reads `has_red_key` from whoever walks into its Sensor area, and either opens with **Tween Property** and tells the level, or says why not and tries again. **The checkpoint** stores its own position with **Get Property** and writes it into the player's `respawn_point`, which the player's **Dead** state moves back to, a life fewer. Falling out of the level counts as dying too. The player's **Playing** state checks its height with a **Property** check and sends DIED once it is 25 metres below the floor, so nobody falls for ever. **The exit** sends LEVEL_DONE.
- **The level** has its own machine, **Level**, on the scene's root, which is in the group `level` so everything else can reach it with **Send Event** set to **A Group**. Its **Playing** state writes the score and the timer on the screen and holds five parallel states. **Clock** runs a **Timer** with Count Up ticked. **Combo** waits for KILLED, which a dying soldier sends carrying its `points`. Each kill adds one to `combo` (at most five), multiplies the points by it with **Math**, and waits three seconds, and another KILLED in that time starts the wait again. **Bonus** adds 50 for each HEADSHOT the rifle and pistol send. **Messages** shows whatever a PICKUP event carries for a moment. **Objective** moves from finding the key to opening the vault to reaching the exit as KEY_TAKEN and VAULT_OPEN arrive. The events it listens for are written down in its Events tab, with what each carries, so the transitions can store it. LEVEL_DONE leads to **Won** and OUT_OF_LIVES to **Lost**. Each pauses the game with **Pause Game**, shows its screen with the score, time and kills, and waits for the button's `pressed` signal with **Event On Signal** to start the scene again with **Change Scene**. Winning first goes through **Checking Win**, which loads the saved best score and time,

keeps the larger score with **Math**, and takes the new time when it is the first win or faster. Both end screens save them again, show them, and have a **Main Menu** button.

- **The title screen**, `menus/title.tscn`, has one machine, **Title Screen**. **Starting** frees the mouse and loads the best score and time with **Save Game**. **Press Any Key** waits for `Any Key` or `Button`. **Main** shows the four buttons and puts the focus on Play with **Grab Focus**, so the arrow keys, the D-pad and the mouse all work, and each button's `pressed` signal is an event. **Level Select** writes the best score and time on the arena's card and shows levels 02 and 03 as disabled buttons, locked for now. The arena's card leads to **Starting Level**, a **Change Scene** to `arena.tscn` by path, with `menus/loading.tscn` as its Loading Screen. **Credits** and **Quit Check** are pages of their own, and Escape and the pad's right button go back from each.
- **The Options page**, `menus/options.tscn`, is one scene used twice, on the title screen and inside the level's HUD. Its machine, **Options**, loads the settings as it starts, sets the Master, Music and Effects volumes with **Set Volume** and goes fullscreen if that was chosen, so the settings take hold before anything plays. It opens on `OPEN_OPTIONS`, which the title screen or the pause menu sends to the group `options`. Its **Open** state puts each slider and tick box where its variable is, and holds two parallel states. **Pages** shows one tab at a time, changed by clicking a tab or with the pad's bumpers. **Settings** keeps each slider's value in its variable (**Event On Signal** with **Store Argument In** and no event), moves the volume the moment a volume slider moves, and writes the values beside the sliders with `{look_sensitivity:0.00}x` and `{master_volume:%}`. Closing saves the settings, sends `SETTINGS_CHANGED` to the player and `OPTIONS_CLOSED` back to the menu that opened it. The settings are global variables, so the player reads the same `look_sensitivity`, `invert_look` and `field_of_view`, and loads them itself as the level starts.
- **The Keys page** is twelve copies of one small scene, `menus/key_row.tscn`, each with its own **Key Row** machine. The only difference between them is the row's `control` variable, set under Starting Values (tick **Editable Children** on the copy to reach its Machine). For walking forward it is `Key: W` or `Pad: Left Stick Up`, and so on. **Rebind Input** reads that variable, shows the keyboard key in one box and the gamepad input in the other, and waits for a new one of that kind when a box is clicked, so changing a key never loses the gamepad button. While a box waits, B and the bumpers count as the new button, so the words along the bottom change to `ESC / SELECT CANCEL`. The row's two waiting states set the global `back_hint` with **Set Variable**, **Showing** sets it back, and the Options page's **Open** state shows it with **Set Text**. **Reset to Defaults** puts every control back and sends `REFRESH` to the group `key_rows`.
- **The pause menu** is a Control in the HUD with its own machine, **Pause Menu**. Escape or Start leads to **Pausing**, which runs **Pause Game** (the menu's Process Mode is Always, so its buttons work), and **Paused** offers Resume, Options, Restart Level and Main Menu. Once the level is won or lost it sends `LEVEL_OVER`, and the pause menu stays out of the way of the end screens.

- **Sound and feel.** The player's **Head Bob** sends STEP and LANDED. A **Footsteps** parallel state plays one of four footsteps with **Play Random Sound**, and a landing thud with a small **Camera Shake**. Each weapon has a **Weapon Sway** in Playing. **Hit Direction** waits for PLAYER_HIT, stores who sent it with **Get Event Data**, and shows the red arrow with **Off Screen Indicator** set to **Around The Middle**, fading it with **Tween Property**. **Low Health** compares health with 35. Below it the screen's edges redden (a TextureRect with a round gradient, eased with Tween Property) and a looping heartbeat plays. Over an enemy the **Target** states set `aim_friction` to a half, which slows the right stick and not the mouse. The level plays its music with **Play Music** on the Music bus and keeps a three-line kill feed from KILLED and HEADSHOT with **Format String**. Soldiers shout on spotting the player, grunt when hit and cry out as they fall. Sparks crack on what they hit, and the menus click. Every sound goes through the Effects or Music bus, so the Options sliders reach all of it. The sounds and music were made for the game and ship with it in `sound/`.

8.5 How to make a 2D roguelike (Hollow Spire)

NodeWorthy ships a top-down 2D roguelike made the same way, with no scripts at all, in `addons/nodeworthy/games/hollow_spire/`. Open `menus/title.tscn` and press **Run Current Scene** to play it from its title screen, or run `floor.tscn` to go straight into a run. You climb three floors, each laid out afresh from rooms drawn by hand and each ending in a boss whose bullets fill the screen. On the way there are forty items that change how you shoot, hearts, coins, keys and bombs, a shop, treasure and secret rooms, a collection that remembers every item you have ever found, a second character to unlock, and the menus a player expects around it. W A S D walks, the mouse aims, the left button fires, Space dashes, Shift slows you down for fine dodging, Q uses your item, E drops a bomb and Tab opens the collection. A gamepad works throughout.

- **The player** is a CharacterBody2D in the group `player`, with a Sprite, a small Hurtbox area, a Dot that marks it, and an Aim node carrying an Arrow. Its machine, **Player**, spends the whole run in **Alive**, which holds eight parallel states, one for each thing the player does at once. Alive itself runs a few actions of its own. **Camera Follow** has its Offset read from a `lead` variable, which a **Vector** action keeps at the aim times 22, so the view leans toward where you are aiming. A **Look At** turns the Aim node along `aim`. A **Camera Zoom** set to show 216 pixels of height in whole steps keeps the pixel art crisp at any window size. And a **Clamp** keeps `health` between 0 and `max_health`.
- **Walking and dashing** are the **Movement** parallel state. **Walking** has an **Get Input Vector** into `move` and a **Move Character** with Falls off and its Speed read from `pace`. A Check on Space, the right mouse button or the pad's bottom button leads to **Choose Dash**, which dashes the way you are walking, or the way you are aiming when you are standing still. **Dashing** turns the Hurtbox off for a quarter of a second with **Set Collision**, so a dash passes through bullets, moves at 380 for 0.14 seconds, and **Cooldown** counts 0.3 seconds with **Timer** before the next one.

- **Aiming** swaps between **Mouse Aim** and **Pad Aim** by itself. **Last Device** writes which one was touched last into `device`, and a Check on it changes state. Mouse Aim shows a reticle with **Set Mouse Mode**, reads **Mouse Position** and makes `aim` the direction from the player to it with **Vector**. Pad Aim hides the pointer, shows the Arrow, and reads the right stick with **Get Input Vector** and **Keep Last Direction** ticked, so the aim stays where it was when the stick is let go. **Looks** plays the idle, walk, dash and hurt animations with **Play Sprite Animation** and turns the sprite with **Flip Sprite**. **Focus** halves `pace` and shows the Dot while Shift or the left bumper is held.
- **The gun** is the **Gun** parallel state, with **Holding Fire** and **Firing**, which is one **Bullet Pattern** from the Aim node. Its Count, Spread, Every Seconds, Damage, Speed, Pierce, Bounces and Homing all have Var pressed and read the player's variables `shots`, `shot_spread`, `fire_delay`, `damage`, `shot_speed`, `pierce`, `bounces` and `homing`. That is the whole trick behind the items. An item never touches the gun. It changes a number, and the gun reads the number.
- **The bullets** in `bullets/` are plain scenes made of an Area2D, a shape and a picture, with no machine. Bullet Pattern moves them and looks after what they hit. The collision layers keep everything apart. The player's bullets touch walls and enemy hurt boxes, enemy bullets touch walls and the player's Hurtbox, and each pattern's **Hits** (the enemies or the player) says whose machine is sent HIT, carrying the damage.
- **Getting hurt** is the **Hurt** parallel state. HIT leads to **Hit**, which brings a **Camera Shake**, the Hurtbox off for a second so one bullet cannot take two hearts, a heart off `health`, and a Check on none left. **Downed** looks at `lives`. With one to spare, **Reviving** takes it, fills the health, spawns a blast and gives two safe seconds. With none, **Out** sends PLAYER_DIED, a global event, which takes the Player to **Dead** and the floor to its summary.
- **The enemies** in `enemies/` are each a CharacterBody2D in the group `enemy` with a Hurtbox, a Touch area and an Eye to fire from, and every kind is built on the same four parallel states inside **Alive**. **Hurt** reads the damage with **Get Event Data**, takes it off `health` with **Math**, flashes the sprite by setting `self_modulate:g` and easing it back, and knocks the enemy away from the player with **Vector** and **Apply Force**. **Touch** sends HIT to whatever walks into it, stored by **Event On Collision**. **Rank** reads a `champion` variable. `imp_champion.tscn` and `boar_champion.tscn` are the same scenes with `champion` ticked under Starting Values, which tints them, grows them and gives them two and a half times the health. **Behaviour** is the only part that differs. **Dying** turns the collisions off, spawns a burst, rolls a **Chance** Check for a coin and another for a heart, sends ENEMY_KILLED and removes itself.
- **Each kind's Behaviour** is short. The imp and the slimelet **Chase** the player with **Keep Apart** beside it, so a crowd spreads out and does not stack into one spot. The slime does the same, and its Dying spawns two slimelets. The puff drifts slowly with a **Bullet Pattern** ring that turns a little each shot. The sentry stands still and fires a fan of three at the player. The boar stalks until a **Distance** Check says the player is near, shows its tell while it stores the direction, charges along it with **Move Character**, and is stunned when a **Touching**

Check finds a wall. The cultist holds its ground with **Keep Distance** and every few seconds **Spawns** two imps, each with a **Warning Scene** that marks the spot first, and a **Group** count stops it when the room is already crowded.

- **The bosses** in `bosses/`, the Warden, the Brood Mother and the Hollow King, share one shape. **Alive** sends `BOSS_APPEARED` and keeps `share`, the health divided by the most it can be, up to date with **Math** every frame. A **Phase** parallel state watches `share` pass 0.66 and 0.33 and sends `PHASE_TWO` and `PHASE_THREE` with a roar and a shake. The **Attack** parallel state is a ring of states, each one or two **Bullet Patterns** (rings, spirals made with **Turn Each Shot**, aimed fans, walls with a **Gap** to slip through, slow orbs with **Homing**) and a **Send Event** `NEXT` with a Delay that moves on to the next. The phase events jump the ring to harder patterns. Dying sends `BOSS_DEFEATED`.
- **A floor** is made by the **Floor** machine on the root of `floor.tscn`. **Starting** loads what has been unlocked with **Save Game**, clears the held items with **New Run**, and picks a seed with **Random Seed**, which is shown on screen so a good run can be played again. **Floor One**, **Floor Two** and **Floor Three** each run **Build Dungeon** with the start room, the three fight rooms in `rooms/`, six, seven or eight of them to use, that floor's boss room, and the treasure, shop and secret rooms. **Playing** keeps a **Clock** and a **Tally** of kills and items in parallel states, fed by the global events `ENEMY_KILLED` and `ITEM_TAKEN`. `NEXT_FLOOR`, which the stairs send, leads on to the next floor, and `BOSS_DEFEATED` sets `warden_beaten`, `brood_beaten` or `king_beaten` and saves. Winning or falling leads to **Summary**, which works out minutes and seconds with **Math**, adds a line to the `history` text with **Format String**, saves, and waits for R to climb again or T for the title. Everything a floor leaves lying about is in the group `litter`, and one **Queue Free** set to that group clears it before the next floor is built.
- **The rooms** are scenes of one size with a Floor and a Walls tile layer, four doors, an Inside area, an Enemies node and a Things node. The enemies are placed by hand under Enemies, whose **Process Mode** is Disabled, so they and their machines wait until the room wakes them. Every room runs the same **Room** template. Walking into Inside keeps the camera within the room with **Camera Limits**, then Checks read `room_kind`, which Build Dungeon handed over. A fight room sends `LOCK` to its four doors, sets Enemies' `process_mode` to 0 with **Set Property**, and counts the `enemy` group with **Group, Only Under** its own Enemies node, until the Empty Event says the room is clear. Then it sends `UNLOCK` and the global `ROOM_CLEARED`, and a **Random Event** sometimes drops a chest. A boss room does the same with boss music from **Play Music**, and leaves the stairs and a gold chest.
- **A door** is a StaticBody2D in the group `door` with a Brick picture and a Bars picture. Its **Deciding** state reads `connected` and `secret`, which Build Dungeon also hands over. With no room behind it, it becomes wall. Otherwise it is **Open** until `LOCK` arrives, and barred until `UNLOCK`. A secret room's door shows a cracked wall with **Set Texture**, and opens when the player leans on it for most of a second (**Event On Collision** for touching, and another for no longer touching, which cancels it) or when a bomb's **Explosion** reaches it.

- **Pickups and chests.** A coin, key, bomb or heart is an Area2D whose machine waits for the `player` group with **Event On Collision**, adds one to the global `coins`, `keys` or `bombs` (a heart adds to the player's health with **Machine Variable**), plays its sound and removes itself. The chest's machine serves three scenes, and `chest_locked.tscn` and `chest_gold.tscn` only tick `locked` or `gold` under Starting Values. A locked chest needs a key and takes it. A plain one rolls a **Random Event** for a key, a bomb, a coin or a heart, and a locked or gold one spawns a pedestal with an item on it.
- **The items** are files in `items/`, made in the Inspector, not machines. Each has a name, a line, a picture, a rarity, a price and a list of **Changes**. Twin Shot adds one to `shots`, adds 8 to `shot_spread` and multiplies `damage` by 0.85, and the gun reads all three. The four in `items/unlocked/` name a saved variable under **Unlocked By**, so they only turn up once that boss has been beaten. A **Pedestal** chooses one with **Offer Item**, shows it with **Show Item**, and on a touch hands it over with **Give Item**, which makes the changes on the player and sends ITEM_TAKEN carrying the item. The shop's pedestals are the same scene with `for_sale` ticked. They show the price, compare it with `coins`, take the coins with **Math**, or flash the price red when you cannot pay.
- **Active items and bombs** are the player's **Items** parallel state. ITEM_TAKEN stores the item in a variable, and when Show Item says it is an active one it becomes the `active item`. Each ROOM_CLEARED adds one to `charge`. Q runs **Use Item**, which sends the item's own Use Event. NOVA fires a ring of 24 shots, TIME_SLOW runs **Set Time Scale** at 0.4 for three seconds of real time, AEGIS turns the Hurtbox off for three seconds, and BOMB_SACK adds three bombs. E spawns `effects/bomb.tscn`, whose machine waits out its fuse and then uses two **Explosion** actions, one hurting the `enemy` group and one sending BOMBED to the `door` group.
- **The HUD** has one machine. **Showing** writes the coins, keys and bombs, reads the player's health, lives and charge with **Machine Variable** every frame, and sets the **Value** and **Maximum** of two **Icon Meter** nodes for the hearts and the lives. A **Minimap** node draws the floor with nothing to set but its Player. Four parallel states run inside it. **Popup** names each new item for a moment, **Boss** finds the boss on BOSS_APPEARED and fills a bar from its `share`, **Book** opens the collection on Tab, and **Pause** opens the pause menu on Escape or Start, with the run's numbers, the items held (an **Item Collection** set to **Held This Run**), a bigger map, Options and Quit To Title. Book and Pause each step aside while the other is open, so the two can never overlap.
- **The title screen**, `menus/title.tscn`, has one machine, with a state for each page. They are **Main**, **Character Select**, **Collection** (an **Item Collection** node showing everything ever found), **History** (the saved `history` text, the number of runs and the deepest floor), **Credits** and **Quit Check**. Each button's `pressed` signal is an event through **Event On Signal**, **Grab Focus** puts the focus somewhere sensible on every page, and Escape or the pad's right button goes back. Choosing a character writes the global `character`, and the Rogue's button stays disabled until `king_beaten` is true. In the level, the player's

Character parallel state reads `character` and, for the Rogue, uses five **Modify Stat** actions, which make it faster and give it a quicker trigger, lighter shots and a heart fewer.

- **Options and keys** work as they do in the shooter (8.4). `menus/options.tscn` is one scene used on the title screen and in the pause menu, opened by `OPEN_OPTIONS` sent to the group `options`, with its settings in global variables saved by **Save Game**. Its Screen Shake slider stores into the global `screen_shake`, which every **Camera Shake** in the game is scaled by. The Keys page is ten copies of `menus/key_row.tscn`, each given its `control` under Starting Values.
- **Sound and practice rooms**. Anything that makes a noise as it disappears, such as a coin, a dying enemy or a bomb, uses **Play Sound At Position**, which outlives the node that played it. Bullet Pattern carries its own shot sound, and music changes between the title, the floor and the boss with **Play Music** and its fade. Everything goes through the Effects or Music bus, so the Options sliders reach all of it. Three small scenes in `rooms/` are there to try things in. `test_room.tscn` is for the player alone, `pattern_gallery.tscn` for a turret that works through every kind of bullet pattern in turn, and `enemy_arena.tscn` for seven waves of enemies.

8.6 How to make a rally game (Handbrake Rally)

The third game, in `addons/nodeworthy/games/handbrake_rally/`, is a rally game against the clock, and like the others it has no scripts. Open `title.tscn` and press **Run Current Scene** to play it from its title screen, or run a stage in `stages/` to go straight to the start line. There are three stages, SS1 on gravel, SS2 on tarmac and SS3 on snow, each with a co-driver calling the corners, five rivals to beat and a best time that is kept. Drive them one at a time, or all three as a championship with the times added up. W A S D or the arrow keys drive, Space is the handbrake, E puts the car back on the road, R starts a single stage again and Escape pauses. On a gamepad the triggers, the left stick, the bottom button, the top button, Select and Start do the same.

- **The car**, `car/rally_car.tscn`, is a `CharacterBody3D` with a `Body` model, four wheel models under a `Wheels` node, a `Dust` particle effect and five sound players, which are `Engine`, one road noise each for gravel, tarmac and snow, and `Skid`. On a stage it is in the group `car` and runs the **Rally Car** template. **On The Line** holds it still with the chase camera behind it until the clock sends `Go`, which leads to **Driving**.
- **Driving** is one **Drive** action, set to Acceleration 22, Turn Speed 110, **Turn At Top Speed** 0.45, so a fast corner has to be braked for, **Handbrake Input** Space, and its Top Speed and Grip read from the variables `top_now` and `grip`. Grip is how firmly the car's travel follows its nose. Pulling the handbrake drops it, the tail steps out and the car slides until you let go. **Body Model** points at the `Body` with **Lean Degrees** 6, so the shell rolls to the outside of a corner while the wheels stay flat, and `Drive` turns the wheel models it finds by their names. **Camera Follow** keeps the chase camera behind it with **Behind It** ticked, so the view turns with the car through the corners.

- **The surface** is a parallel state. A **Raycast** looks two metres straight down for the road, which is on a physics layer of its own, and sends OnRoad or OffRoad. **On The Road** and **On The Verge** each copy two numbers into `grip` and `top`. On the road they are its grip and top speed, and on the verge they are its lower ones. Those four numbers are variables too, and each stage gives its car its own under **Starting Values**. Tarmac grips hard and is fast, and snow barely grips at all. The same goes for the colour of the dust and the tyre marks, and for which road noise is heard.
- **Dust, sound and the speedometer** are worked out every frame in Driving with **Math**. The dust's **Amount Ratio** is the speed times `dust_per_speed`, kept between 0 and 1, and its colour is set from `dust_colour`. The engine's pitch rises with the speed and drops back each time the speed passes another 9 metres a second, which sounds like changing up through the gears, and **Set Pitch** plays it. Each road noise's volume is the speed as a share of the top speed times `gravel_mix`, `tarmac_mix` or `snow_mix`, so only the surface you are on is heard. The speed times 3.6, rounded, is written on the Speed label as km/h.
- **The tyres** are a parallel state that reads `slide`, which Drive stores each frame. Above 0.22 the car is **Sliding**. Two **Leave Marks** actions lay tyre marks under the two back wheels, their colour read from `mark_colour`, and **Play Sound** plays the Skid player's own looping sound. Below 0.12 it is **Gripping** again and the skid fades out. The two numbers are apart on purpose, so a car on the edge of a slide does not flick between the two.
- **Damage** is the third parallel state. Drive sends Crashed when a hit takes off at least a quarter of the top speed at once, and stores how hard it hit in `hit`. **Dented** takes a share of that off `condition`, kept between 0.6 and 1, plays the crash where it happened with **Play Sound At Position**, and shows `condition` on the Condition bar. Driving multiplies `top` by `condition` into `top_now`, so a battered car is slower. A harder hit is a puncture, so **Punctured** sets `pull` to 0.12, which Drive's **Pull** reads, and the car drags to one side for the rest of the stage. The hardest sends Rolled.
- **Rolling and recovering**. **Upside Down** stops the car, turns the Body and the wheels over with **Tween Property**, sends Penalty carrying 5 to the clock and goes on to **Recovering**. Recovering is also where E leads, for a car stuck in a ditch. **Path Progress** finds the nearest point of the stage's route and the way the road runs there, and the car is put on that point, facing down the road, for another five seconds. A roll therefore costs ten.
- **The gates** Split1, Split2, Split3 and Finish are Area3D nodes under banners, each with a two-state machine. **Armed** has a **Touching** Check for the group `car`, and **Passed** sends Split1, Split2, Split3 or Finish to every machine. Passed has no way out, so a gate counts once. The Start gate is only scenery.
- **The clock** is the **Stage Clock** machine on each stage's root. **Get Ready** loads the best time with **Save Game** and counts down three seconds with a beep for each, then **Running** sends Go to every machine and counts `time` up, with **Format Time** and **Set Text** keeping the Clock label up to date. Inside Running, the **Splits** parallel state writes each split time on the line as its gate is passed, and the **Penalties** parallel state adds whatever Penalty carries to `time` and shows it. Finish leads to **Finished**, which sends StageDone carrying the time,

and a better time than `best` is saved. The stage's name, its save slot, the next stage and the rivals' times are all Starting Values, so the three stages share one template.

- **The co-driver** is the **Co-Driver** machine on the PaceNote label. Its Starting Values hold two lists. `notes` holds the calls ("4 left tightens", "caution, 1 right", where the number is how tight the corner is, from 6, barely a bend, down to 1), and `note_at` holds how far down the road each corner is. **Reading** takes the next of each. **Listening** runs **Path Progress** on the car every frame and waits until the corner is less than 110 metres ahead. **Calling** shows the note and speaks it with **Text To Speech**, at the volume of the global `voice_volume`, and moves on to the next.
- **The results.** The Results panel's machine waits for StageDone. It copies the stage's rivals into a table with **Machine Variable** and adds YOU with your time using **Dictionary. Keys By Value Ascending** puts them in order, and **For Each** writes one row per driver with its place and time. In a championship, the global `championship` table adds each driver's time to their total with Dictionary's **Add**, and the standings are written the same way. Enter goes to the next stage. After the last stage your total is compared with the saved championship record. A single stage offers R to try again, but a championship turns R off, so there are no second tries.
- **The title screen**, `title.tscn`, has the car on a slow **Turntable** behind a **Title Menu** machine. It starts by reading the best time of each stage and the championship record from their saves, one after another, with **For Each. Main** offers Championship, Single Stage, Options and Quit, and Single Stage opens a page with the three stages. Starting sets the global `mode` to `championship` or `single`, which the results and the restart key read. The title music fades out as the stage loads.
- **Pause and options.** Escape or Start on a stage leads the **Pause Menu** to **Paused**, which uses **Pause Game** with the menu itself kept running, and offers Resume, Options and Quit To Menu. `menus/options.tscn` works as it does in the other two games (8.4). It is one scene used on the title screen and in the pause menu, opened by OPEN_OPTIONS sent to the group `options`, with its settings in global variables saved by **Save Game**. Its first page has the co-driver's voice slider. Its Keys page is seven copies of `menus/key_row.tscn`, one for each control.
- **The test pad**, `pad/test_pad.tscn`, puts two cars side by side on a square of gravel with a skid ring and a row of cones. One is a VehicleBody3D riding on four VehicleWheel3D wheels with real suspension, and the other is the arcade car. Both run the same **Drive** action, which works out which kind it has been put on. Tab swaps between them with **Switch Camera**, **Enable Machine** and **Freeze**. It is the quickest place to change a number on Drive and feel what it does.

8.7 How to make an arena shooter (Swarm Break)

The fourth game, in `addons/nodeworthy/games/swarm_break/`, is a top-down arena shooter against swarms of glowing shapes. Open `title.tscn` and press **Run Current Scene** to play it

from its title screen, or run an arena in `arenas/` to go straight in. Fly with W A S D or the left stick, aim with the mouse or the right stick, fire with the left mouse button or the right trigger, or just push the aim stick over, and fire the slow heavy shot with the right button or the left trigger. Escape or Start pauses. Waves come faster and in more kinds the longer you last. Your score is what you killed times the multiplier, and each arena keeps a top ten.

- **The pilot**, `pilot/pilot.tscn`, is a `CharacterBody3D` in the group `player`, with a Ship model, a Muzzle at its nose, a Camera looking down from above and a Reticle. Its **Pilot** template spends the whole run in **Flying**. There, **Get Input Vector** reads the left stick or W A S D into `move`, **Move Character** flies it, and **Camera Follow** keeps the camera above it, leaning a little towards where it aims. Six parallel states do the rest.
- **Aiming** has two states, **Mouse Aim** and **Pad Aim**, and moves to whichever one you touched last. Mouse Aim uses **Mouse Position** to find the spot under the pointer on the floor and turns the ship to it. Pad Aim reads the right stick into `aim` with Get Input Vector, which keeps the last direction when you let go.
- **Gun** and **Heavy Gun**. **Firing** is one **Bullet Pattern** action whose Count, Spread, Speed and Every Seconds read the pilot's variables `shots`, `shot_spread`, `shot_speed` and `fire_delay`, so a power-up changes the gun just by changing a number. Its **Sound** plays for every shot through one shared player, so a stream of shots stays a sound rather than a roar. The heavy shot is a second Bullet Pattern that goes through what it hits, then waits to be ready again.
- **Hurt**. A touch sends HIT. **Checking Hit** asks whether `shielded` is true and, if it is, lets it pass. Otherwise **Hit** shakes the camera, takes the pilot off its physics layer for two seconds with **Set Collision** For Seconds, so nothing can hit it again while it blinks, and takes a life. The last life leads to **Destroyed**, which hides the ship, leaves a burst and sends PILOT_DOWN to every machine. The pilot picks things up on a layer of its own, so gems and power-ups are still collected while it blinks.
- **Power** and **Shield**. A power-up sends POWER carrying its kind. **Powered** reads it with **Get Event Data** and String Compare checks send the pilot to **Rapid**, **Spread** or **Bombing**, or send SHIELD_UP. Rapid is one **Modify Stat** multiplying `fire_delay` by 0.4 with **For Seconds** 8, so it takes itself away when the time is up. Picking up another before then starts the time again. Spread adds to `shots` and `shot_spread` the same way. Bombing is a **Explosion** over the whole arena. The Shield state shows the Bubble, sets `shielded` for six seconds and hides it again.
- **The enemies**, `enemies/`, are five scenes sharing one template, **Swarm Enemy**, each with its own Starting Values for `kind`, `health`, `speed` and `points`. **Alive** has **Keep Apart**, which pushes each enemy clear of the others near it, so a swarm spreads round you instead of piling into one spot. Its **Behaviour** parallel state starts in **Choosing**, where String Compare checks on `kind` pick a behaviour. Chasers **Chase**. Weavers take turns at closing in with Chase and circling round you with **Strafe Around**, changed by a Send Event with a Delay. Dashers stalk, wind up and dash. Gunners **Keep Distance** and fire a volley every

couple of seconds. Splitters chase and burst into three chasers when they die. **Hurt** takes what a hit carried off `health`, flashes and knocks the enemy back. **Touch** sends HIT to the pilot when it touches it.

- **Dying** plays a pop with **Play Sound At Position**, so the sound outlives the enemy, spawns a burst and a gem, sends ENEMY_KILLED carrying its points to every machine, and with a **Chance** check of 8% leads to **Dropping**, which picks one of four power-up scenes from an Array with **Array**'s Pick Random and spawns it. Instantiate takes the scene's file name from a variable here, which is also how the splitter spawns chasers without the template holding the scene that holds it.
- **Gems and power-ups**, `pickups/`. A gem lies still until the pilot is within seven metres, then **Chases** it, and touching it sends GEM to every machine. A power-up spins where it fell for ten seconds, then goes.
- **The score** is the **Score Keeper** machine on the Score label. ENEMY_KILLED leads to **Scored**, which adds one to `chain`, works out the multiplier as one more for every ten in a row, up to nine, with **Math**, and adds the enemy's points times the multiplier to `score`. A gem is worth 25 times the multiplier. **Hot** waits two and a half seconds. If nothing is killed in that time, **Cooling** puts the chain and the multiplier back.
- **The waves** come from the **Wave Director** on each arena's root. Inside **Playing**, **Get Ready** counts the next wave and shows it. Then a **Counting** and a **Sending** state for each kind work out how many to send with Math and send them one at a time with **Instantiate Somewhere Inside** the arena's SpawnArea, never within twelve metres of the pilot, each after a warning ring. **Fighting** counts the `enemy` group every frame with **Group** and, once it is empty, leads to **Wave Clear** and round again.
- **Three arenas, one director**. How many more of each kind a wave brings is a variable, and each arena sets its own under **Starting Values**, with its name and its save slot. **Pillar Ring** has eight pillars round the middle and more weavers and dashers. In **Closing Walls**, a **Closing Walls** machine on the Closer node shows a warning after twenty seconds, then moves the four walls in nine metres with **Tween Property**, pushing you ahead of them, and shrinks the SpawnArea with them, so nothing appears inside a wall. Fourteen seconds later they open again.
- **The end of a run**. PILOT_DOWN leads the director to **Game Over**, which reads the arena's top ten from its slot with **Save Game** and asks **High Score Table**'s **Would Enter** whether the score makes it. If it does, **Naming** shows a text box, and **Event On Signal** keeps its text in `player` as it is typed and sends NAMED when Enter is pressed. **Entering** puts the name and score in with High Score Table's **Enter** and saves the table. **Final** writes the table out with **Write Out** beside the GAME OVER banner. Enter flies again and Escape goes back to the title.
- **The title screen**, `title.tscn`, reads all three top tens as it starts and offers Play, High Scores, Options and Quit. Play opens a page with the three arenas. **Pause and options** work as they do in the other games. The Pause Menu on each arena uses **Pause Game** with the menu kept running, and `menus/options.tscn` has a screen shake slider, which

every **Camera Shake** follows through the global `screen_shake`, the volumes, and six key rows.

- **The pilot pad**, `pad/pilot_pad.tscn`, is a square with dummies to shoot at. Key 1 sets the gun to a stream and 2 to a spread of five that bounces, by changing the pilot's variables. Keys 3 to 7 send in a handful of one kind of enemy through the gates, and 0 clears them away. It is the place to try a change before taking it into an arena.

8.8 How to make a 2D platformer (Lantern Hop)

The fifth game, in `addons/nodeworthy/games/lantern_hop/`, is a platformer in four levels, each built round one ability. Open `menus/title.tscn` and press **Run Current Scene** to play it from its title screen, or run a level in `levels/` to go straight in. The arrow keys or A and D move, Space jumps (hold it to jump higher), Shift dashes the way you hold, any of eight ways, J or Z grabs a wall or a ring, K pulls you towards metal and L pushes you away, and I or X fires the blaster once you have found it. Down crouches, and Down with Space drops through a thin bridge. Escape pauses. On a pad you use the left stick, A to jump, the right bumper to dash, the right trigger to grab, the left bumper and trigger for the magnet, X to fire and Start to pause. Each level keeps its best time, its most coins and whether you found its hidden heart.

- **The hero**, `player/hero.tscn`, is a `CharacterBody2D` in the group `player`, with a `Picture`, `Dust particles`, a `Rope` and a `Beam line`, and a `Blaster` node. Its **Platform Hero (2D)** template spends the level in **Playing**, whose sub states are the things it can do. They include `On The Ground`, `Jumping`, `In The Air`, `Holding A Wall`, `Climbing`, `Sliding Down`, `Dashing`, `Crouching`, `Swinging`, `Flung`, `Magnetised` and a few more. The actions on `Playing` itself run whichever of them it is in. **Get Input Vector** reads `way` and `facing`, **Camera Follow** keeps the camera a little ahead of where it faces, a **Tile** check sends `Died` on spikes, and a colour shows whether the dash is ready.
- **How it feels** is all numbers on the template, such as `jump_height`, `falls_faster`, `hang_at_top`, `coyote_time`, `jump_buffer`, `fastest_fall` and the rest, so tuning it is changing a variable, never a state. Letting go of Space cuts a jump short. **Jumping** and **Landing** squash and stretch the picture with **Punch** and puff dust with **Emit Particles**.
- **The dash**. **Dashing** reads the way you hold, rounds it to one of eight ways in one **Expression**, and **Dashes** `dash_distance` along it, leaving afterimages with **Instantiate**. It spends `dash_ready`, which the ground gives back. A **Dash Crystal** gives it back in the air by setting the hero's `dash_ready` with **Machine Variable**.
- **Climbing**. Holding grab against a wall leads to **Climbing**, where **Stamina** spends `stamina` while you hold on. When it runs out, **Sliding Down** lets you slip. Climbing past the top leads to **Topping Out**, a small hop over the edge. On the ground it comes back.
- **The rope**. A **Grapple Ring** tells the hero it is within reach by setting the hero's `ring` and `ring_at` with **Machine Variable**. An **Expression** works out `can_swing` and a check leads to **Swinging**, where the **Swing** action hangs the hero from the ring like a pendulum. Up and down climb the rope, and left and right pump it higher. Letting go leads to **Flung**, which

keeps the speed. In `moving_ring.tscn` a **Moving Platform** machine carries a ring back and forth.

- **Magnetism.** A **Metal Block** or **Metal Crate** near the hero sets its `metal_at`, and a **Rail Magnet** sets the nearest point along its rail. The hero works out `magnet_force` in one Expression, from whether K or L is held and how far the metal is, and **Apply Force** pushes it. **Magnetised** turns the hero's gravity off with **Set Gravity**, which puts it back when the state ends. The **Magnet Beam** machine on the Beam line draws the pull, blue, or the push, orange.
- **The blaster.** The **Blaster Pickup** sets the Blaster's `carried`. **Firing** rounds the way you hold to one of eight, fires a bolt with **Bullet Pattern**, and kicks the hero the other way with **Apply Force** as a single push, so a shot fired down in the air is an extra jump. Two shots in the air, given back on the ground. Bolts fly through thin bridges, so you can shoot down through the one you stand on.
- **The enemies** are in `things/`. The Crawler and the Spiky share the **Walker** template, each with its own Starting Values. The Bat is a **Flyer**, and the Hopper a **Hopper**. A Walker walks, and turns at a wall or with the **Touching** check's **At A Ledge**. **Meeting** works out from the hero's speed and height whether it was landed on. If it was, **Squashed** sends Bounce to the hero, whose **Bouncing** state bounces it up and holds time still for a moment with **Set Time Scale** and a **Wait** in real time. Otherwise **Biting** sends Died. The Spiky cannot be landed on and takes two shots. The **Spike Ball** rolls about and is metal. Push it with L and it flies, and whatever it hits is gone.
- **The world.** **Room** keeps the camera inside its rectangle while the hero is in it, with **Camera Limits**, and shows its own `tip` in the caption, or its `pad_tip`, the same tip in controller buttons, when **Last Device** says the player last used a controller. A change of device sends an event that writes the caption again. The **Tip Popup** on each level's HUD, `menus/tip_popup.tscn`, watches that caption and pops each new tip up at the bottom of the screen for a few seconds, once, clear of where you are looking, and an **Array** keeps the tips already shown. There are **Checkpoints**, **Springs**, **Crumbling Platforms**, **Moving Platforms**, a **Switch** and a **Door** picked for it, a **Target** you shoot to open a door, **Crates** to push, **Pits**, and the **Exit**, the lantern. There, **Reached** sends LevelDone to every machine, and **Leaving** goes to the level named in its `next_level`, set under Starting Values. The last one leads back to the title screen.
- **The Crate King**, `things/crate_king.tscn`, waits in **Idle** until you come near. **Fighting** rests, then **Choosing** picks a move by how much health it has left. While it is strong it hops at you. Lower down it charges, and if it charges into a wall it is **Stunned**. Near the end it calls two bats, then hops. Two small machines on its children do the touching. **Boss Head** sends Bounce to the hero and takes one off its health when it is landed on, and **Boss Harm** sends Died to the hero if it walks into the rest. Its Hurt Box, from the right-click menu's **Give It**, takes the blaster's bolts. **Beaten** sends Open to itself, which opens the gate.
- **Coins, hearts and best times.** A **Coin** adds one to the global `coins`, and a **Heart** sets `heart_found`. The **Progress** machine on each level's Stats label has the level's number

under Starting Values. **Starting** reads the saved lists with **Save Game**. **Picking Music** plays that level's tune with **Play Music**, and **Playing** counts the time with **Math** per second, writes it with **Format Time** and shows it with **Set Text**. LevelDone leads to **Saving**, which keeps the better time, the heart and the most coins in their lists with **Array**, opens the next level, and saves.

- **The title screen**, `menus/title.tscn`, reads the same save as it starts. **Main** offers Play (Continue once a level is done, to the highest level open), Levels, Settings and Quit, and counts the hearts found. **Level Select** writes each level's button, with its best time and its heart, in one Expression, and greys out the levels still locked with **Set Property** on the button's Disabled. **Settings** moves the Music and Effects buses with **Set Volume** as the sliders move, turns full screen on and off with **Screen Settings**, and keeps all three in a save of their own. Every level has `menus/pause.tscn` in it. Its **Pause Menu** uses **Pause Game** with the menu kept running, and offers Resume, Restart level and Title screen.
- **How it looks and sounds**. Behind each level, on a CanvasLayer of its own, the **Far Hills** machine reads where the camera is and slides the hills more slowly than the ground, round and round, so they seem far away. Each level sets its own `top` and `repeat`. Things that free themselves, a coin or a popped enemy, play their sound with **Play Sound At Position**, so the sound outlives them. The sounds and music in `sound/` were made for the game from plain tones.
- **The labs**, `lab/feel_lab.tscn` and `lab/enemy_lab.tscn`, are small rooms to try the movement and the enemies in, the place to test a change before taking it into a level.

8.9 How to give your game to other people

While you build, your game runs inside the editor. To give it to somebody who has neither Godot nor NodeWorthy, you **export** it, which means Godot packs the game into a program for Windows or an app for a Mac. The finished game carries everything it needs. Whoever plays it installs nothing, and nothing in it says how it was made.

With the NodeWorthy app everything is ready. The parts Godot needs to build each kind of game came in the folder, and every new game starts with its export settings already made for Windows and the Mac.

1. In the editor, open **Project > Export...** and look at the list on the left, which shows **Windows Desktop** and **macOS**. Click the one you want.
2. Press **Export Project...** at the bottom. Make a new, empty folder for it, since some kinds of export write several files, give the file a name, and untick **Export With Debug**, which is only for hunting problems and makes the game bigger and slower.
3. Press **Save**. After a few seconds the folder holds your game.

Your game's icon. A new game starts with a plain placeholder icon, `icon.svg` at the top of the FileSystem dock, and the finished game shows it on the desktop and in the Dock. Put your own

picture in its place with the same name, or pick another in **Project > Project Settings > Application > Config > Icon**.

You can build both on one computer, for example a Windows game from a Mac or a Mac game from Windows. You can only *try* each one on the kind of computer it is for, so ask a friend who has one before you hand it round.

With NodeWorthy as a plugin in your own Godot there are two things to do once. First, fetch Godot's export templates with **Editor > Manage Export Templates...**, then **Download and Install**. Second, in **Project > Export...** press **Add...** and pick the kind of game. On the new entry's **Resources** tab, find **Filters to exclude files/folders from project** and type

```
addons/nodeworthy_editor/*, addons/nodeworthy/*.pdf, addons/nodeworthy/*.html.
```

That leaves out the editor's panels and the two books, which a finished game never uses. For a Mac game, also fill in **Bundle Identifier** with a name of your own in the shape `com.yourname.yourgame`. Then carry on from step 2 above.

What comes out, and how to pass it on.

- **Windows:** a `.exe` file with a `.pck` file beside it. They belong together, so zip the folder and send the zip. An empty game is about 110 MB, nearly all of it Godot itself, so a big game is not much bigger than a small one.
- **Mac:** one `.zip` file with the app inside, about 60 MB. Send it as it is.

Good to know. To keep the download small, remove what your game does not use before you export. A game started from an empty one still carries the games, the recipes and the examples in `addons/nodeworthy/games/`, `addons/nodeworthy/recipes/` and `addons/nodeworthy/examples/`. If you have borrowed nothing from them, add `addons/nodeworthy/games/*`, `addons/nodeworthy/recipes/*`, `addons/nodeworthy/examples/*` to the filter on the Resources tab. If you have borrowed a scene, a picture or a sound from one of them, leave that folder in. A ready-made 3D player (4.3) wears the animated person kept in `addons/nodeworthy/recipes/3d/character/`, so a game with one borrows from the recipes. Leave the recipes in.

The warning your players will see. Windows and the Mac both check whether a program was *signed* by a developer who has registered with Microsoft or Apple, which costs money every year. A game you export is not signed in that way, so the first time somebody opens it their computer warns them. The game is not broken and nothing is wrong with it. This happens to every small game that is handed round by itself. Tell your players what to expect.

- **On Windows** a blue box says *Windows protected your PC*. Press **More info**, then **Run anyway**. It asks only the first time.
- **On a Mac** a box says the app cannot be opened because Apple cannot check it. Press **Done**, open **System Settings > Privacy & Security**, scroll down to the line naming your

game and press **Open Anyway**. It asks only the first time.

The warnings go away for good only when a game is signed. For a Mac that means joining Apple's developer programme and filling in the **Codesign** and **Notarization** parts of the macOS export settings. For Windows it means buying a code signing certificate. Neither is worth doing until you are selling a game. A game sold through a store such as Steam is started by the store's own program, and players there do not see these warnings.

Art you borrow from the games. The art in the games that come with NodeWorthy may go into a game you make and sell. Most of it is by Kenney and Quaternius, given away under CC0, so nothing is owed, though a thank you in your credits is kind. Emberdeep's characters, tiles and chests are by Jana Ochse of 2D!PIXX, under Creative Commons Attribution 4.0, so a game that uses them must say so in its credits: her name, the licence with its link, and any change you made. The exact words are in `CREDITS.txt` in Emberdeep's folder, and every artist is listed in `ART_CREDITS.txt` in the licences folder beside the NodeWorthy app.

9. Recipes, small scenes that each show one thing

The `addons/nodeworthy/recipes/` folder holds small scenes, not levels. Each one shows a single idea with a machine or two, and says on screen what to press and what happens. Open one, press **Run Current Scene** at the top right of the editor, try it, then click its **Machine** in the Scene dock to see how it is built. Everything in them was made with the graph and the tabs, the same way you would make it. The recipes are sorted into `2d`, `3d` and `ui`, and then by topic.

A recipe puts a few actions together to make something work. To see one action on its own, open its example instead. The start of chapter 10 says how.

Borrow from a recipe, never change it where it is. The recipes live inside NodeWorthy's own folder, which an update replaces. To use one in your game, copy the scene into a folder of your own, and use **Make It Mine** for its templates.

Recipe	What it shows
<code>2d/basics/blink_light</code>	A light that switches itself on and off. Two states take turns, each with Set Visible and a Wait , leaving on FINISHED.
<code>2d/basics/traffic_light</code>	Red, green and amber in a loop, each state lighting its lamp, putting the others out and waiting its own time.
<code>3d/basics/hopping_ball</code>	The ball that hops from 8.1, built exactly as written there. It is a RigidBody3D with its mesh and shape under it, plus Apply Force and Wait .
<code>ui/events/button_tells_another_machine</code>	A button's machine sends OPEN_DOOR to a door's machine with Send Event Reach A Node, and the door fades away and back. The event is written down in the Events tab.
<code>2d/events/event_with_a_delay</code>	Press Space and the bell rings two seconds later. It uses Send Event with a Delay, and a transition waiting for RING.
<code>2d/events/event_carrying_data</code>	Both targets run one Target template that sends SCORED carrying its <code>points</code> variable. The big target's Machine starts it at 50 under Starting Values . The score's transition keeps the points in a variable and Math adds them up.

2d/input/hold_to_light	A lamp lit for as long as Space or a gamepad button is held. Event On Input uses Pressed and Released, with a key and a pad button sending the same event.
2d/input/walk_with_keys_or_stick	Walking with the arrow keys, the D-pad or the left stick. Get Input Vector puts the direction in a variable, and Move Character reads it. A gentle push on the stick walks slowly.
3d/input/walk_with_keys_or_stick	The same in 3D, with the character turning to face the way it walks.
2d/input/hold_to_charge	Holding a button fills a bar, and letting go throws a ball as high as it was charged. Store Held Seconds In fills a charge that is Max 1, Set Value shows it, and Apply Force throws the ball.
2d/input/double_tap_to_dash	Tap Left or Right twice quickly, or flick the stick that way twice, and the square dashes that way, because Event On Input Double Tapped leads to one state with a Dash .
ui/input/secret_combo	Up, Up, Down, Down in time slides a door open for a moment, using Input Combo and Move To on a control.
2d/input/mouse_aim_click_drag	A turret that follows the pointer (Look At The Mouse), a lamp switched by clicking it (the <code>pressed</code> event), and a crate you drag about (Drag).
2d/input/trigger_throttle	A cart that goes faster the further the right trigger is pulled, or at full speed on Up. One Get Action Strength reads "Key: Up or Pad: Right Trigger", and Move Character reads the speed.
2d/input/rumble_and_shake	Space or a pad button shakes the camera and makes the pad rumble (Camera Shake and Vibrate). The words sit on a CanvasLayer so they stay still.
ui/input/which_controller	A card saying whether a key, the mouse or a gamepad was used last, what make of pad it is, and how many are plugged in, built from Last Device , Joypad Connected and Set Text .

2d/input/tap_and_swipe	A tap on the lamp switches it, and a swipe left or right slides the box. Both come from Touch Gesture . The mouse stands in for a finger on a computer.
2d/input/touch_stick_and_pinch	An on-screen stick on the left half of the screen walks a square, and two fingers or the mouse wheel zoom the camera, with Virtual Joystick and Pinch .
ui/input/rebind_a_key	The lamp answers to a control named Light, which starts on Space. Rebind Input moves Light onto any other key the player presses, and every action naming Light follows. Escape keeps the old key, and the choice is kept for the next time the game runs.
3d/input/click_to_mark_wheel_to_zoom	A click on the floor plants a flag on that spot (Mouse Position and Move To reading its variable), and the mouse wheel zooms the camera in and out (Camera Zoom).
2d/variables/health_counts_down	Standing on the spikes takes one health a second. Their Hit Box takes it off the <code>health</code> the player's Hurt Box looks after, the player blinks, safe for a moment, and at 0 is sent DIED. <code>health</code> is Min 0 and Max <code>max_health</code> , and a Stat Panel shows it with nothing in the template writing it.
2d/variables/coin_counter	Five coins share one Coin template. In it, Event On Collision notices the player, Send Event tells the player COIN and Queue Free removes the coin. The player's Adding One state adds one to <code>coins</code> with Set Variable (Add), and a Stat Panel shows it by itself.
2d/variables/loot_drop_by_chance	Clicking the crate breaks it, and Random Event with weights 1 and 2 sends GEM one time in three and EMPTY otherwise. Two variables count the crates broken and the gems found.
2d/variables/health_and_mana_that_start_full	The mage's <code>health</code> has a Max of 100 and Starts Full. Its <code>mana</code> has a Max of 50 and Starts At A Share , 40%, so they begin at 100 and 20. Space casts for 10

	mana, refused below 10 by a Compare , and H takes 15 health, which never goes below 0. Two Bars follow them, full at their most.
3d/detection/guard_that_sees_you	A guard with a Perception node cannot see you behind the pillar. Step out and a Can See check sends SPOTTED, keeping who it saw in a variable that Look At reads. Hide again and it sends LOST.
2d/detection/proximity_alarm	A Distance check sends NEAR when the player is within 300 pixels. The Alarming state's Set Visible blinks the lamp with Blink Every until the check sends FAR, and leaving the state puts the lamp back as it was.
3d/detection/pressure_plate	A Touching check on the plate sends OPEN to the door while someone stands on it and CLOSE when they step off. The door moves up and down with Move To , and either event turns it round halfway.
2d/ai/patrol_between_points	A guard walks round three markers with Patrol , resting a moment at each. The markers are empty Marker2D nodes placed in the level, listed in the order it walks them.
3d/ai/chaser_that_gives_up	A chaser with a Perception node waits at home. Step out from behind the pillar and Can See sends SPOTTED and it follows you with Chase . Stay hidden for two seconds and Target Lost sends GAVE_UP, so it walks back to its Home marker with Move Character , ready to chase again.
3d/ai/pathfinding_round_walls	Click the floor and the walker plants a flag there, then finds its own way through a small maze with Navigate To . The floor and walls sit under a NavigationRegion3D whose mesh was baked once in the editor, and the walker has a NavigationAgent3D under it.
2d/structure/sub_state_unlocks_the_next	A door with two locks. Locked holds three sub states that take turns. First Lock waits for 1, Second Lock for 2, and Both Undone is marked as an End State .

	Reaching it finishes Locked, and a transition on FINISHED opens the door. Pressing 2 first does nothing, because only the sub state that is running listens.
2d/structure/walk_and_blink_at_once	A robot's Awake state holds two parallel states. Walking follows the arrow keys while Blinking flashes its lamp, and neither waits for the other.
2d/structure/rooms_and_a_minimap	A floor of rooms and a map of it. Build Dungeon lays out a start room and five more from three room scenes, new each time, and puts the player in the first. Each room has four copies of one small door scene, whose machine reads <code>connected</code>, which Build Dungeon hands it, and becomes an opening or stays wall. The map in the corner is a Minimap node, added from Nodes To Add, with nothing set but its Player. R builds another floor.
2d/presentation/sound_that_outlives_its_object	Click a balloon to pop it. Play Sound At Position leaves the pop playing where the balloon was, so Queue Free can remove the balloon straight away and the pop still plays to its end. The three balloons share one Balloon template.
2d/presentation/animation_per_state	A hero whose Playing state walks it and turns its picture with Flip Sprite. Inside it, the sub states Standing, Walking and Jumping each play their own animation with Play Sprite Animation. A Compare check on how far the stick or keys are pushed moves between standing and walking, and a Touching check (On The Floor) lands the jump.
2d/presentation/particles_on_hit	Click the target and its Hit state moves the Sparks to the spot clicked with Move To, starts them with Emit Particles, and shows a white flash for a moment before going back to Waiting.
ui/presentation/mute_toggle	An Options panel with a Sound tick box. Save Game loads the choice as the panel opens. Sound On and Sound Off each Mute or unmute the Master bus, tick the

	box to match, and save the choice again, so the next run starts the way it was left.
ui/presentation/hearts_for_health	Health as a row of hearts. Space takes one off and H puts one back with Set Variable , and <code>health</code> is Min 0 and Max 6 so it stops by itself. One Set Property each tick writes it into the Value of an Icon Meter node, which draws whole, half and empty hearts from three pictures.
ui/presentation/your_own_buttons	A pause menu whose panel and buttons are pictures of the recipe's own, stretched with their corners kept. <code>my_look.tres</code> in the folder is a Game Look using them, and the theme made from it is on the Control holding the menu, so only the menu looks like this. The first button has the keyboard, and the arrows, Enter and a click all work.
ui/presentation/collection_page	A collection page. Four items are files in an <code>items</code> folder. Three buttons each hand one over with Give Item , and an Item Collection node pointed at the folder shows the ones ever found in colour and the rest as dark shapes. Hover over one to read it. What has been found is kept, so it is still there the next time the scene runs.
ui/input/menu_with_keys_and_pad	A main menu played with the arrow keys or the D-pad. Each page's state shows its buttons and uses Grab Focus to choose the first one. Event On Signal hears which button was pressed, and one Event On Input hears Escape or the B button to go back. Quit asks first, with No already chosen.
2d/variables/save_and_load	Space adds a coin to a whole number variable. S keeps it with Save Game and L puts it back, even after quitting. Loading with nothing saved sends an event that says so.
2d/flow/change_level_with_a_fade	Two small levels. Each door's Change Scene fades to black, loads the other level and stands the player at a spot by the door, named in Arrive At . The player's template has Carries Over Between

	Scenes ticked, so a level left behind is remembered and its gem stays taken. The score is a global variable. Both doors run one Door template, and each door's Machine starts <code>leads_to</code> and <code>arrive_at</code> at values of its own, set in the Variables tab.
<code>2d/flow/pause_menu</code>	Escape or Start runs Pause Game with the menu under Keep Running, so the ball stops but the Resume and Restart buttons still work. Resume, Escape or Start carries on, and Restart uses Change Scene Reload Current Scene.
<code>2d/dialogue/answer_buttons</code>	A question on a sign, with no dialogue. Show Choices puts a button in the sign for each answer, and each answer sends its own event. The right one opens the gate, and a wrong one says so and asks again.
<code>2d/platformer/moving_platform</code>	A platform on an AnimatableBody2D whose two states glide it back and forth with Move To. Stand on it and it carries you across the gap. The player walks in a Playing state, with Ready and Jumping inside it.
<code>2d/platformer/checkpoint_and_respawn</code>	Event On Collision on the player's feet keeps the last flag touched in a <code>checkpoint</code> variable, and the flags share one template that turns them green. The spikes' Hit Box takes a life from the <code>lives</code> the player's Hurt Box looks after. On HURT, Move To reading <code>checkpoint</code> puts you back at the flag, and on DIED, with no lives left, the game is over. A Stat Panel shows the lives.
<code>2d/platformer/stomp_an_enemy</code>	Two beetles share one Beetle template. Walking has three Touching checks. Against a wall, or At A Ledge, it turns round, so the one on the shelf never walks off. Touching the player, it keeps who it was in <code>hero</code>. Meeting asks in one Expression whether the player was falling onto it from above. If so, Squashed sends Bounce to the player, flattens with Scale To and goes with Queue Free. If

	not, Biting sends Bitten. The player's Bouncing state throws it up with Set Velocity , and Bitten puts it back where it started.
2d/platformer/swing_on_a_rope	The player has a small Reach area. In Running , a Touching check keeps the ring within reach in <code>ring</code> , an Input check keeps whether J is held in <code>grab</code> , and an Expression set to run every frame joins them into <code>can_swing</code> , which leads to Swinging . There the Swing action hangs the player from the ring, pumped by the way held and drawing the rope on a Line2D. Letting go of J leads to Flying , which keeps the speed. Falling Off puts the player back at the start and waits a moment, so the pit it has just left is not still counted.
2d/camera/camera_that_follows	A level four screens wide. Camera Follow glides the camera after the player, and Camera Limits keeps it inside a ReferenceRect covering the level, so it stops at either end.
3d/carrying/pick_up_and_throw	E or X reaches for the crate. A Touching check on a Reach area keeps the crate in a variable, Carry picks it up at a Hand marker, and E or X again throws it the way the player faces.
ui/variables/countdown_timer	Space starts a minute and a half. One Timer action counts <code>seconds_left</code> down and sends TIME_UP when it reaches nothing, and Set Text shows it with <code>{seconds_left:mm:ss left}</code> .
3d/character/watch_a_character_run	There is nothing to press. The runner heads for a marker at one end with Move Character set to Toward A Node , which sends an event when it arrives, turns round and heads back. Play Animation runs its clip and Camera Follow keeps it in shot.
3d/character/move_a_character	The arrows or the stick walk the same character left and right. Its Move Character was given the Tight Feel , so it eases in and out of a walk instead of starting and stopping dead. Standing and

	Walking each play their own clip, and the walls at each end keep it on screen.
3d/character/third_person_over_the_shoulder	Walk, run, crouch and jump under a camera set Over The Shoulder, each clip's Speed read from a variable so the feet keep in step with the ground. Crouch makes the body shorter and will not stand up under a ceiling, and Esc gives the pointer back.
ui/variables/every_kind_of_variable	One variable of every kind on one board, and Space changes each the way that kind changes. An int counts and stops at its Max limit, a float adds a half, a bool flips, a String grows, a Vector2 or Vector3 moves, a Color darkens, an Array gains an item and a Dictionary counts by name, while a Node and a Resource picked in the Variables tab stay as they are.
ui/variables/text_formatting	One Set Text writes a score padded with noughts ({score:0000}), a clock ({time:mm:ss}) counting down by itself with Set Variable Per Second, a speed to one place ({speed:0.0}) and a share as a percentage ({share:%}).
ui/variables/counting_things_by_name	One Dictionary variable holds how many of each kind of enemy are left in a wave. S, B and G each send BEATEN carrying the kind, the transition puts it in <code>kind</code>, and Dictionary Add takes one off it, crossing a kind off when it reaches nothing with Remove At Nothing, and Count says when the wave is cleared. For things the player carries, use an Inventory.
ui/variables/score_shared_by_every_machine	Three buttons from one template all add to one Global <code>score</code> while each keeps its own count of presses in a variable that is not Global. The scoreboard's own machine shows the score, says when it reaches 20, and R sets it back to 0 for every machine.

2d/variables/save_the_whole_scene	Pick up coins, save with S, take more, then load with L. Save Game's Save The Scene and Load The Scene put back where the player stood, the coins taken since the save, and the count.
2d/items/wear_what_you_find	A pack and a set of worn slots, each in its own Inventory Grid . Tab (one Event On Input) swaps the keyboard between them. Choosing a thing sends WEAR or TAKE_OFF, and Equip Item or Unequip Item wears it or puts it back, while a Stat Panel shows Damage and Armour follow by themselves.
2d/items/gear_changes_your_stats	A hero, a pack with a sword and a cap, and a Stat Panel . S wears the sword, C the cap, R takes everything off. The panel reads the hero's stats by itself, so a worn sword shows as Strength 15 (10 + 5), and taking it off takes the 5 away again. Nothing in the template writes the numbers, because Equip Item and Unequip Item make and undo each item's own stat changes.
2d/story/talk_to_someone	Walk up to the guard and press E and his dialogue plays in a Dialogue Box , with two answers that each lead to their own reply, in a voice of his own. The hero is held still while he talks. Double-click <code>talk_to_the_guard.tres</code> to see it as a graph.
2d/story/a_choice_that_opens_the_gate	The answer <i>Please open it</i> sends OPEN_GATE, set with the ☞ beside it. The gatekeeper's template goes to Opening , which slides the gate away. <i>No, thank you</i> leaves it shut.
2d/story/a_quest_from_a_conversation	<i>I'll find them</i> starts a quest shown in a Quest List , and walking onto three coins ticks the first step off. <i>Here are your coins</i> , offered only while the quest is under way, leads to a state where Remove Item takes the three coins, which finishes the quest and pays 50 gold, and she thanks you. With fewer than three she says so and takes nothing.

2d/story/a_quest_with_a_map_and_a_log	A Top-Down Hero and a Villager from the right-click menu, and no state added by hand. The villager's Dialogue and Gives Quest settings give <i>Light The Beacons</i> when the talk ends. Its two steps are each The Player Reaches A Place . The Quest List shows it, the Minimap marks the beacons still to light, and Q opens the Quest Log .
2d/story/someone_worth_talking_to	The farmer's lines are said by an Actor , <code>old_farmer.tres</code> . It keeps a name, a colour, a speed and a talking sound in one place. The greeting's answers are Picked At Random , and <i>Any news?</i> is offered Only Once . The gate's template asks Dialogue Line Said every frame, opening once the news has been said.
2d/combat/a_sword_that_hits_a_slime	J or a click swings with Attack. 1 takes the wooden sword and 2 the iron one. A swing does the sword's damage plus the hero's strength to a slime's Hurt Box , so the iron sword beats one in two swings, the wooden in four.
2d/combat/an_enemy_that_hurts_you	A slime comes for the hero, and its Hit Box takes 3 health a touch, with a knockback and a moment of safety. At 0 the hero falls, and R gets them up.
2d/combat/beat_slimes_level_up	Each slime is worth 40 XP. The third punched takes the hero to level 2 through the Experience node, then strength and health grow and <i>Level up!</i> shows. The Stat Panel shows the XP out of what the next level needs.
2d/items/a_potion_that_wears_off	A potion made once as an item file. Its What It Is is Used Up , with Strength + 10 under When Used for 5 seconds, and one Use Item drinks it from the inventory, wearing off by itself, or sends <code>NONE_LEFT</code> when there are none. H heals 8 and D takes 7, and Health's Max and Min keep it between 0 and <code>max_health</code> .
2d/items/a_shop_and_your_coins	A shelf, an inventory of twenty coins and prices. The arrows look along the shelf

	and Enter buys. Show Item puts the price in a variable, and Remove Item takes that many coins, or, when the purse is short, takes none and sends TOO_DEAR, so Move Item only hands the goods over once they are paid for.
2d/items/open_a_chest	A chest and a pack side by side, each drawn by a Inventory Grid. The arrows move the highlight along the chest and Enter takes that whole stack across. The chest's panel sends the item it chose, an Event On Signal puts it in a variable, and Move Item carries it over. Grab Focus gives the chest's panel the keyboard as the scene opens, so the arrows work without a click first.
2d/items/drop_it_and_pick_it_up	WASD or the left stick walks a character round a room. Tab, or the gamepad's top button (Y / Triangle), opens the pack. In Choosing the character stands still and Grab Focus gives the pack's Inventory Grid the keyboard, so the arrows or the stick move the highlight and Enter or the bottom button (A / Cross) puts that stack on the floor beside them. Tab or Y again, or Escape or B, closes it, and Walking takes the keyboard back with a Grab Focus set to Release Instead, so walking never moves the highlight. The dropped stack is an area with an Inventory of its own and no template at all. Walking into it fires Event On Collision and one Move Item (with no item named, so it moves the whole inventory) empties it back into the pack before Queue Free clears the empty stack away.
2d/maths/spinning_square	A square that spins at a steady speed on any computer. One Rotate turns it 90 degrees a second, and Get Property reads its angle for the label.
2d/maths/speed_that_builds	Hold a button and a fan winds up to top speed, let go and it winds down. It uses Math Move Toward, Per Second, with Rotate reading the speed.

2d/maths/light_that_pulses	A glow that breathes. Math turns a growing angle into a sine wave, and two more Maths make it into a glow that sets how solid and how big the light is, through Set Property on <code>color:a</code> and on <code>scale</code> , where one number sets it both ways.
2d/maths/move_only_the_y	A lift that follows the mouse up and down only. Get Vector Components takes out the Y, Min and Max keep it in the shaft, and Set Property changes just <code>position:y</code> .
2d/maths/random_point_in_a_rectangle	Space sends a star to a random place in a box, using two Random Numbers , Set Vector Components to put them together, and Move To .
2d/maths/turret_that_turns_the_short_way	A turret turns toward the mouse at 90 degrees a second, always the short way round, with one Look At set to The Mouse and a Turn Speed.
2d/maths/hovercraft_that_drifts	Steer a hovercraft that takes time to change course. Move Character's Acceleration makes its speed build up and die away, and Vector Length shows how fast it goes.
2d/maths/ease_a_colour_and_a_place	Space sends a card across and changes its colour. Ease Variable eases a Vector with the Back curve, which goes past and settles, and a Color with Sine, and Set Property puts both on the card.
ui/maths/calculate_damage	One Expression works out a hit from strength, armour and a dice roll, doubling it on 18 or more, and Set Variable's Subtract takes it off health. Health and armour stay in range by their own Min and Max limits, and a real game gets the same sum from a Hit Box and a Hurt Box's Defence Variable.
ui/branching/switch_by_weapon	Space cycles a weapon number, and Switch with Is Equal To sends the event for fists, sword or bow, and Math Wrap brings the number back round to 0.

ui/branching/switch_by_health	One machine changes a Global health , kept between 0 and 100 by its Min and Max. Another uses Switch with Is Below, 25 and 50, to show Critical, Hurt or Fine, changing only when the range changes.
2d/basics/spinning_pickups	Gems turning with Rotate . Walk over one and it spins fast, shrinks with Tween Property and is destroyed. The three gems are one gem duplicated, sharing one template.
3d/basics/turn_tip_and_roll	Space swings a door with Rotate To's Turn and T nods the camera down with its Tip, while Rotate turns a gem about its up axis and rolls a wheel.
3d/landscape/glide_from_a_hilltop	A glider made of actions. The Third Person Hero stands on a peak of a Terrain , and a node called Glider under it has a Machine of its own. Pressed again in the air and held, the Space bar makes the fall light with Set Gravity until it is let go, and the hero comes down gently a long way off (6.21).
3d/landscape/swim_and_hold_your_breath	Swimming and holding your breath, made of actions. The hero walks into a Terrain's lake, and a node called Swimmer under it has a Machine of its own. Waist deep, Swim takes over, Space swims up and C dives. With the head under, In Water notices and an Air bar runs down, it comes back above the water, and with none left the hero is put back on the bank (6.21).
2d/flow/for_each_over_a_hundred	Repeat spawns a hundred enemies, each somewhere inside an arena's rectangle by Instantiate's Somewhere Inside. Space makes a list of the group with Group List , and For Each hands them out one at a time to a state that sends each one FREEZE.
2d/physics/knockback_away_from_the_player	Space knocks every nearby crate away from the player with one Explosion action (Damage 0). The closer a crate is, the harder it flies, and one out of reach stays put.

2d/physics/prize_wheel

Space spins a wheel with **Apply Force** Torque. A **Property** check waits for it to stop, then **Math** (Wrap, Divide, Floor) finds the quarter under the pointer and **Array** reads that color from an Array.

10. The actions and checks that ship

Every action has the three settings in 6.3 plus its own. The actions are grouped as the Add Action window groups them, from the state machine itself to the game systems. Where a setting says "or from a variable", you can type a number in or name a variable to read it from while the game runs.

Every action has an example. `addons/nodeworthy/examples/` holds one small scene for every action and every check, sorted into the same groups as the Add Action window. For example, the example of Wait is `examples/time/wait/wait.tscn`. Each one is bare. It holds the action, the fewest things it needs to be seen, and a line on screen saying what to press and what to watch. The quickest way there is **Open Example** in the Add Action window. Pick an action and press it, and its example opens ready to run with **Run Current Scene** (F6). Click the example's Machine to see the template, which holds the action and not much else. Like the recipes, copy an example into your own folder before changing it.

This chapter is the short tour. The **NodeWorthy Action Reference**, the second book that comes with NodeWorthy, gives every action and check its own page. Each page has a picture of the action as it appears when you add it, what it is for, and a line on every one of its settings. It also has an "I want to..." list, for looking up what you are trying to do when you do not know what it is called.

The groups below are the ones the Add Action window shows, in the same order. A check sits in the group of what it asks about and is marked *check*. It asks its question and sends one event for yes and another for no (6.6).

10.1 State Machine, for events, states and other machines

Action	Does
Cancel Delayed Events	Call off events a Send Event with a Delay is still waiting to send, while the state that sent them is still running, such as a fuse cut before BOOM or a guard who sees you again before GIVE_UP. All or One Event by name. The Send Event called off counts as finished, so its state can still finish.
Enable Machine	Start, stop, restart, pause or resume another machine. Stopping leaves its states properly and begins from the top next time. Pausing holds it where it is, timers and all, and resuming carries on, so a patrol paused for a cutscene picks up where it stopped. Resuming also puts back what its every-tick actions show, such as a Set Text, in case something else wrote over the label meanwhile. This means four weapons can share one ammunition counter, and the one taken out shows its own count at once.
Forward Events	While its state runs, pass the events this machine gets on to another Machine . For example, a player sitting in a car hands HIT to the car. Every Event , Only These or All But These . With

	Set As Handled this machine does not act on them itself. FINISHED is never passed on.
Get Event Data	Read the value an event arrived with into a variable, such as who touched the door or how much damage was dealt. Send Event can carry anything, and this is one way the state that catches it gets hold of it. Put it in the state the transition leads to. For an event written down with what it carries, the transition itself can store the value instead. Store Sender In keeps who sent the event as well. That is the node whose machine sent it with Send Event, or whose Explosion or Alert Allies it was, such as the soldier whose shot hit the player.
Get Last Event	Put the name of the event that led to this state into a String variable, for a Hurt state that sounds different for HIT_BY_ARROW and HIT_BY_SWORD. It gives FINISHED when the state before finished, and is empty in the first state or after Go To State.
Get Previous State	Put the name of the state the machine came from into a String variable. Leaving a state with states inside it gives the innermost one that was running. The first state gives an empty name.
Get State	Put the name of the state a machine is in into a variable, once or every frame. A door can ask what the switch is doing, or a guard's state shown over its head while you test. Leave Machine empty for this machine. A state inside another gives the inner one, and a machine that is not running gives nothing.
Go To Previous State	Go back to the state the machine came from, with no transition naming it. Think of a pause menu going back to whatever was running, or a Back button reachable from several screens.
Go To State	Jump straight to another state without waiting for a transition.
In State <i>check</i>	A machine is in a state, picked from that machine's states, such as a door that opens only while the switch is in On. A state counts while a state inside it runs. Leave Machine empty to ask about this machine.
Machine Variable	Read a variable from another machine, or write one to it, such as a door asking the key holder or a boss telling its minions. Add adds an Amount to their number. With an Amount of -1, a bullet takes 1 off the health of the enemy it hit.
Next Frame Event	Send an event on the next frame, to let everything settle, pass straight through a state, or break a circle of states that lead straight on to one another. Sent as FINISHED, it completes only the state it is in, never the state that one sits inside.
Random Event	Send one of several events at random, with optional weights. A weight missing from the end of the list counts as 1, so adding an event and forgetting its weight does not throw the others away.
Run Template	Run another template inside this state, on the same node. Build a behaviour such as “take a hit” once and use it from many templates. The state carries on when the other template reaches a state marked as an end state, or sends Done , and leaving early stops it. Shared Variables are handed to it as it starts and taken back when it ends.
Send Event	Send a named event to this machine, to every machine in a group, or to every machine in the game. Can carry a value, typed in or read from a variable. With a Delay it waits that many seconds first. Its state does not send FINISHED until the event has gone, and leaving the

	state before then means it is never sent. An event the template declares global always goes everywhere, whatever this says. Set Reach to A Node to tell one particular machine, such as the enemy a bullet just hit, and press Var on To Node to use the node in a variable. A machine on a node that the one picked sits inside hears it too, so hitting an enemy's hurtbox reaches the enemy.
Sub State Finished <i>check</i>	A state nested in this one, marked as an end state, has been reached and finished.

10.2 Logic, for true and false, choices, and doing something more than once

A state machine loops by going round a ring of states. These three actions decide when to go round again and when to stop, and the ring stays two states.

The shape is always the same. Put the loop action in one state, point its Next or Again event at the state that does the work, and point that state's **FINISHED** back at the first. Done leads out of the ring. Each of them keeps its count in a variable you can see in the Variables tab, so you can watch the loop go round and reset it yourself if you want to.

A question in NodeWorthy is asked by a Check action in the state, and it reads in the Actions tab as a sentence, such as "fuel < 20 → EMPTY", so you rarely need to work anything out by hand. Bool Operator and Check are for the times you want to work an answer out and *keep* it.

Action	Does
Bool Operator	True-or-false sums into a variable, such as both of these, either of these, not this, or one but not both. Work "has the key AND the door is unlocked" out once into <code>can_open</code> , and every door just asks <code>can_open</code> .
Chance <i>check</i>	A roll of the dice, once per visit or every time it is asked.
Check	What every question marked Check in Add Action becomes (see 6.6). It is not added on its own. It asks the question, stores the answer as true or false when you name a variable, and sends one event when it is true and another when it is false. On Every Frame it keeps asking. It asks every frame, or, for a question about variables alone with Only When It Changes ticked, whenever one of them is written. It sends only when the answer changes unless you untick Only When It Changes. Told to wait for the other actions on Once, it asks once they have finished. Ask The Opposite turns the question round ("is not touching the ground"), which also turns round the answer kept in Store In, something swapping the two events cannot do.
Compare <i>check</i>	Two numbers compare as you say, either of them from a variable, for example health below 30 or score at least 10.
For Each	Hands out the next item of a list each time the state is reached, sending Next while items remain and Done when they run out. It suits every enemy in the room, every spawn point, every line of dialogue.

Repeat	Sends Again until the state has been reached a set number of times, then Done. Fire three shots, or spawn five coins.
Sequence	Sends a different event each visit, working down a list in order and optionally wrapping round, like a boss with three attacks in turn.
Switch	Send one of several events by what a variable holds, in place of a Check for every case. Is Equal To matches numbers or words, such as weapon 0, 1, 2, or north and south. Is Below matches ranges, listed lowest first. Health below 25 sends Critical, below 50 Hurt. Otherwise is sent when nothing matches. On Every Frame it speaks only when the case changes. Cases without events and ranges out of order are on the Problems button.
Variable check	A variable equals, differs from, is greater or less than a value, or is true or false.
Variable Changed check	A variable's value changed since the last check.

10.3 Math, for numbers and their sums, formulas, limits and chance

Action	Does
Clamp	Keep a variable inside a range and send an event when it hits either end. Think of health that cannot go below zero, or a capped score. A variable nothing has written yet is left alone, so it cannot report hitting the bottom before anything has set it. On Every Frame it looks again whenever its variable is written and costs nothing in between.
Ease Along Curve	Move a number, a Vector or a colour variable toward a value over time along a Curve you draw, where Ease Variable offers ready-made shapes. It suits a jump that hangs at the top or a light that flares and settles. At a height of 1 it has arrived, above 1 it goes past, and a Curve that ends at 0 brings it back to where it began. Finishes when the time is up.
Ease Variable	Move a number, a Vector or a colour variable toward a value over time instead of jumping to it, such as a fuel gauge settling, a light dimming, a place drifting to another place, or a colour fading to another colour. Its To is the variable's own kind, and its curves are Tween Property's. Finishes on arrival, and a whole number stays whole.
Expression	Work out a formula written in one line, with the machine's variables by name, and keep the answer in a variable. Examples are <code>speed * 2 + boost</code> , <code>clamp(health - damage, 0, 100)</code> , <code>(target - here).normalized() * speed</code> . A name with a space goes in braces, <code>{time left}</code> , as Set Text writes it. <code>choose(health < 30, 2, 1)</code> picks one of two answers (a formula cannot use if and else). <code>delta</code> is the seconds since the last tick, and a Node variable's properties read with a dot, <code>player.position.y</code> . Angles are in degrees through <code>sine()</code> , <code>cosine()</code> , <code>tangent()</code> , <code>asine()</code> , <code>acosine()</code> , <code>atanangent()</code> , <code>angle_of()</code> and <code>turn_between()</code> . Godot's own functions such as <code>clamp</code> , <code>lerp</code> , <code>remap</code> , <code>smoothstep</code> , <code>snapped</code> , <code>sqrt</code> and <code>pow</code> work too. The answer keeps its variable's kind. A formula that cannot be read is on the Problems button. The Action Reference lists everything a formula can use. Everything it does can be done one step at a time with Math, Vector and Set Variable.

Math	Add, subtract, multiply, divide, what is left over from a division, the smaller or larger of two, a power, a square root, absolute, round, floor, ceiling or sign, into a variable. Lerp goes part way from one number to another and Move Toward steps toward it without going past. For angles, all in degrees, there are Wrap (370 in 360 is 10), Angle Difference (the shortest turn from one angle to another), Sine, Cosine, Tangent, and Angle Of Direction (the angle a direction points). On Every Frame, Add, Subtract, Lerp and Move Toward can be Per Second, so they run the same at any frame rate. Dividing by zero gives 0 and says so in the Output. Either side can come from a variable.
Random Number	Put a random number into a variable, or a whole number, each one from the lowest to the highest as likely as the others.
Random Seed	Decide how chance plays out from here on. A New One picks a fresh seed and stores it. This One uses the seed you give, so every roll, pick and chance after it comes out the same each time, for a shared run or a daily challenge. Store the seed in a variable to show it on screen.
Set Variable	Set, add to, take away from, multiply or toggle one of the machine's variables (in the Add Action list it is under Math). On Every Frame, Add can be Per Second , and then 90 adds ninety a second on any computer, however fast it draws, and an angle, a timer or a speed grows steadily. An int stays an int. A fraction added to one is rounded, and the Output says so. Math does the same. Picking the variable makes the Value beside it the same kind, so a true or false variable gets a tick box. Subtract takes a number, or another variable's value, off it, so "health take away damage" needs no formula. The variable's Min and Max still hold.

10.4 Vector, for Vector2 and Vector3, which are places and directions

Action	Does
Get Vector Components	Take a Vector apart into Numbers, putting its X, its Y and in 3D its Z each into a separate variable. How high the player is, how far across the mouse is, how fast something falls.
Random Vector	Put a random direction, or a random place in a box or a circle around Center or a Center Node , into a Vector variable. It decides which way a wandering animal sets off, where a spilled coin lands, or where the next enemy appears. In 3D, On Ground keeps it across the ground.
Set Vector Components	Change one part of a Vector variable and leave the rest. Set a place's height and keep where it is across, or build a Vector from two or three numbers, typed in or read from variables.
Vector	The same sums for places, directions and speeds. It can add two together, take one from another, make one longer or shorter, multiply, divide, or take the smaller or larger part by part, cut it down to a length of one, keep it no longer than a top speed, turn it by some degrees, go part way to another, turn it round, bounce it off a surface, or measure the length, the distance, the angle (0 to 180 degrees), the signed turn from one to the other (-180 to 180), the direction between two of them, or how much they point the same way. On Every Frame, Lerp, Move Toward and Rotate By can be Per Second. Either side can be a place you type in, a variable, or where a node stands. Type two numbers and it works flat, so the one action suits a 2D game and a 3D one.

10.5 String, for text and building it, formatting it and comparing it

Action	Does
Format String	Put text together into a variable, dropping in any <code>{variable}</code> , with the same ways of showing a number as Set Text, for a score line, a name or a message.
Format Time	Write a number of seconds the way a clock shows it, into a text variable, such as <code>1:03.4</code> for a stage time, <code>0:58.90</code> for a lap, <code>12:05</code> for a round timer, <code>+0.6</code> for a gap to a best. Point it at the variable Timer counts in and Set Text shows the result. It keeps up every frame and cuts the fraction rather than rounding it, so the screen never shows a tenth that has not gone by.
String	Work with text and keep the result in a variable. The choices are Length , Left , Right , Substr , Replace , Split into a list, Join a list, To Upper , To Lower , Strip Edges and Find . A name in capitals, a code typed in without its stray spaces, a line of items turned into a list.
String Compare check	A variable's text equals, differs, contains, starts with or ends with something, or is empty.

10.6 Array, for many values in one variable

Action	Does
Array	Add to a list, put an item in at a place, change or remove the one at a place, read an item, find where one is, count it, take one at random, sort it, Shuffle it, Reverse it, ask whether it holds something, add every item of another list with Append Array , deal one into another list with Move Item , make it a given length with Resize , or empty it. A list is how a machine remembers several things.
Dictionary	Set an entry by its name, add a number to one (one more potion, ten fewer coins), read one, ask whether it is there, remove one, count them, list their names, or empty the table. Keys By Value Ascending and Keys By Value Descending list the keys in order of their values, which is a race's results with the fastest at the top, or a high-score table. A table is how a machine remembers a value for each of several names, such as how many of each item the player carries or the best time on each level. With Remove At Nothing ticked, an entry that Add brings to 0 or less is taken out, so a used-up name is no longer listed or counted.
High Score Table	A top ten, kept in a variable. It holds each run's name and score, best first, with only the best few kept and the same name allowed twice, as on an arcade table. Would Enter asks whether a score makes the table, so the game asks for a name only when it does. Enter puts it in and gives its place, and Write Out turns the table into text for a label. Best Is Smallest keeps times instead. Save Game keeps the table.

10.7 Data, for records and tables of your game's things

Action	Does
--------	------

Get Data	Read one field of a Data record into a variable. The record can be a Data file, one kept in a variable, or the Data node under a node. See 6.20.
Set Data	Change one field of a record, or add to a number field, such as taking this enemy's own Health down by one. A value that does not suit the field is refused and the Output panel says why.
Find Row	Pick a row of a Data Table and keep it in a variable. It picks by name, by place, at random, the first whose field holds a value, or the one with the highest or lowest number. Sends Found Event or Not Found Event.
Data (check)	Ask whether one field of a record is equal, not equal, greater or less, or whether an On/Off field is on.

10.8 Time, for waiting, timers and the speed of time

Action	Does
Get Time	Put a time into a variable. It can be the seconds since the game or the scene started, which carry on through a pause, the seconds this state has run, the time of day on the player's clock (as text, and as the hour, minute and second), or the last frame's delta . Timer is the one for a countdown or a stopwatch that pauses with the game.
Set Time Scale	Speed the game up or slow it down, putting it back when the state is left, for a hit stop or a slow-motion finish. When slow-downs overlap, the game goes back to the one still in force, and to normal speed once the last has ended.
Timeout check	The state has been running for a number of seconds, plus an optional random extra.
Timer	Count a variable down in game time and send an event at zero. It makes a round timer, a fuse or a power-up wearing off. Tick Count Up for a stopwatch instead. Then it counts up from zero, for how long a level has taken, and stops at Seconds with its event, or never stops when Seconds is 0. Show it with <code>{time:mm:ss left}</code> in a Set Text, or <code>{time:mm:ss}</code> for a stopwatch. It follows the game's speed, so slow motion slows it and a paused game holds it.
Wait	Wait a number of seconds, plus an optional random extra, then finish. The wait stands still whenever its machine does (paused, the game paused around it, or switched off with Process Mode set to Disabled on the machine or a node above it) and carries on from where it was. Tick Real Time to count real seconds whatever speed the game runs at. A hit stop sets the speed to 0 with Set Time Scale , and only a wait on real time ends while the game is frozen. The problems button says so if a state freezes the game beside a wait that would never end.

10.9 Node, for finding, making, freeing and changing nodes

Action	Does
Add Node	Add a new node of one of Godot's own kinds while the game runs, such as a Timer for a fuse, an Area2D that notices what walks in, or a light that comes on. Pick the kind from the list, the Parent it goes under (Self when empty), a name if you like, and keep it in a Node variable to set it up next. Instantiate adds a whole scene, but this adds one plain node.

Call Method	Call a method on any node, picked from a list of what that node can do, with up to two arguments, optionally storing the result. Most of what a game does has an action already, and this reaches the rest, such as a node from another plugin.
Call Static Method	Call a method that belongs to a class rather than a node, such as Time's time of day, OS's system name or FileAccess's file check, or a static func of a class_name script you wrote, with up to two values, and keep what it gives back. Call Method is for a node in the scene.
Duplicate	Copy a node already in the game, beside it, with everything under it, and store the copy in a variable. A copy whose machine copies itself as it starts is held to one round a frame, with a warning, rather than stopping the game.
Emit Signal	Make a node send one of its signals, as if it had sent it itself, such as a button's pressed, a timer's timeout, or a signal from your own script. Everything connected to it hears. Argument Count must match what the signal carries.
Enable Node	Switch a node off completely, or on again. Off, it is hidden, not running (its machine, animations and timers hold where they are) and out of the physics world, so things fall and walk through it. Switched on, it is put back exactly as it was. Switching off the node a Machine is on switches that Machine off too.
Event On Signal	Listen to any node's signal while this state runs and send an event when it fires. In Godot this one action covers collisions, areas, buttons, sliders, timers and anything else that emits. Listening never holds the state up, so a state can watch for something while it gets on with something else. A signal that hands over several things stores the one Argument says, where 1 is the first. With Store Argument In set, Event Name can be left empty. A slider that is only kept in a variable then sends nothing.
Find Child	Find a node by its name, in the whole scene or under Self, and store it in a variable for other actions to work on. A * matches any letters, so <code>Enemy*</code> finds the first enemy.
Get Children	The nodes directly under a node, given as a list of all of them, one by its number (counting from 0, with -1 the last), or how many there are. Count the enemies left in a wave, hand a patrol its markers.
Get Parent	Put the node a node sits under into a variable, such as the enemy a hurtbox belongs to or the hand carrying a crate.
Get Property	Read any property of any node, picked from a list, into a variable. It might be a position, a health field on your own script, or whether something is visible.
Group	Put a node in a group or take it out, count how many are in a group, or list them for For Each . Empty Event is sent when a count finds nobody left, when the last enemy is defeated or the last coin collected. Only Under counts or lists only the ones inside one node, such as the enemies in this room. Add Children and Remove Children put every child of a node into the group, or take them all out.
Group Switch	Send one of several events, picked by which group a node is in, for example HIT_ENEMY for an enemy and HIT_WALL for a wall. The first group listed that the node is in wins, and Otherwise is sent when it is in none.
Has Child Of Type	A node has a node of a kind under it, such as an AnimationPlayer, a Hurt Box or an Inventory. Kinds built on the one asked about count. Untick Recursive to look only straight under it.

<i>check</i>	
Instantiate	Add a scene to the game At Self , at another node such as a gun's muzzle, at a place such as where the mouse points, or wherever the scene itself was saved, keeping the scene's own size, optionally storing the new node in a variable. The Offset turns with the node it appears at, so a bullet a little in front of a gun stays in front however the gun turns. It works in the state a machine starts in, too. At One Of The Nodes Under picks one of the nodes under another node at random, for enemies coming in from a set of spawn points. At Somewhere Inside picks a place at random anywhere inside another node's collision shape, a box or a circle drawn over the arena, so enemies can come from anywhere in it. In 3D it appears at the height of the shape's middle. Every Seconds keeps spawning another every so often while the state runs, which is a gun's fire rate (0.25 is four shots a second). Stand On turns the new node so its up points along a direction, such as the way a surface faces from a Raycast's Store Normal In, for a mark on a wall or sparks off it. Press Var on Parent to put it under the node that was hit, and it moves with it. Set a Warning scene, such as a flashing mark, and it shows where the scene will appear for Warning Seconds first, so the player sees an enemy coming. The scene then appears where the warning stood. With At One Of The Nodes Under or At Somewhere Inside , Away From and At Least pick only places far enough from a node such as the player, so nothing appears on top of them. When every place is too close, the farthest is used. Its Scene is picked from your scenes by name and kept as the scene's path, so a scene can spawn copies of itself, as a splitter breaking into smaller ones does. It can be read from a variable, letting each node spawn a different scene.
Is Child Of <i>check</i>	A node is somewhere under the Parent you pick, like the crate on the lift or the sword in the hand. Untick Recursive to count only a node straight under it.
Node <i>check</i>	A node still exists, is in a group, or is visible. Use it to notice a target dying.
Property <i>check</i>	Any property of any node compares as you say.
Queue Free	Remove a node, now or after a delay. An effect scene that removes itself this way never stacks up. Name a Group instead to remove every node in it at once, such as all the pickups and chests left on a floor before the next one is built. The machine doing it is left alone.
Reparent	Move a node under another node, which it then moves with, such as a sword into a hand or a passenger onto a moving platform. Keep Global Transform leaves it where it is in the world. With New Parent empty it goes back under the scene.
Set Process Mode	Set Godot's Process Mode for a node. Always keeps a pause menu or an on-screen clock running while the game is paused, Pausable stops with the pause, When Paused runs only then, Disabled never runs, and Inherit does what the node above does.
Set Property	Set any property on any node, picked from the list of everything that node has. It sets it once, as the state is entered. Set Run Mode to Every Frame when the value should follow a variable that keeps changing, such as a light's Energy worked out by Math each tick, or has to be held against something else changing it, because on Every Frame nothing else can change that property while the state runs. To reach inside a value, put a colon between the names, as in position:y or material:albedo_color. A turn is always listed in degrees. Where Godot keeps a turn in radians, the list offers Rotation Degrees in its place. A single number for a property with parts, such as scale , sets every part to it, so 1.5 makes it 1.5 each way, typed in or read from

a number variable. A colour needs a colour, and a number given to one is refused, and the Output says why.

10.10 Transform, for moving and turning things, paths and distances

Action	Does
Carry	Pick something up and carry it, put it down, or throw it the way the carrier faces. Set Carried By to a marker at the character's hand. While it is carried it moves with the carrier, and a body stops falling and bumping into things. Putting it down puts all that back.
Distance check	Another node, or the nearest member of a group, is within a distance. Can store what it found.
Follow Path	Move along a Path2D or Path3D drawn in the scene. It goes once, round and round, or back and forth, turning to face the way it goes. A moving platform on a curve, a car on a track, a ghost on its round. Going once, the state finishes at the end.
In Range check	How near it is to what it is dealing with, whether close enough to attack, too close and backing off, or far enough away to give up. It asks about the senses' target, and the template works whoever it is chasing.
Look At	Turn a node to face another node, the mouse pointer, or a place, instantly or at a turn speed. It keeps facing it while the state runs. With Face set to The Mouse , a turret or a top-down hero aims wherever the pointer goes. With Face set to A Direction it aims the way a direction points, such as the right stick's from Get Input Vector, and keeps its last aim when the stick is let go.
Move To	Move a node, or a panel, to a place over time, absolute or relative. Finishes on arrival. On an AnimatableBody it moves in the physics step, so a moving platform carries whoever stands on it smoothly.
Path Progress	Find out how far along a Path2D or Path3D something has got, however it got there. That tells you how far round a track a car is, which call a co-driver should make next, or how much of a level the player has crossed. The thing does not have to be on the path. It can also store the share of the path covered, the nearest point of the path and the way the path runs there, which is where to put a car back after it has left the road, and how far off the path the thing has strayed.
Punch	Kick a node's Position , Rotation or Scale and let it spring back to exactly where it was, wobbling as it settles. Think of a coin that squashes and stretches, an enemy that shudders when hit, or a button that bounces. The kick is added to whatever else moves the node, so a character keeps walking while it shudders, and leaving the state part way puts it straight back.
Rotate	Keep a node turning at a steady speed in degrees a second while the state runs, such as a coin spinning, a fan, or a pickup turning to catch the eye. The same speed on any computer. Like Translate it does not hold the state up.
Rotate To	Turn a node to an angle in degrees, with one Angle for a 2D node and Tip , Turn and Roll for a 3D one. Once, it turns there over its Duration, the short way round, so from 350 degrees to 10 is twenty degrees, not three hundred and forty. On Every Frame it is set to the angle every

	frame and follows a variable that keeps changing. Make a Number called angle, add to it with Set Variable Per Second, and Rotate To keeps the square turned to it.
Scale To	Grow or shrink a node to a size over time, then finish, the way Move To moves it. A size of 1 is the size it was made at, and 2 is twice that. Relative multiplies the size it has, and one number scales it the same every way. A RigidBody2D keeps its own size, so scale the picture under it.
To Global	Turn a place in a node's own space into a place in the world. For example, it finds the spot two metres in front of a gun, (0, 0, -2), wherever the gun is and whichever way it points. On Every Frame the answer follows the node as it moves.
To Local	Turn a place in the world into a place in a node's own space, measured from the node along the way it faces. It tells whether something is in front of the player or behind, or left or right of a turret. In 3D a node's front is minus Z.
Translate	Keep a node moving at a steady speed while the state runs, forward (the way it faces) or along a direction, like a bullet flying the way it was fired or a cloud drifting. A physics body is given the speed, and anything else has its position moved. Every node kept moving is moved together in one step, so a thousand bullets in the air cost little and their machines rest until something happens to them. A setting read from a variable is read again every step.
Tween Property	Ease <i>any</i> property, picked from a list of the ones that node can ease, to a value over time with a choice of curve. Slide a panel, fade a colour, grow a pickup or drop a door. One action covers move, rotate, scale and fade, because they are all just properties.

10.11 Physics, for forces, velocity, gravity, raycasts and collisions

Godot moves a walking character with a **CharacterBody2D** or **CharacterBody3D**, and such a body only moves when something calls its `move_and_slide` each physics step. **Move Character** is that something. Setting a speed on its own (with Set Velocity, or with Jump) is only half the job. A state that has no movement action in it does not move the body at all, and the character will hang where it is.

For dropping through a platform with the **Set Collision** action, see 8.2.

The usual shape is to put a Move Character on the **parent state**, so everything nested inside it keeps walking while the sub states get on with jumping, attacking or being hurt. Parallel states do the same job side by side.

Action	Does
Apply Force	Push a physics body with a steady force, a sudden impulse or a spin, in world or local space. Force is two numbers in a 2D game and three in a 3D one, and a torque in 2D is one number, Torque. Torque on Once is one sudden turn, and on Every Frame a steady twist that keeps it spinning faster. On a character body it is a knockback. While Move Character, Chase or another movement action is walking it, the push carries it on top of the walking and dies away by itself within a moment.
Drive	Drive a vehicle with the keys or a gamepad. It can accelerate, brake into reverse, and steer, turning more the faster it goes. On a VehicleBody3D, Godot's car on wheels, it works the

	engine, brake and steering. On a CharacterBody2D or CharacterBody3D it drives the arcade way, for a top-down racer or a kart, and turns the wheel models the car came with (any part named wheel, tyre or tire) so the front ones steer and all of them roll, and leans its Body Model into the corners as if on springs. Name a key or a button under Handbrake Input and the car has a handbrake. Grip is how firmly the car's travel follows its nose, and Handbrake Grip is the same while the handbrake is pulled, so a small number lets the tail step out round a hairpin. Store Slide In keeps how far sideways the car is going, for tyre smoke or a drift score. A trigger pulled half way is half the throttle, and Store Speed In feeds a speedometer. Turn At Top Speed takes some of the turning away as the car goes faster, so a fast corner has to be braked for. When the car hits something hard enough to lose at least Crash Share of its top speed at once, it sends Crash Event and keeps how hard it hit in Store Crash In, for dents, punctures and a crunch. Scraping along a wall is not a crash. Pull steers a little by itself, as a puncture or a bent track rod does.
Event On Collision	Send an event when something touches this node, or stops touching it, such as the player entering a trigger, a bullet hitting a wall, or a coin being picked up. Point it at an Area2D or Area3D, limit it to one group so a door ignores everything but the player, and it remembers what touched it for the next action to use. On a rigid body it turns on contact reporting itself.
Explosion	An explosion's reach. Every node in a group within Radius is sent an event carrying damage, greatest at the centre and falling away evenly to Damage At Edge, and a transition can store it in a variable. Each machine is told once, however many of its nodes are in the group, so a soldier whose body and head are both enemies takes one blast. Rigid bodies inside it are thrown outward by Push, and with Walls Shelter ticked, anything behind a wall is spared. Store Count In keeps how many were caught. Put two in one explosion to hurt the enemies and the player by different amounts. Nothing in the template has to wait for its event, because the machines it reaches do. Leave Group empty to reach every machine within Radius, and leave the event empty for a blast that only pushes. Anything it reaches with a Hurt Box is hurt through it instead of being sent the event, as a Hit Box would hurt it, with health off, a flash, and HURT or DIED.
Freeze	Hold a RigidBody still where it is and let it go again, or switch its gravity off and back on. It makes a crate that stays put until pushed, a block caught by an ice spell, or a balloon that floats. It can change back after a time too.
Get Contact	Where a rigid body is touching something and how hard. It gives the node, the point, the way the surface faces and how fast they came together. Hit Event is sent each time something new starts touching it, on each bounce and each landing. It switches the body's Contact Monitor on by itself. For a character body, use Get Last Collision.
Get Overlapping	Ask an area what is inside it right now and put the answer in a list, such as every enemy in the blast or everything standing on the plate. It can also store how many there are and which is closest.
Get Velocity	How fast a character body or rigid body is moving and which way, as a velocity, a speed (one number, 0 when still), or both. Local Space gives it along the body's own directions.
On Screen check	The node is inside what the camera shows right now, in 2D or 3D, but a hidden node is not. Margin lets a big thing count while any of it shows. Something standing in the way still counts. For a clear line of sight, the check is Can See.

Raycast	Look along a line and store what is there, how far and where, sending an event on a hit or a miss. It finds the ground under a foot or a wall ahead. When it stops hitting, the variable holding what it hit is emptied, but the place it last hit is kept. In a 2D game, Y counts down the screen, as everywhere else in 2D. Pick a node for Toward , or press Var to use the player in a variable, and it checks the line of sight to it instead. A miss is a clear view, and a hit is whatever is in the way. Spread aims a random amount off, up to that many degrees, each time it looks, as a gun does. Press Var to read it from a variable. Store Normal In keeps the way the surface hit faces.
Raycast All	Like Raycast, but finds everything on the line, not only the first thing, and stores lists of what, where and how far, nearest first. Think of a bullet that goes through a row of enemies. Max Results caps how many.
Set Body Properties	Change a physics body's Mass , Linear Damp , Angular Damp , Bounce , Friction or Lock Rotation while the game runs. Bounce and friction go on a material of the body's own, so other copies of the same scene are left as they were.
Set Collision	Switch a body's collision off or on, or change the layers it is on and the layers it bumps into, ticked by name, such as World, Player, Enemies, Pickups, or one of your own (2.6). Give it a time and it changes back by itself. A player drops through a platform for 0.3 seconds, or a ghost walks through walls for five.
Set Gravity	Change gravity for one character or physics body. Set Strength to 0.5 for the moon or 2 for heavy boots, or pick Upside Down , Right Way Up and Flip , which make it fall to the ceiling and walk on it, with its picture turned over too. Jump and Move Character follow the change by themselves.
Set Joint	Connect two bodies with a joint already in the scene, let them go, or change one of its values by name, for a chain that snaps, a rope grabbed and let go, or a hinge's limits or motor. Connect with neither body picked joins again the two it let go of. Angles are in degrees.
Set Velocity	Set how fast a body moves, per axis, on a rigid body or a character body, 2D or 3D, with two numbers in a 2D game and three in a 3D one. Leave an axis unticked to keep its fall.
Shape Cast	Everything inside a circle (a ball in 3D) or a box, with no area needed, or, with a Distance , the shape swept along a line to see what it runs into first, like a raycast with thickness. It stores the list, the thing found, the point, the distance and the normal, and sends an event on a hit or a miss, like Raycast.
Swing	Swing a character from a point as if on a rope, such as a grappling hook caught on a ring, a vine or a chain. It swings the way a pendulum does, fastest at the bottom and slowing at the ends. Holding a way pumps it higher, and with a Climb Speed holding up or down climbs the rope. Put it on its own in the state it swings in, on a CharacterBody2D or CharacterBody3D. It moves the body itself, so it needs no Move Character beside it. The point is any node, picked in the scene or held in a variable. The state is left by an event, such as the key let go or a jump, and the character flies on with the speed it had. Stiff makes it a pole that can go all the way round, and a Line2D picked in Rope Line is drawn as the rope.

Touching <i>check</i>	Something is inside this area right now, or a character body is on the floor, against a wall or against the ceiling. Can be limited to one group, and can store what is being touched. Where Event On Collision fires as something arrives, this asks whether it is still there, so a state can be left the moment the contact ends.
---------------------------------	--

10.12 Character, for a character that walks, jumps, dashes, swims and climbs

Action	Does
Click To Move	Walk where the player clicks, as action role-playing games do. Click the ground and the character walks there, round walls when the level has navigation mesh (10.21). Hold the button and it follows the pointer. It fills in the same direction variable Get Input Vector does, so Move Character walks it, and the keys still work. Click something on a layer in Targets , such as Enemies, and it walks up to within Reach and sends In Reach Event , keeping what was clicked in Store Target In for the state that strikes it. Click something on a layer in Uses , such as Pickups, and it walks up and sends that thing its Used Event instead, so a chest opens on USED. A click on an open inventory or a button is theirs, not a walk.
Crouch	Makes a character shorter while the state runs and its own height again when the state is left, for crouching, ducking under a swing or crawling through a gap. It changes the collision shape, not just the picture, so a shorter character really does fit under things a tall one cannot, and it keeps the feet where they are so it ducks rather than sinking. While crouched it watches the space overhead and sends Room Above when there is room to stand and No Room Above when there is not. Leave the crouch on the first of those and a character can never stand up inside the ceiling it just crawled under. The shape is copied rather than changed, so other characters built from the same shape stay their full height. Simplest of all is Stays While Held . Name the control the player holds to crouch and the action stays down while it is held and stands up by itself the moment it is let go <i>and</i> there is room, sending Stood Up . That is the whole of hold-to-crouch in one box.
Dash	Sends a character shooting off in one direction for a moment, then finishes. Think of a dodge roll, a dart across a gap, or an air dash to a ledge one jump too far. You give it a distance and a time and the speed is worked out for you, so every dash covers the same ground. It can go The Way It Is Moving , The Way It Is Facing , along a direction (the variable a Get Input Vector fills in drops straight in) or at another node. Gravity is off while it runs unless you ask otherwise, and an air dash does not sag. Wait Between stops it being held down, and One Until It Lands stops a player crossing a level without ever touching the ground.
Get Last Collision	What a character body last bumped into as it moved, meaning the node, where they met and the way that surface faces. A character on the ground always touches the floor, so pick a group in Only to find the last enemy or wall it ran into.
Get Water Depth	Stores how far under the water a character's feet are in Store In , in metres in 3D or pixels in 2D, and 0 out of the water. Lower a Move Character's Speed by it, so the deeper the water the slower the wading. Water picks the water, an Area or a Terrain with a lake or sea, and left empty it is any water in the level.
In Water	A check, true while a character is in the water as deep as How Deep says. Feet In Water is as soon as its feet are wet, Waist Deep when the water reaches its middle, and Head Under when it is all under, which is when to start holding its breath. Water picks the water, an Area or

	a Terrain with a lake or sea, and left empty it is any water in the level, which is every Area in the group water. The recipe Swim And Hold Your Breath uses it.
Jump	Makes a character body jump. You say how high in metres or pixels and it works out the push from the gravity it is under. A jump is set in something you can see. It can refuse unless the body is standing on something, and sends an event either way, so the graph can do something else when the jump is refused. Pick a Cut Short Input, normally the jump button, and letting go of it while still rising cuts the jump short, so a tap is a hop and a held press goes the whole height. Tick Off Walls to jump off a wall the character is sliding down, pushed away from it by Away From Wall. That is a wall jump. Two settings decide how forgiving it is, and both start at a tenth of a second. Still Allowed After A Ledge keeps the jump available for a moment after the character walks off the edge, and Waits To Land For holds a jump pressed just before it lands and takes it the instant it touches down. Together they are the difference between a jump that feels fair and one that feels like it missed your press. Set either to zero for the strict behaviour.
Ledge Grab	Catches the top of a wall the character is pressed against and pulls it up onto it, then finishes. A jump that falls a little short becomes a climb instead of a drop. Reach Up is the highest top it can take, Over The Edge how far past the lip it ends up, and Seconds how long the pull-up looks like it took. It places the character up first and over second, the way a person climbing really moves, and sends an event when there is no ledge within reach so the template can go back to falling.
Move Character	Walks a CharacterBody2D or CharacterBody3D. It pushes it along, keeps it on the ground, slides it around walls and, if you ask, turns it to face where it is going. Go the way a direction points (the variable the Get Input Vector action fills in drops straight in) at its Speed. It can go at A Velocity, a direction whose length is the speed, such as a direction a Vector action has scaled or an Expression has worked out. Or it can head for another node and stop short of it, sending an event when it gets there. Directions can be read as the world's, the body's own, or the camera's. In a 3D game, players expect the keys to follow the camera. Speed, acceleration and gravity are all settings, and gravity can be turned off for a top-down game. With gravity on, four more settings shape the fall. Falls Faster Than It Rises pulls the character down harder than it went up, so a jump rises floatily and drops smartly. Floats At The Top eases gravity around the top of the arc so it hangs there a moment. Fastest Fall caps how fast a long drop can get. And Acceleration In The Air lets steering build up more slowly off the ground than on it. All four start at the plain arc and change nothing until you set them. In a 2D side-on game, Slips Round Corners forgives a jump that clips the corner of a ceiling by a few pixels. It slides sideways round it and carries on up instead of bumping its head. If you would rather not think about any of them, use Feel. Pick Tight, Floaty, Heavy or Arcade and the five numbers are filled in for you, worked out from the Speed you have set so that one choice is right in a 2D game and a 3D one alike. It goes back to Custom once it has, because from then on they are yours to change. Jump's two forgiveness settings are not part of it, because they already start where they should, and setting both to zero is what makes a strict arcade jump. Seen from the side (a body that falls), turning to face flips its sprites left and right and keeps it upright. Seen from above, it turns the body. Toward A Place walks it to a place and stops, such as where the player clicked in a point-and-click game. Press Var on the place to read it from Mouse Position's variable.
Stamina	Keeps a meter that empties while something is being done and fills up again while it is not, such as stamina for running, breath underwater, a shield that holds only so long, a boost tank

	or a torch that runs down. It keeps one variable between zero and Most, so the rest of the template reads it like any other variable. Show it on a bar, ask about it in a Check, or leave the state when it runs out. Put it in the state that is spending and it drains there. Put it in the states that are not and it fills there. Rests First is what stops a player tapping the button to keep the meter full, and unticking Spends Here makes a state one that only lets the meter come back. Spending in one state and resting in the one it returns to is a whole stamina system with no sums in it at all.
Step Up	Lets a walking character walk up a step instead of being stopped dead by it, such as a kerb, a stair, the lip of a doorway or a low crate. Without it a character body treats every one of those as a wall. Put it in the same state as the Move Character that is walking the character, above it in the list, and set Tallest Step to the highest thing it may climb. Anything taller is still a wall. It works in 2D and 3D and does nothing at all unless the character is standing on something and pressed against something else.
Swim	Lets a character move through water instead of falling through it. The water is an Area you draw in the scene, in any shape, or a Terrain with a lake or sea. Inside it gravity stops pulling and the character settles, sinks or bobs up at Settles At, with Up Input and Down Input to swim at Swim Speed. Put it beside the Move Character that is steering the character and lower that Speed for the swimming state. Its weight comes back the moment it is out of the water, and again when the state ends, so a character can never be left hanging in the air.
Wall Cling	Catches a falling character on a wall so it slides down at Slide Speed instead of dropping past, or sticks to the wall when that is zero. With a Jump set to go Off Walls it is the other half of wall jumping, and on its own it turns a shaft into something a player can climb. Set Only While Held to a control and the player has to press into the wall to hold on, and Lets Go After puts a time limit on it so a wall cannot be rested on for ever. It sends an event when it catches the wall and another when it comes off, whichever way that happened.

10.13 Input, for keys, mouse, touch and gamepads

Every setting that asks what to watch lists the keys, mouse buttons and gamepad inputs by name (**Key: Space, Mouse: Right Button, Pad: Left Stick Left, Pad: Right Trigger**), and they work with nothing set up in Project Settings. A stick or trigger counts how far it is pushed, so a character walks slowly on a gentle push. As a press, a stick pushed one way counts only once it is half way that way (for Pressed, Just Pressed, Just Released, Double Tapped, Held For and every “while held” setting), so a thumb a little off level while running never presses the way beside it. A trigger counts from a light pull, unless it shares a setting with a stick direction, when it counts from half way too. An input action you added to the Input Map, used on its own, keeps the Deadzone you set there (set 0.5 on one for a stick, or a thumb a little off level presses it). Joined to other inputs with **or**, it counts like them. Input actions you have made in Project Settings > Input Map are listed first and work the same way. They are handy when one action should answer to several keys at once. Two inputs that do the same job can also share one setting, joined with **or**. For example, **Key: W or Pad: Left Stick Up** walks forward from the keyboard and from a gamepad. In the list beside the field, tick **Add the one picked to what is there** and pick the second one.

One press, one step. A press that has just moved a machine into a state does not count again for the state it arrived in. So the same key can crouch in Standing and stand up in Crouching, and one tap does one of them, not both. Parallel states that were already running still all hear the same press.

Trying a phone game on a computer. The touch actions count the left mouse button as a finger and the mouse wheel as a pinch, so a mobile game can be played in the editor before it ever reaches a phone. Untick **Mouse Counts** on one that should ignore the mouse. On a phone, taps also work as clicks, so buttons and clickable nodes made with **pressed** need nothing extra.

Action	Does
Device Motion	Read a phone's or tablet's motion sensors into Vector3 variables, every frame. There is Store Gravity In for which way is down (tilt steering reads this), Store Accelerometer In for a shake, Store Gyroscope In for turning, Store Magnetometer In for a compass. A computer has none and reads (0, 0, 0). On a phone each sensor also has to be ticked in Project Settings, under Input Devices > Sensors, and the Output panel names any that are not.
Drag	Pick something up with the mouse and move it. Press on it, move, and let go to drop it, with an event for each. It needs a collision shape to be pressed on, unless it is a sprite or a control, which are pressed on where they are drawn. A rigid body is held still while carried. In a 3D game it slides across the ground at the height it was picked up. It never holds the state up.
Event On Input	Send an event when a key, a mouse button or a gamepad input is pressed, released or held, tapped twice quickly (Double Tapped , for a dash, where a stick must come back near the middle between the taps), or held down for a time (Held For , for a long press). Store Held Seconds In keeps how long it has been held, so a shot can charge while the button is down and fire on release with the charge. Pick Any Key or Button for "press any key to start". Like Send Event, it can carry a value with its event, typed in Data or read from a variable. Three keys can each send BEATEN carrying a different kind, and one transition puts whichever it is into a variable.
Get Action Strength	Store how far an input is pressed, from 0 to 1. A trigger pulled half way for a car's accelerator reads 0.5, a stick pushed one way, or a key, which is 1 while held and 0 when not.
Get Input Vector	Put the direction the player is holding into a variable, for a machine to drive a character with. A stick pushed a little gives a short direction and pushed all the way a full one, so Move Character walks slowly or at full speed to match. Tick Keep Last Direction for aiming with a stick. When the stick is let go, the direction stays the way it last pointed. Without it the direction drops to 0. Clear Up and Down for a side-scroller and the character only goes left and right, and the pair you leave filled in carries on working.
Input check	An input action is held, or was just pressed or released.
Input Combo	Send an event when keys or buttons are pressed one after another in order, each within Within Seconds of the last, such as Down then Attack for a slide kick. Pressing something else, or waiting too long, starts again.

Joypad Connected	Store how many gamepads are connected, and send an event when one is connected or disconnected while the state runs. For example, pause the game when the player's pad runs out of battery.
Last Device	Store which device the player used last, as the words <code>keyboard</code> , <code>mouse</code> , <code>gamepad</code> or <code>touch</code> , and the make of gamepad as <code>xbox</code> , <code>playstation</code> , <code>nintendo</code> or <code>other</code> , sending an event when the player changes device. A Check on the variable then shows "Press E" or the pad's button picture. A small knock of the mouse or a stick resting off centre does not count as a change.
Mouse Motion	Store how far the mouse moved since the last moment, as two numbers, across and down in pixels (0, 0 while it is still). It can turn a turret, steer a ship or spin a dial with the mouse. It works with the mouse hidden and held by Set Mouse Mode. Sensitivity scales it, and Invert Up And Down flips up and down. Mouse Look is simpler for a first-person view.
Mouse Pick	Store what is under the mouse, casting from the camera through the cursor, optionally only while a button is held.
Mouse Position	Store where the pointer is as a place in the game's world, and if you like its place on the screen. In a 3D game the place is the spot on whatever is under the pointer, or on flat ground at Ground Height . Put it in a variable and Move To, Instantiate or Look At can use it. Store Wheel In keeps how far the mouse wheel turned, in clicks, for zooming by hand.
Pinch	Two fingers zoom a camera. Spread to zoom in and pinch to zoom out, between Most Zoomed In and Most Zoomed Out . Works with a 2D and a 3D camera, or stores the zoom in a variable instead.
Rebind Input	Let the player choose their own keys. The state waits for the next key, mouse button or gamepad input and binds it to a control, sending Bound Event , or Cancelled Event if the player gives up with Escape (or, with Which Inputs set to Keyboard And Mouse or Gamepad , a gamepad's Select / Back). The choice is kept and comes back the next time the game runs. It can also only store which key a control is on, for the label beside it, or put every control back as it was. Which Inputs makes a Controls page with a keyboard column and a gamepad column. Keyboard And Mouse waits for and shows only keys and mouse buttons, Gamepad only gamepad buttons, sticks and triggers. With either, a gamepad's Select / Back gives up as Escape does, and a press of the other kind does nothing at all while it waits, so the pad's right button cannot close the menu behind a keyboard box. A control on a gamepad button never takes a stick push, so a thumb resting on the stick as the page waits cannot put Dash on walking left. Walking, on a stick or on keys alone, takes one. A new input only ever replaces inputs of the same kind, so a control on <code>Key: Space</code> or <code>Pad: Bottom Button (A / Cross)</code> keeps its gamepad button when Space is changed. The press taken as the new key does nothing else until it is let go. It does not also jump, and the Escape that gives up does not also close the menu. Input Action can come from a variable, so one row template serves every control. Give the control a name of its own, such as Jump . Pick its key in any input setting and choose Give these inputs a name... from the list. Every action naming Jump then follows the player's new key, and the name stays true. Rename this control... renames it in every template the scene uses, as one step to undo, and the problems button says when two actions start the same control on different keys.
Screen Orientation	Get puts portrait or landscape into a variable, from the shape of the window, and sends Changed Event each time the screen turns while the state runs. It works on a computer too,

	when the window is dragged taller than it is wide. Set turns a phone's screen to Landscape, Portrait, their Reverse or Sensor ways, or Sensor, which follows how it is held. The way a game starts is set in Project Settings, under Display > Window > Handheld.
Set Mouse Mode	Show, hide, capture or confine the cursor, for example visible for a menu and captured for a first-person view. Set Cursor to Picture to use your own picture as the pointer, such as crosshairs, with Hotspot the pixel that points. Normal puts the arrow back.
Touch Count	Put the number of fingers on the screen into a variable, every frame, such as two fingers to open a map or three to pause. The left mouse button counts as a finger unless Mouse Counts is off.
Touch Gesture	Send an event for a tap, a double tap, a long press, or a swipe left, right, up or down, on a phone or tablet. Leave an event empty to ignore that gesture. With Must Start On It ticked, only touches that begin on the node count, which makes an on-screen button out of any sprite.
Vibrate	Make a gamepad buzz for a time, with a Gentle fast buzz for small things and a Heavy slow rumble for big ones, or stop it. Pad 0 buzzes the gamepad the player used last. On a phone or tablet with no gamepad, the device itself buzzes.
Virtual Joystick	An on-screen joystick. A thumb put down in the left half of the screen (or the right, or anywhere) and pushed gives a direction in a variable, just as Get Input Vector does from the keys, and Move Character walks with it. It draws itself where the thumb lands. Two sticks, one each side, work at once. To walk with the keys as well, put Get Input Vector above it in the same state, storing in the same variable. The thumb has its way while it is down.
Warp Mouse	Move the mouse pointer at once, onto a node (a button's middle, a character), to a Position on the screen, or to the middle. Only Windows, macOS and Linux let a game move the pointer. Anywhere else, a warning in the Output panel says it cannot.

10.14 Animation, for AnimationPlayer, AnimationTree and animated sprites

A 2D character is a picture, and these are the two things every 2D game does to one. They work on a Sprite2D and an AnimatedSprite2D, and on their 3D cousins for a game that draws flat art in a 3D world.

Action	Does
Animation Event	Send an event at a moment in an animation as it plays, loop after loop, such as a footstep as each foot lands or the hit on the frame a sword swings through. Say whether the moment is counted in seconds, as an AnimationPlayer does, or in frames, as an animated sprite does, and give the one number.
Flip Sprite	Turns the picture around so it faces the way it is going. Almost every 2D character is drawn facing one way and flipped for the other. Left on Face The Way It Moves, this watches the body and does it for you, with nothing else to set up. Untick <i>Drawn facing right</i> if your art faces the other way. Flipped, Not Flipped and Toggle turn it once, as the state starts.
Play Animation	Play a clip on an AnimationPlayer, optionally waiting for it to end. Blend Seconds smooths it out of the clip before it, and Speed plays it faster or slower. Read Speed from a variable and

	the clip keeps in step with it, so a run can match how fast something is really moving. A clip set to loop never ends, so with one of those it finishes at once.
Play Sprite Animation	Plays one of an AnimatedSprite2D's named animations, such as walk, jump, attack or die. Left to wait for the end, the state finishes when the animation does and a transition on FINISHED carries straight on. An animation set to loop never ends, so with one of those it finishes at once. Put in a state that already does something and leaves on FINISHED, such as a jump, it starts not waiting, so the clip never holds the jump up. When Moving , such as run, plays instead while the character moves, and the Animation, such as idle, while it stands. One state that both stands and runs looks right either way. With no animated sprite on the character, the problems button says so and how to add one.
Playing <i>check</i>	An AnimationTree is in a named state, an AnimationPlayer or an animated sprite is playing, or a clip is a given fraction through (a clip that has played to its end counts). Leave a state when its animation ends. It asks about a sound player and about the music too, so a state can wait for a sound to finish.
Seek Animation	Jump a clip on an AnimationPlayer to a moment in it. Seconds of 0 rewinds it, and the clip's length shows a chest already open. A playing clip plays on from there at the speed it had. Pause holds it still at that moment, and Stop Animation's Resume plays it on. Leave Animation Name empty for the clip the player has. Stop Sprite Animation's Show Frame does the same for an animated sprite.
Set Animation Speed	Change how fast an AnimationPlayer, AnimatedSprite2D or AnimatedSprite3D plays, while it plays, for a fan winding up or a walk into a hurry. Speed Scale 1 is as made, 2 twice as fast, 0 holds it still, below 0 plays it backwards. It stays for every clip after this one until it is set again. On Every Frame with Speed Scale read from a variable, the speed follows the variable. A player an AnimationTree drives keeps the tree's own speed, and the Output panel says so.
Set Tree Parameter	Set one of an AnimationTree's parameters, picked from the tree's own list. A blend, as a number or a direction, from a value or a variable and optionally eased, tells the character how fast it is going. A condition, turned on or off, lets the tree's state machine move on. A OneShot, fired, aborted or faded out, plays an attack over the walk. Kind follows the parameter picked, and Wait For End holds the state until a fired OneShot has played out.
Stop Animation	Stop, pause or resume a clip on an AnimationPlayer, such as a fan switched off, a clip cut short, or everything held still behind a pause menu. Stop goes back to the clip's start unless Keep State is on, Pause holds it where it is, Resume plays on from there, and Toggle does whichever of the two it is not doing.
Stop Sprite Animation	Stop, pause or resume an AnimatedSprite2D or AnimatedSprite3D, or hold it on one frame with Show Frame . Think of a coin that stops spinning once taken, or a switch drawn as frame 0 off and frame 1 on. A frame the animation does not have is said in the Output panel.
Travel To	Move an AnimationTree's state machine to one of its states, such as start running or draw a weapon. The tree blends between the clips itself.

10.15 Audio, for sounds, music, volume and speech

Action	Does
--------	------

Mute	Silence a mixer bus, one player, or the music, and let it be heard again. This is the Mute box in an Options menu. Muting a bus leaves every volume as it was, and unmuting brings everything back where it stood. The music means the music itself, whichever bus it happens to play through, so silencing it leaves the rest of the game as loud as it was.
Pause Sound	Hold a sound still and let it go on again, for example the music while a pause menu is up or an engine noise while a car is stopped. A held sound carries on from where it was rather than starting again. Works on the music, on one player, or on every sound at Self.
Play Music / Stop Music	Play a music track, or stop it. The music is kept outside the scene, so changing scene does not cut it off, and it plays on while the game is paused, under a pause menu. One track plays at a time. Asking for the one already playing leaves it playing rather than starting it again, and asking for a different one fades the old out as the new fades in.
Play Random Sound	Play one of several sounds, picked at random, never the same one twice in a row, such as four footsteps, three grunts or a handful of impacts. The loudness and the pitch can wander a little each time, so a footstep played twenty times then sounds like twenty steps. It has a Bus setting as Play Sound does.
Play Sound / Stop Sound	Play a sound at the node the Machine works on, or through a player you point at. It can also stop it, or fade it out. A temporary player is removed once its sound ends or is stopped, and a looping one stops with its state. A one-shot sound plays to its end even after its state is left. Bus says which mixer bus a temporary player plays through, such as Effects, so the Options menu's volume reaches it. A player you point at keeps its own. Point it at a player and leave the sound empty to play that player's own sound, such as a looping skid already set up on the car.
Play Sound At Position	Play a sound where something is, rather than on it, such as an explosion as the barrel is destroyed or a splash where the stone went in. The sound is left standing in the world on a separate player and plays to its end even though the thing that made it has gone. Play Sound cannot do that, because its player belongs to the node.
Set Bus Effect	Switch an effect on a mixer bus on or off, or Toggle it, for everything muffled under water or an echo in a cave. The effect is added to the bus in Godot's Audio tab first, and picked here by name from the ones that Bus has.
Set Pitch	How fast a sound plays, which is also how high it sounds. Think of an engine rising with speed, a voice slowed to a growl, or a pickup that goes up a step each time one is collected. It can wander a little each time as well.
Set Sound Loop	Make a sound player's sound, or the music, loop, or stop looping so it plays to its end, even while it plays. The player gets its own copy of the sound, so the file and every other player using it are left as they were, and a playing sound carries on from where it was.
Set Volume	How loud something is, whether one sound player, the music, or a whole mixer bus. A bus is how a settings menu works. The sliders in Options set the Music and Effects buses, and everything playing through them follows. The buses offered are the Audio tab's, plus Music, Effects and Voice, which are made while the game runs where the project has none of that name. Given a number of seconds it eases there, for ducking the music under a line of dialogue and bringing it back.

Text To Speech	Say words out loud in the computer's own voice, with no recording, such as a co-driver calling the corners, a narrator, or a menu read out for somebody who cannot see it well. The words can hold variables, such as a score or a name. The voice is the computer's, so it sounds different on a Mac and on Windows, and it does not go through the game's buses, so its own Volume sets how loud it is. A computer with no voice installed says nothing, and the Output panel says why.
-----------------------	--

10.16 Camera, for cameras that follow, shake, zoom and look around

The first four camera actions work with a Camera2D and a Camera3D alike. Point them at the camera and they take the right path themselves. Mouse Look and Orbit Camera are for 3D games.

A first-person player and a third-person player built from these actions are in 8.2.

Action	Does
Camera Follow	Keep the camera on something, easing after it, with an offset that decides where it sits on screen, an option to turn and face it, and Behind It , which turns the offset with what it follows so a chase camera stays on a car's tail through the corners. It never holds the state up, so put it beside whatever else the state does.
Camera Limits	Keep a 2D camera inside an area so it never shows past the edge of the level. Left with no Area , it is the whole level, meaning every solid thing in the scene (the ready-made Ground and Walls, any body with a shape) and every tile layer, put together, so one Camera Limits in the player's first state is all a level needs. Pick a node and it is the solid things and tiles under that node, the tiles of a TileMapLayer, or a room drawn as a ReferenceRect. Change it as the player walks into the next room, or lift the limits. With nothing to measure, the problems button says so.
Camera Shake	Rattle the view for an explosion or a heavy landing. The shake fades out on its own and is put back where it was when the shake ends or the state is left, so it can never leave the view crooked. It moves the camera's offset, not its position, so a camera that is following something keeps following while it shakes. Two shakes on one camera at once, an explosion during a heavy landing, settle together where the view was. Scale By Setting names a global number from 0 to 1 that every shake is multiplied by, <code>screen shake</code> unless you change it. A Screen Shake slider on an Options page that stores into a global variable of that name turns every shake in the game down, or off, for a player who wants less. A game without that variable shakes at full strength.
Camera Zoom	Zoom in or out over time, closer for a dialogue and wider for a fight. A 2D camera's zoom is a multiplier, where bigger is closer. A 3D camera's is its field of view in degrees, where smaller is closer. For pixel art, set Show Height instead. The 2D camera zooms so that many pixels of the game fill the window's height, on any screen and in any project, and keeps doing so while the state runs if the window changes size. With Whole Steps ticked it zooms only by 2, 3, 4 and so on, so every pixel of the art is drawn the same size.
Fade Screen	Fade the whole screen to a colour, or back from it, such as black when the player dies, a white flash, or a level that opens out of black. Out covers the screen and In takes the colour

	away. An In on a screen not covered starts fully covered. It finishes when the fade is done, ready for the next state. It counts real seconds, so it keeps going while the game is paused, and the colour stays through a change of scene until a Fade Screen In takes it away. To fade only while going to another scene, Change Scene's own Fade Seconds does it in one action.
Get Current Camera	Put the camera the game is seen through now into a Node variable, 2D or 3D. This lets you go back to it after a cutscene's Switch Camera, or shake whichever camera is in use.
Head Bob	Bob the view as a first-person character walks, harder the faster it goes, and dip it on a hard landing, sending Landed Event. Step Event is sent as each foot comes down, more often the faster the character walks. Lead it to a state with Play Random Sound for footsteps. Stride is how far a step carries the character at Full Speed. Going faster lengthens it, the way a person runs with longer steps and not only quicker ones, so a sprint does not patter. Put a plain Node3D between the head and the camera and bob that, so looking up and down is never disturbed. The bob is added on top of wherever the node is, so a crouch easing the same node down still works, and a Walking and a Sprinting state each with their own Head Bob carry on one smooth bob between them. The state can still finish while it bobs.
Mouse Look	Look around the way a first-person game does. Moving the mouse left and right turns the body, up and down tilts the head, and the right stick on a gamepad does the same. Set Node to the character's body and Head to the camera under it. Lowest and Highest stop the view tipping over. Press Var on Sensitivity to read it from a variable that aiming down the sights or an options menu can change. Smoothing takes the jitter out of a shaky mouse or stick. The mouse is hidden and kept in the window while the state runs, and given back when it is left. It does not stop the state finishing. While it has the mouse, Esc gives it back and a click in the game takes it again, with nothing to build.
Orbit Camera	A third-person camera. It circles whatever Around is set to, turned by the mouse or the right stick, zoomed with the wheel between Closest and Farthest, and pulled in front of a wall rather than looking through it. It starts from wherever the camera was placed. Set Only While Held to Mouse: Right Button to turn it only while that button is held, leaving the pointer free. Set Over The Shoulder to slide the whole view to one side, so the character stands off-centre and you can see past it, the way most third-person games are played. It stays on that shoulder as the view turns. Its state can still finish while it runs.
Recoil	Kick the view up, and a little to one side, the way a gun climbs when it fires, then let most of it come back down (Comes Back). Put it in the state that fires one shot. Held fire climbs, because each kick lands before the last has settled. It turns the camera on top of wherever the player is looking, and keeps settling after the state is left.
Screen To World	Turn a point on the screen into a place in the world, for Move To, Look At or Instantiate, such as the place under a crosshair in the middle of the screen, or where a finger touched. In a 2D game it follows the camera wherever it has moved or zoomed to. In a 3D game Depth picks the place on the line from the camera. It can be a distance in front of it (Z Depth), flat ground at Ground Height, or the first thing the line meets (Intersect Ray). Mouse Position does the same for the pointer.
Set Background	Change the colour behind everything, at once or over Over Seconds, for a sunset, a red flash or a dark room. In 2D it is the clear colour, which stays into the next scene. In 3D it is

Color	the WorldEnvironment's colour when its Background Mode is Custom Color. A sky has no colour to change, and the Output panel says so.
Switch Camera	Look through a different camera, such as a cutscene angle, a security camera, or the view over a player's shoulder.
Weapon Sway	Let something held in front of the camera, such as a gun, lag a little behind as the view turns and drift back when it stops, so it feels held, not glued to the screen. It follows the turning itself, so the mouse, a stick and a recoil kick all sway it. Amount is how far a quick turn swings it, Most the furthest it goes, Return Speed how quickly it comes back. It moves the thing on top of wherever it is, so easing it for aiming still works, and puts it back when the state is left. One for each thing held.
World To Screen	Find where on the screen a node or a place shows, as a point in pixels from the top left, to put a name or a health bar over a character with Set Property. Store In Front In says whether it is in front of the camera, which matters in 3D, where something behind the camera still gets a point. Store Inside Screen In says whether it is on the screen at all. Off Screen Indicator is simpler when all you want is a marker kept over something.

10.17 Effects, for particles, marks, screen effects and shaders

Action	Does
Emit Particles	Start or stop a particle effect, restarting a one-shot burst.
Flicker	Make something flicker while its state is active. Visible switches it out of sight and back at random, like a failing bulb or a ghost. Energy wavers a light's brightness like a flame. Leaving the state puts it back as it was.
Leave Marks	Leave marks on the ground where something goes, for as long as its state is on, such as tyre marks behind a sliding car or tracks in snow. Point it at the part that touches the ground, such as a back wheel, and put it in the state that means sliding. The marks start and stop with the state and stay on the ground after it, following the ground over hills. The oldest make way once there are Marks Kept of them. Its Color can be read from a variable, giving each stage a different one. 3D only.
Screen Effect	Change the whole screen, with brightness, contrast or saturation on the WorldEnvironment in a 3D game, or the tint of a CanvasModulate in 2D. Fog and glow are Set Environment's. At once or eased over a time, such as grey when health is low.
Set Environment	Change the WorldEnvironment's fog density or colour, ambient light colour or brightness, glow, or background colour or brightness, at once or eased over a time. Think of fog rolling in, a cave turning cold and blue, or the sky going red. Screen Effect changes brightness, contrast and saturation.
Set Shader Parameter	Change a value a shader reads, picked from the shader, to flash a sprite white when hit or dissolve an enemy. At once or eased over a time. With no shader material, a name the shader lacks or a value of the wrong kind, it says so in the Output panel.

The first explosion is as smooth as the rest. A graphics card has to get each effect ready the first time it is drawn, and on a slower computer, or one running the game for the first time, that used to

freeze the game for a moment. NodeWorthy gets every effect, bullet and enemy a machine can put in the game ready while the level loads, out of sight, so there is nothing to set up. Because of this a level can take a moment longer to load the very first time, and after that it is quick.

10.18 UI, for labels, textures, values and choices on the screen

The pieces of a menu, built without a line of code. There is a panel that fades in, words that follow a variable, a bar that follows a number, a button the keyboard starts on, and the game held still behind it.

How an Options menu is put together from these actions is in 8.2.

A button becomes an event with **Event On Signal** listening to its `pressed` signal, and a slider with `value_changed`. There is nothing to spell, because the **Signal Name** field lists every signal the node really has, with what each one hands over. That is the whole trick. The menu's buttons send events, and the machine decides what each one does.

Action	Does
Get Focus	Put the button, slider or box that has the keyboard or pad focus into a Node variable, left empty when nothing has it. On Every Frame it follows the focus round a menu. Grab Focus sets the focus, and this reads it.
Get Text	Read the words in a label, a text box, a button or a rich text box into a text variable, such as the name typed on a new-game page or a code typed into a keypad. On Every Frame it keeps up as the player types. A rich text box gives its words without the tags that style them.
Get Value	Read a slider, a spin box, a scroll bar or a progress bar into a variable, and keep it up to date while the state runs. Value Changed Event is sent each time the value changes, carrying the new value, with the variable already written. A volume slider on an Options page is this and Set Volume .
Grab Focus	Put the keyboard on a button, for a menu played with a pad or the arrow keys. The arrow keys and the D-pad move between buttons and Enter presses one. So does the pad's bottom button (A on an Xbox pad, Cross on a PlayStation one) while Pad Presses Buttons is ticked, as it starts. Godot's own settings leave that button out, so untick it to keep the Input Map exactly as the project has it. A button inside a scrolling list is scrolled into sight, even on a page the same state has only just shown.
Off Screen Indicator	Keep a marker you made, such as an arrow, over something in the world. When it is off the screen the marker waits at the nearest edge, turned to point at it, for objectives. Names and health bars can ride over characters the same way. Tick Around The Middle for a damage direction arrow instead. The marker then sits on a ring Radius pixels round the middle of the screen, turned the way the thing is, straight ahead being up and behind being down. Read the target from a variable, such as the attacker Get Event Data stored.
Option Button	Fill, clear, read or set a drop-down list, such as window sizes, languages, or Easy, Normal and Hard. Add Items fills it, emptying it first unless Clear First is unticked. Select shows a saved

	choice without counting as the player's. Item Selected Event is sent each time the player picks one, with Store In (the item's number, from 0) and Store Text In (its words) written first.
Pointer Over UI check	The mouse pointer, or a finger, is over something on the screen that takes clicks, such as a button, a slider, a panel or an open inventory. Ask it before a click walks the hero or fires, so a click on a button only presses the button. Words that let clicks through, and an empty container stretched over the screen to hold the HUD, do not count, and clicks there reach the game.
Set Disabled	Grey a button out so it cannot be pressed, or make it pressable again. Think of Continue with no saved game, Buy with too little gold, or an Upgrade already bought. Press Var beside Button Disabled and set Run Mode to Every Frame , and the button follows a true-or-false variable by itself.
Set Text	Put words on a label, a button or a rich text box, with any <code>{variable}</code> filled in, so a score board is one action. After a colon, say how to show a number. The choices are <code>{time:mm:ss}</code> as minutes and seconds (1:05), <code>{time:mm:ss left}</code> the same for time left, rounded up, so a countdown starts on 1:30 and shows 0:00 only when the time is up, <code>{time:h:mm:ss}</code> with hours, <code>{score:0000}</code> padded with zeros (0042), <code>{speed:0.0}</code> with one decimal, <code>{coins:0}</code> as a whole number, <code>{volume:%}</code> a share from 0 to 1 as a percentage (0.8 shows 80%, for a volume slider). A Vector shows each part that way, so <code>{place:0}</code> is (100, 20). A list shows as its items with commas between, and a table as each name with its value, such as <code>potion: 2, key: 1</code> . A colour shows as the colour picker writes it, <code>#ff664d</code> . A node, an item, a picture or a sound shows by its name, and a variable holding nothing shows nothing. What a variable holds is shown as it is, even if it looks like a <code>{name}</code> itself. Give it Reveal Seconds and the words appear a letter at a time, the way a character speaks. Its state waits until they are all shown, and a Skip Input shows the rest at once.
Set Texture	Change the picture a Sprite2D, Sprite3D, TextureRect or TextureButton shows, such as the portrait beside the words, an inventory icon, or a door drawn open.
Set Value	Move a slider, fill a progress bar, set a spin box, from a number or a variable, as a raw value or as a share of its range. Health bars are this. Read from a variable with Min and Max , such as health never above 3 or above <code>max_health</code> , the bar goes from one to the other by itself, so a health of 3 is a full bar, not 3 of a hundred.
Set Visible	Show, hide or flip anything, with an optional fade, and stop a hidden menu answering the mouse. The fade keeps running while the game is paused (a pause menu can fade in) and ends at the panel's own see-through setting. A new fade ends one still running. A 3D node, or a CanvasLayer holding a whole screen such as a collection page, is shown or hidden at once. Blink Every keeps flipping it between shown and hidden, that many seconds apart, for as long as the state runs, for an alarm light or a warning that flashes. The state then never finishes, and leaving it puts the thing back as it was before the blinking began.
Show Choices	Offer the player answers as buttons in a box of your own, such as a VBoxContainer in the dialogue panel. Each answer sends its own event, and Store Choice In keeps which was picked. Name an Array Variable and the answers are its items instead, for choices only known while the game runs, such as the saves there are. Wait for Any Answer Event , and Store Words In says which was picked. The first button is ready for the keyboard or a gamepad, and the buttons go once one is picked. Give it a Row Button , such as Delete, and each answer gets a small button at the end of its row that sends Row Button Event instead.

Toggle Button	Read or set a tick box, a switch, or any button with Toggle Mode on, for fullscreen, subtitles or inverting the camera. Toggled On Event and Toggled Off Event are sent when the player turns it on or off while the state runs. Set and Toggle change what it shows without sending them, so a saved setting shown as the page opens is not taken for the player's.
----------------------	--

10.19 Scene, for changing scenes, pausing, quitting and the game window

Action	Does
Build Dungeon	Build a floor of rooms for a roguelike, laid out afresh each run from rooms you draw. Every room scene is the same size (Room Size), with its walls drawn and a door node in the middle of each side named DoorNorth , DoorSouth , DoorEast and DoorWest . Give it a Start Room , a list of Fight Rooms and how many to use, and if you like a Boss Room (put as far from the start as it can be), a Treasure Room , a Shop Room and a Secret Room (tucked between rooms). Every room can always be reached. Each room's machine is handed <code>room_kind</code> (start, fight, treasure, shop, boss or secret) and <code>room_cell</code> , and each door's machine <code>connected</code> (a room is on the other side) and <code>secret</code> (that room is the secret one), so rooms and doors can lock, open and hide themselves. Give those machines variables with those names. Player is moved to the start room, onto its PlayerStart node if it has one. The same Floor Seed builds the same floor. Zero builds a new one every time, and Store Seed In keeps the one used. Building again takes the old floor away.
Change Scene	Go to another scene, by scene or by path, from a variable if the level is chosen while the game runs, or Reload Current Scene to restart the level after losing, with the global variables put back as they were when it began (all but those under Keep As They Are , or none with Put Globals Back unticked). A player whose template has Carries Over Between Scenes ticked takes their variables, stats and inventories along and stands at Arrive At , and the scene left behind is remembered (see 6.19). With nobody carried over, the new scene's own player is stood at Arrive At. Like the path, Arrive At can be read from a text variable, so one door template can lead each copy somewhere different. It lets the game go first, so the new scene does not start frozen. Fade Seconds fades the screen out to Fade Color and back in around the change. A Loading Screen scene is shown while a big level loads in the background, and how far it has got, from 0 to 1, goes into the global variable Store Progress In names. Give the loading screen's own machine a variable of that name with Is Global ticked, and a Set Value on a progress bar shows it. A second change asked for while the screen is still fading out or loading is ignored, so a button pressed twice changes scene once. One asked for while the new scene is already in and only fading back in happens as soon as it has, and a Main Menu pressed the moment a level appears is not lost. Scene is picked from your scenes by name and kept as the scene's path, so a title screen and its level can go to each other. The problems button warns when a Change Scene holds a scene that leads back to its own machine. The list shows your levels and screens first. A coin's, an enemy's or an effect's own scene is greyed underneath, with the reason on hover, because a thing like that is put into a level with Instantiate, not gone to.
Get Scene	Put the running scene's Name or its Scene File Path into a String variable, for the level's name on the HUD or a Game Over screen whose Retry goes back to the level with Change Scene.

Pause Game	Freeze the game or let it go again. The machine that pauses is set to carry on by itself, so the state that unpauses can still be reached. Once the game runs again it goes back to how it was set, and the mouse goes back the way the game had it. Pick your pause menu under Keep Running and its buttons can still be pressed while everything else holds still.
Procedural Placement	Put many copies of a scene over an area at once, such as trees, rocks, grass or enemies. The area is a collision shape, as for Instantiate's Somewhere Inside. Scatter puts Count of them anywhere in it, and Grid puts one in each cell, nudged off the lines by Jitter . By Chance places them evenly at random. By Noise gathers them in clumps, woods and clearings, set by Frequency and Threshold . The same seed places them the same every time. 0 follows the game's own random choices, which Random Seed sets, and Store Seed In keeps the seed to place the same again after loading. Clear First takes away the copies placed before.
Quit Game	Close the game, now or after a moment so a sound can finish.
Screen Settings	Go fullscreen, go back to a window, switch between the two, set the window's size, or turn V-Sync on or off. It can also store whether the game is fullscreen and whether V-Sync is on, for a menu's tick boxes to start out right. In the editor's own Game tab the window cannot change, so run the game in its own window to see it.
Set Language	Show the game in another language, or step on to the next one the project has, and store the language's name for a label. Every label, button and line of text with a translation follows at once. The translations are added in Project Settings, under Localization.
System	Open a web page, copy text for the player to paste or read what they copied, store today's date, the time and the day of the week for a daily reward, store what the game runs on (windows, mac, linux, android, ios or web) and whether it has a touch screen, open a file or a folder with the computer's own program (a manual, the folder a screenshot went into), or store where one of the computer's folders is (the player's Documents, Desktop or Downloads, where the game itself is, or the game's save folder).

10.20 Save & Files, for saving the game, and files and folders on the computer

For a tool or a launcher made with NodeWorthy more than for a game. They ask where something goes, copy it there, and start another program. They need a real computer, and do nothing on a phone or in a web browser. Paths take `{name}` to drop a variable's value in, so a path can be built from a folder the player chose and a name they typed.

Action	Does
Change Text In File	Replace some words in a text file, everywhere they are found, with others that can come from a variable, such as the name the player typed into a new project's settings.
Choose Folder	Ask the player to choose a folder with the computer's own folder window, and store its path. The state waits until they choose or close the window, and sends Chosen Event or Cancelled Event . Start In opens it somewhere sensible, such as the Documents folder the System action found.

Copy Folder	Copy a folder and everything in it to another place. To is the new folder's own path. With Only If New ticked, as it comes, it refuses when anything is already there, so nothing of the player's is written over. Store Problem In keeps the reason as a sentence to show them.
Move To Bin	Move a file or a folder, and everything in it, to the computer's Bin (the Recycle Bin on Windows), such as an old save or a level the player made. The player can still take it back out of the Bin by hand. The top of a drive, the home folder, Documents and the game's own folder are never moved. Ask first, on a page with a Keep It button.
File Exists <i>check</i>	There is a file, a folder or either at a path on the computer. It can tell whether a save is there before offering Continue, or whether a name is already taken. A picture, sound or scene of the game itself, asked for by its <code>res://</code> path, counts too, in the finished game as in the editor.
Read File	Read a text file into a String variable. It can be one of the game's own, picked from the project, or one in the game's save folder by a plain name such as notes.txt. A missing file sends Failed Event, or is said in the Output panel. A game's own .txt file only goes into the exported game when the export's filters for files that are not resources include *.txt.
Run Program	Start another program and carry on, handing it Arguments such as a file to open. On a Mac an app is started the way double-clicking it would. Failed Event is sent when there is nothing at the path. To open a document with whatever the computer normally uses, System's Open A File is simpler.
Save Game	Keep variables between one run of the game and the next, such as a high score, the level reached, or whether the tutorial was seen. Save, load, delete a slot, or just ask whether one exists. With no variables picked it keeps every one the machine has and every global variable in the game, so the score and a key found come back too. Save The Scene keeps the whole scene. That means every machine's variables and state, where each machine's node stands and which way it faces, whether it is shown and how fast it moves, what every Inventory is carrying, the changes gear and potions are making to its numbers with the time each has left, what was destroyed and what was spawned. Load The Scene puts it back in that scene. What was destroyed before the save stays gone, what was destroyed since comes back, and what was spawned returns. Each machine starts again in the state it was in, then its variables are put back. Every value comes back as the kind it went in, a whole number, a position, a colour or a list, and a variable holding a node is neither saved nor overwritten. A file written by hand is read as it is. It writes to Godot's own save folder and needs no path from you. The file is readable JSON. That also means a determined player can edit it, so keep nothing in there that would spoil the game.
Write File	Write words into a text file in the game's save folder, or add them to its end, each on a new line, for notes the player typed, a diary, or a log of each run. The words can take variables in curly brackets, or a whole String variable with Var. It never writes into the game's own files, which cannot change once the game is exported.

10.21 Navigation, for finding a way round walls on the navigation mesh

Move Character walks in a straight line and knows nothing about walls. These find their way over **navigation mesh**, which is the floor of the level with its walls cut out. Adding **Navigate To** makes it for you the first time, laid over the whole level and baked from its walls, floors and tiles on the

World layer, and gives the character the **NavigationAgent** it walks with. It goes round the furniture instead of into it. Players and enemies are not walls to it. After moving a wall or a floor, right-click the level and choose **NodeWorthy** → **Bake Navigation Mesh**, or the characters walk round the level as it was. It also makes it in a scene that has none, and the problems button says when a template needs it. It is a node called WalkableGround at the top of the scene. Both work in 2D and 3D from the one action, for a character of any height.

Action	Does
Get Navigation Point	Put a place on the mesh into a variable. It can be somewhere random within a distance, for wandering, or the nearest spot that can actually be stood on, for putting a target back onto navigation mesh. Feed it to Navigate To and a wandering guard is two actions. With no navigation mesh in the level it sends its nothing event and stores nothing.
Grid Path	Find the shortest way across a TileMapLayer's grid, round its walls, and walk it cell by cell, or only keep the path in a list. It suits a tactics unit, a puzzle piece, or click to move on tiles. Solid says which cells are walls (tiles with collision, custom data, every tile on a walls layer, or every empty cell on a floor layer), and a list variable can block more, such as where the other units stand. No way there sends No Path Event .
Navigate To	Walk to a place, or to a node it keeps following as that node moves, along the navigation mesh. It finishes on arrival for a transition on FINISHED, sends an event when it gets there, and sends another when there is no way through at all (once the place has been out of reach for a moment, so a level still loading does not count). A machine can then give up and do something else.
Navigation Reachable check	There is a way across the navigation mesh to a place or a node, or (with <i>It has arrived</i> ticked) the walk is finished. Inverted, it is "there is no way through", so a machine can notice it is stuck.

10.22 TileMap, for reading and changing tiles and grid cells

A **TileMapLayer** is Godot's grid of tiles, one layer of a 2D map. Cells are counted in whole tiles from the layer's own origin. The cell offset (0, 1) means one tile below, the ground you are standing on.

What you usually read is the tile's **custom data**. The tile set author marks some tiles "ice" or "lava" under a name like `surface`, and these read that name back, so one action covers every kind of ground.

Action	Does
Get Cell	Asks what is at a place, At a node or the mouse pointer, such as what am I standing on, is there floor ahead, or which tile was clicked. Stores the tile's custom data, whether there is a tile there at all, and which cell it looked at, and sends one event when it found a tile and another when the cell was empty.

Grid Map Cell	The same for a GridMap , Godot's 3D grid of pieces. It can put a piece in a cell, turned a quarter at a time, empty a cell, or look in one and store the piece's name. At a node, with a cell offset such as (0, -1, 0) for the cell underfoot, or where the mouse points. The Item is picked by name from the grid's mesh library, and Orientation turns it.
Paint Terrain	Paint cells with a terrain from the layer's tile set while the game runs. Every tile picks its own edges so they join up, as the terrain brush in the TileMap panel does. Dig a tunnel, build walls, or lay a road. At a node, the mouse, one cell or a list of Cells , where Mode Path joins each cell only to the ones before and after it. Erase digs cells out, and the tiles round them close their edges. The terrains are set up in the TileSet panel first.
Set Cell	Changes the map while the game runs, to dig a tile out, build a bridge, drop a block or open a hole under someone. At a node or wherever the mouse points, for a building game. The tile, in Atlas Coords , is picked from the layer's tile set by its picture, so there are no sheet or tile numbers to look up. Custom Data Layer on Get Tile and the Tile check lists the custom data the tile set has.
Tile check	There is a tile at a place, or the tile there has a given custom data value. Inverted, "there is a tile ahead and below" is how an enemy turns round at the edge of a platform instead of walking off it.

10.23 Debug, with messages, checks and drawings for finding what is wrong

Action	Does
Assert	Make sure something that must always be true is true, with the same kind of question a Check asks, and stop the game the moment it is not. Pause freezes it where it went wrong so you can look round it, Quit closes it, Carry On only says so. The Output panel names the machine, the state and your Message . On Every Frame it never holds its state up, and Debug Only skips it in an exported game.
Draw Debug	Draw a Line , Ray , Circle or Box in the running game, over everything, in 2D or 3D, such as the line from a guard to what it chases or the way a gun points. On Every Frame it follows the node while the state runs. On Once it stays for Duration seconds.
Print	Print a line, or a warning or error, to the Output panel. <code>{target}</code> , <code>{state}</code> and <code>{variable}</code> are filled in, the same way as in Set Text.

10.24 AI, for guards, hunters and things that run away

All of these read the character's **Perception** node, described in 6.13, and all of them walk along the navigation mesh when the character has a NavigationAgent under it and in a straight line when it has not, so the same template works in a maze and in an open field. A straight line really is straight. Without a mesh, a character told to get behind a wall will walk into the wall.

Keep Distance and **Strafe Around** are made to run together, in the same state, in either order. One owns the movement towards and away from what it is dealing with, the other owns the movement round it, and each leaves the other's half alone, so you get an enemy circling at arm's length out of two actions and no arithmetic.

Crowds. Several characters holding *exactly* the same distance end up on the same ring and shove each other to a standstill. Three things help, and they are best tried in this order. First add **Keep Apart** to the state, which pushes them off one another. Then give Keep Distance a little **Spread**, so they do not all want the same ring in the first place. And if you want something cleverer, tick **Avoidance Enabled** on the character's NavigationAgent. NodeWorthy hands the agent what the character wants to do and uses the answer it gives back, but it ignores an answer of "stand still" and moves anyway, because a character frozen by its own avoidance is a fault with no visible cause. Godot's avoidance needs a navigation mesh in the level, but Keep Apart needs nothing.

Action	Does
Alert Allies	Tell the others. It hands every friend in range the target directly, so their own Perception node knows who without having to see them first, and sends each of them an event so their graph can react. One guard spots you and the whole room comes, so the enemies fight as a group. Untick Give Them the Target for an alarm that only makes them come and look, such as a gun going off. Then it needs no Perception node of its own.
Can See <i>check</i>	The character can see what it is dealing with right now. That means in range, in the cone and not behind a wall. Its False Event is sent the instant sight breaks.
Chase	Go after what the senses found, keeping after it as it moves, stopping a little short of it and sending an event when it arrives, where an attack usually begins. Losing what it chases never counts as arriving. It stands where it is until Look For Target or Target Lost says it has gone.
Flee	Run away from it, aiming at open ground on the far side, until it is a safe distance off or has forgotten it, then say so. It goes on watching, so danger that comes back while the same state is running is run from again.
Interaction	Put it on the player to use things by walking up to them. The nearest thing in Usable Group within Reach shows a prompt, "E to open", and the key sends it Used Event , USED, carrying who used it. Click To Move sends the same event, so one chest works with both. Only something whose machine has a way out on USED in the state it is in is offered, so an open chest stops asking. The words after the key come from the thing's own text variable called prompt, else Prompt . Pick a Label under Prompt to show them there instead of over the thing.
Keep Apart	Stops a crowd standing in each other. Anything in the group that comes within its personal space is pushed gently away, harder the closer it is. Put it beside Chase, Keep Distance and Strafe Around, because it only ever adds a push on top of what they decided. This is what keeps a pack circling instead of piling into one spot and shoving itself to a standstill.
Keep Distance	Hold a sensible gap. It closes in when too far, backs off when too near and stands still in between. Without it several enemies all walk into the same spot and end up inside each other and inside you. <i>Spread</i> varies the distance a little from one character to the next so a group does not form a perfect ring.
Look For Target	Keep watch while the state runs, and send an event the moment something comes into view or is finally given up on. Somebody it only remembers, out of sight, is not spotted again, so a

	state entered after losing sight of them searches rather than turning straight back. They are spotted again when they come back into view. It never holds a state up. Put it beside whatever else the state is doing, and a guard patrols <i>and</i> watches. It costs nothing while it watches, because the Perception node tells it when something turns up.
Make Noise	A noise that every character with a Perception node within Max Distance , and within its own Hearing Range, hears, through walls and whichever way it faces (a Perception with Hearing Range 0 is deaf), such as a gunshot or a bottle thrown to lure a guard away. Each one is sent Heard Event , HEARD, carrying where the noise was. One dealing with nobody remembers the place, so Search Last Known goes there. At is a node or a place, and Only Group narrows who hears it. It makes no sound itself, so play one with Play Sound At Position. Alert Allies tells friends who was seen, not where a noise was.
Patrol	Walk a round of markers, waiting at each. It goes round and round, there and back, or once through. Where it had got to is kept in a variable, so a guard that breaks off to chase somebody comes back to the right point. Instead of listing markers, Or Walk Past walks everything picked there, such as Everything under Route, in the order they sit in the scene. Press Var and read it from a String variable, and soldiers sharing one template each walk their own route, set under their Machine's Starting Values .
Search Last Known	Go and look where it last saw them, stand and look around, turning one way and the other, then give up. If it can hardly move for a second, because somebody is standing on the spot or something is in the way, it stops where it got to. This is the action that makes an enemy feel like it is thinking. It does not know where you are, but it knows where you <i>were</i> .
Strafe Around	Circle what it is dealing with instead of standing in front of it. Two enemies circling opposite ways read as a pack looking for an opening. Two standing still read as a pair of statues waiting their turn.
Take Cover	Get behind something. It looks at the places you have marked as cover, picks one that really hides it (checking that the thing it is hiding from cannot see that spot) and goes to it, saying so when it arrives and when there is nowhere to hide. Marking cover needs no new node. Put empty nodes beside the crates, under one node, and pick them in Cover Group . It counts as cover exactly what blocks a Perception node's sight, so another character is never taken for cover.
Target Lost check	It had something and has not been able to see it for a moment. The wait is the point, because without it somebody stepping behind a pillar for a tenth of a second would break off the chase. Ticking <i>Only when forgotten</i> waits for it to give up altogether.
Wander	Amble about near a spot with a pause between strolls, picking only places it can actually walk to when the level has a navigation mesh. It is what makes a village look inhabited.

10.25 Combat, for attacks, bullets, health and stats

Action	Does
Attack	Swing the weapon worn in the Weapon slot, the way the character is heading. It deals its damage plus the character's strength to whatever it touches that is in Hurts, once a swing. With nothing worn, a punch. The state waits for the swing. See 6.17.

Bullet Pattern	Fire a pattern of bullets, such as one aimed shot, a shotgun fan, a whole ring, a spiral, or a wall with a gap to slip through. From is where they start, and Aim is which way the middle of the pattern points. It can point the way From faces, straight at another node such as the player, along a direction such as the player's aim, or at an angle (0 is right, 90 down). Count bullets fire each shot, fanned across Spread degrees (360 is a ring). Wobble makes each one a little unsteady, and Gap leaves out that many side by side, somewhere different each time. With Every Seconds it keeps firing while the state runs, and Turn Each Shot turns the whole pattern a little more each time, which makes a ring into a spiral. Start Out starts the bullets a little way out, so a ring begins outside the boss. The bullets are plain scenes, an Area2D (or an Area3D in a 3D game) with a collision shape and a picture or a model, with no machines, moved and looked after together, so hundreds can fly at once. The bullet scene's collision layer says what it is and its mask what it touches (the hurt boxes of what it can hit, and the walls). A bullet that touches something in Hits (or a hurt box under it) sends that thing's machine Hit Event, carrying Damage, and Get Event Data's Store Sender In says who fired it. Then it is gone, unless Pierce lets it pass through more. Anything else solid, such as a wall, stops it, unless it still has Bounces left. Homing turns it toward the nearest thing in Hits, so many degrees a second, and it is gone after Lifetime seconds. Every setting can come from a variable, and items change those variables to make a player's gun faster or give it more bullets. Sound plays each time it fires, through Sound Bus (Effects unless you change it). Every gun firing the same sound shares one player that sounds only a few copies at once, so a room of enemies firing together stays a sound rather than a roar. In a 3D game the pattern lies flat, as a game seen from above needs. The bullets fly level at the height of From, facing the way they fly, and an angle or a direction reads the way a camera looking down sees it (0 degrees is east, 90 is south, and a direction from Get Input Vector drops straight in).
Heal	Give health back through a Hurt Box, never above the health's Max in the Variables tab, for a potion, a first-aid kit, or a campfire on Every Frame with an Update Interval. Its machine is sent Healed Event, HEALED, carrying how much was actually given. When it was already full, nothing is given and no event is sent. Something beaten stays beaten unless Revives is on, and then it gets up as a respawn does, safe for a moment. Hurt is its twin.
Hurt	Take health off something through its Hurt Box with nothing touching it, such as poison, a fall, lava or a trap. Its defence, blink, safe time and the HURT and DIED events are exactly those of a real hit. Pick an Attacker to knock it back from them. On Every Frame with an Update Interval longer than the Hurt Box's safe time, it is poison that ticks.
Modify Stat	Change a number variable under a name so it can be taken away again. For example, a boost adds 50 to speed as <i>Boost</i>, and taking <i>Boost</i> away later puts speed back, whatever else changed it meanwhile. Items do this for you, so this action is for boosts, curses and upgrades that are not items. Give it For Seconds and it takes itself away when the time is up, as a power-up that runs out does.

10.26 Color, for colours and tinting, fading and taking them apart

Action	Does
Get Color Components	Take a colour apart into float variables from 0 to 1. These are its red, green, blue and alpha, or with Color Mode HSV its hue, saturation and value. How see-through a ghost is

	now, how far round the colour wheel a rainbow has got. Set Color Components puts them back.
Random Color	Put a random colour into a Color variable. Any picks from the colour wheel within hue, saturation and value ranges (0.9 to 0.1 gives only reds), and From Gradient picks one of the colours at a drawn Gradient's points, or anywhere along it. A balloon, a slime or confetti in a new colour each time.
Sample Gradient	Pick a colour along a Gradient you draw by a number placed between Min Value and Max Value , into a Color variable. Think of a health bar green when full, yellow at half and red when empty, or a metal glowing hotter. On Every Frame it follows the number.
Set Color Components	Change only the ticked parts of a Color variable and keep the rest, such as its alpha for a fade, its hue to turn a gem blue or its saturation to grey it out. Each part is typed in or read from a float variable with Var, and a hue past 1 goes on round the wheel.
Set Modulate	Tint or fade anything drawn (a sprite, a shape, a Control, a 3D sprite or label). A hit flashing red, a ghost fading to see-through and a button greyed out all work this way. Modulate tints it and everything under it, Self Modulate only itself. Change sets the colour and alpha, only the colour, or only the alpha, for a fade. At once or eased over a time.

10.27 Lights, for turning lights on and off, and their colour and brightness

Action	Does
Set Light	Switch a 2D or 3D light on or off, or set its colour, energy, range or shadows. It suits a lamp on a lever, a torch that dims, or an alarm that turns the room red. At once or eased over a time. Switched off or on over a time, it fades, and keeps its own brightness for next time.

10.28 Material, for the materials 3D things are drawn with

Action	Does
Set Material	Swap how something is drawn for another material, and put the original back. It gives a red glow while enraged, a ghostly look, or a white flash for a moment when hit with For Seconds . In 3D it becomes the Material Override, in 2D the Material. With Include Children a whole character changes. A 3D material on a 2D node, or the other way round, is said in the Output panel.
Set Material Color	Change a 3D mesh's Albedo Color or its Emission glow, at once or eased over a time, for a crystal that glows when charged or a panel that turns green when unlocked. It colours a copy of the material made for that mesh alone, so every other copy of the same enemy keeps its colour.
Set Material Texture	Change the picture painted on a 3D thing (Albedo Texture), where it sits (UV1 Offset), or Scroll it across the surface while the state is active, for a conveyor belt, running water or lava. It changes a copy of the material made for that thing alone, the same copy Set Material Color paints.

10.29 Convert, for turning one kind of value into another

Action	Does
String To Number	Read the first number in some text into a float or int variable, such as the amount typed into a LineEdit or the 12 in "12 coins". An int drops anything after the point. Text with no number sends Not A Number Event , or says so in the Output panel.

10.30 Application & Display, for the game window, the frame rate and screenshots

Action	Does
Display Settings	The rest of an Options menu's display choices. These are the most frames a second the game draws (Max FPS , 0 for no limit), a window with no border, the resolution a 3D world is drawn at (Scaling 3D Scale , 0.25 to 2, where 0.5 is much quicker on a slow computer), where the window sits, or the window put in the middle of the screen. It can also store each of the four, for a menu to start out showing them right. Screen Settings has fullscreen, the window's size and V-Sync. The border and the window's place cannot change while the game is fullscreen or in the editor's Game tab.
Get Screen Size	Put the size of the visible game area and of the window into Vector2 variables, in pixels, for placing things at the edges of the screen. On Every Frame both follow the window as it is resized. Inside a SubViewport the area is that viewport's.
Run When Unfocused	What the game does while the player has switched to another window. It can Keep Running , Pause until they come back, or Lower Frame Rate to a few frames a second to spare a laptop's battery. Set it once as the game starts, and it holds through every change of scene. A game already paused by its own menu stays paused when the player comes back. To show a menu as well, wait for the system event Application Focus Out.
Set UI Scale	Make the whole interface bigger or smaller, for a text size option. You can Set it, make it Bigger or Smaller by a step, or Reset it to the size the game was made at, and store it for a menu and Save Game. In a 3D game only the interface changes. In a 2D game the world is drawn bigger too, as Godot scales everything flat in the window together.
Take Screenshot	Save what is on the screen as a PNG in the screenshots folder of the game's save folder, for a photo mode or a picture beside each save slot, with the file's path in a variable. The date and time go in the name unless turned off, and Hide UI leaves the menus and labels out of the picture. It needs a window, so a game run with no screen says so and sends Failed Event.

10.31 Video, for playing videos

Action	Does
Play Video	Play, pause, resume, stop or jump through a video on a VideoStreamPlayer, such as an intro, a cutscene or a film on a screen in the world. Wait For End holds the state until the video has played through, and Finished Event is sent when it ends, even after the state is left. In a state

that also waits for a key, use the event, as that state never finishes. Godot plays Ogg Theora videos, files ending .ogv.

10.32 Inventory, for items and finding, carrying, wearing and using them

An **item** is a file you make once and use anywhere. Make one with **New Item...** on the NodeWorthy panel's **+** menu (it goes in your project's `items` folder, or one you pick with **Folder...** in its window), with **New item...** at the bottom of any list where an item is picked, or by right-clicking a folder of your own in the FileSystem dock and choosing **Create New > NodeWorthy Item...** It opens in the Inspector. Give it an **Item Name**, a **Description** saying what it does, an **Icon**, and then pick **What It Is**. That choice decides which of its other settings it has.

- **Carried**: kept in the pack. Its **Stat Changes** are made while it is carried, as a lucky charm's are. With none, it is carried and does nothing else, as a key is.
- **Worn**: worn in a slot, picked in **Slot** (Weapon, Armour, Trinket, or a name of your own). Its Stat Changes last while it is worn. A weapon's **Damage**, **Reach**, **Swing Seconds** and **Knockback** are set here too, for the Attack action (6.17).
- **Used Up**: a potion, food or a scroll. Carrying it changes nothing, but using it takes one out of the pack and makes its **When Used** changes on whoever uses it, for **For Seconds** (they wear off by themselves) or for good. Untick **Leaves The Bag** for something kept and used again, such as a spell book. Saving the scene keeps a potion still wearing off, with the time it had left.
- **Used With Button**: used again and again, filling up as rooms are cleared (**Charge**), and sending its **Use Event** to the machine of whoever uses it, which does what the item does.

Max Stack says how many sit in one pile in an inventory. One keeps each in its own slot, which suits a weapon or a key, and twenty or a hundred suits arrows, coins and potions. Under **Extras** are **Type** (your own word, such as potion, that an inventory can sort by), **Price** for a shop, **Rarity** (how often random loot picks it) and **Unlocked By** (a true-or-false global variable that must be true before random loot offers it, such as one set when a boss is first beaten).

Using and wearing from the screen. An **Inventory Grid** draws the pack on the screen. Its **When Chosen** is **Use Or Wear**. When the player clicks an item (or moves to it and presses the accept button), a potion is drunk, a sword put on, an item used with a button used. Chosen in a grid showing what is worn, a worn thing comes off again. The first thing worn makes the player a **Worn** Inventory to wear it in. Set When Chosen to **Nothing** for a grid whose choosing a template decides, such as a shop that sells what is chosen.

From a template, a potion needs one action, **Use Item**, with the potion picked in it. **Remove Item** is for taking something out of the inventory without using it, such as a key handed over at a door or an item sold in a shop. The count drops, and the stack is gone at zero.

Changes stack fairly. A stat always holds what it started as, plus every addition, times every multiplication, whatever order the items came in, and taking an item away undoes only what it changed. Picking up a sword then a rage charm ends where picking up the rage charm then the sword does.

Action	Does
Add Item	Put something in an inventory, such as loot into the player's pack, stock onto a shop shelf or a key into a chest. It sends one event when it all fitted and another when it did not, so you can say "your pack is full" rather than quietly lose the thing. Leave Inventory empty and it uses the Inventory on the node this machine works on.
Count Item	How many of something an inventory is carrying, into a variable, for an arrow counter or a shop comparing coins with a price. It can also count the slots used or free, for "12 of 20" on a panel.
Drop Item	Put something down in the world where it can be picked up again. This covers dropped loot, a weapon thrown away and what a defeated enemy was carrying. It takes the item out of an inventory and puts a scene you choose into the world with the item inside it. That scene needs an Inventory on it, so picking it up again is just a Move Item from that inventory into the player's, and a dropped stack of twenty arrows is still twenty arrows. Nothing is ever dropped into nowhere. With no scene, or a scene with no Inventory, the item stays where it was and says why.
Drop Loot	Spill loot on the floor, as a beaten enemy, a broken pot or an opened chest does. It drops an item from its Items now and then, rarer ones less often as the rarity chances say, and a handful of gold between Gold Least and Gold Most . What drops lies there with a glow and its name in the colour of how rare it is (white, blue, yellow, orange), a stack it carried as one pile with its count, and a player walking over it picks it up. An item goes into their Inventory, and gold onto the variable named in Gold Goes Into , which is theirs, not this machine's. Varies By gives each item dropped numbers of its own, a little better or worse than the item it was made from, kept when the game is saved. Item Sound and Gold Sound play as they are picked up. With Drops What It Carries ticked, as it starts, everything in an Inventory on the one dropping it drops too, every time. Give a boss an Inventory carrying its crown, and it always drops the crown. Names of loot lying together stack one above the other, each readable.
Equip Item	Wear something out of an inventory, such as a sword into the weapon slot or a helmet onto the head. It takes the item out of the inventory it is carried in, puts it in the slot its Slot names, and makes its stat changes on the character. A sword that adds 2 to damage adds it while it is worn and stops the moment it comes off. Whatever was in that slot already comes off first and goes back in the inventory. If there is no room for it, <i>nothing</i> happens and Cannot Equip is sent, because swapping a sword should never lose the old one.
Give Item	Give an item to a machine, usually the player's. Its changes are made there, it counts as held this run, and it goes into the collection. Remove undoes its changes. Item Event is sent afterwards carrying the item, for a HUD to name it.
Inventory Has check	What an inventory is carrying. Has it at least so many of something, is it full, is it empty? The question a door asks about a key, a shop asks about coins, and a "your pack is full" message asks before a pickup. Leave Node empty to ask the Inventory on the machine's own node. Pick the item, or press Var to read it from a Resource variable such as the one Pick Item chose.

Move Item	Move something from one inventory to another, for emptying a chest, buying from a shop or handing over a quest item. Set How Many to zero to move everything there is. Nothing is ever made or lost. What the other inventory has no room for stays where it was, and the action says how many really moved.
New Run	Start a new run, where every item can turn up again and none is held. The collection keeps everything ever found.
Offer Item	The same choosing, without using the item up. It suits a pedestal the player may walk away from, a shop shelf, or a chest opened later. Give Item is what uses it up, so an item nobody took can come round again later in the run. Use Pick Item instead where every pedestal should show something new whatever the player does.
Pick Item	Choose an item at random from a list, and use it up as it is chosen, so the next pedestal shows something else whether or not the player took this one. Rarer items come up less often, as the four chances say, and with Not Seen This Run an item that has already turned up is not offered again. The item goes into a Resource variable, and None Left Event is sent when there is nothing left to choose. The list is written out a row at a time. Press Add Item and pick an item, your own first and NodeWorthy's own under What came with NodeWorthy .
Remove Item	Take something out of an inventory. Spending a key on a door, paying a price and a thief lifting a purse all need it. If there is not enough it takes <i>nothing at all</i> and says so, because a price half paid with the goods refused is worse than one refused outright.
Show Item	Put an item's picture on a sprite or a picture box and its name and line in labels, and store its price, its charge and whether it is active in variables. With no item it clears them. Pedestals, shop shelves, popups and the active item's slot on a HUD all use it.
Sort Inventory	Put an inventory's stacks in order so a panel draws them tidily, by name, by what sort of thing they are, or with the rarest first. Run it when a panel opens, not every frame.
Unequip Item	Take off what is worn in a slot, back into the inventory, with its changes undone. If there is no room it stays on and says so, rather than being dropped where nobody can reach it. Equip Item takes the old thing off by itself when it wears a new one. This is for putting something away.
Use Item	Use an item. One whose What It Is is Used Up , such as a potion, food or a scroll, is taken out of the inventory (the Inventory on this machine's own node unless Inventory names another) and makes its When Used changes on whoever uses it, for its For Seconds or for good. With none left, None Left Event is sent and nothing is used. The item is picked here, or read from a variable when it changes during the game. An active item, used with a button once its charge variable is full, sends its Use Event to this machine and its charge starts again from 0. If it is not charged yet, Not Ready Event is sent instead. Add one to the charge variable when a room is cleared.

10.33 Quests, for things to do, ticked off as they are done

The collection is kept on its own, after the game is closed and in every run after, and every item given is in it. Show it with an **Item Collection** node (10.35).

Quests. A quest is a file too, made with Create New > NodeWorthy Quest (see 6.16).

Action	Does
Abandon Quest	The player gives a quest up. It shows as abandoned, its steps stop counting, no reward is given, its Abandoned Event is sent, and it can be begun again.
Complete Quest	Mark a quest done now and give its reward, whatever its steps say.
Fail Quest	Fail a quest under way. It shows as failed, gives no reward, and can be begun again.
Quest Is <i>check</i>	A quest is not started, under way, done, failed or abandoned. See 6.16.
Quest Step Is <i>check</i>	One step of a quest, picked by its words, is done, not done, or the one to do now.
Start Quest	Begin a quest. It appears in the Quest List, and its steps tick themselves off as the game is played. One under way or done is left as it is, but a failed one begins again.
Track Quest	Track a quest, or stop tracking it. A quest is tracked as it begins. Tracked quests are marked on the Minimap, and a Quest List set to Only Tracked shows only them.

10.34 Dialogue, for characters talking, and the answers the player picks

Action	Does
Dialogue Line Said <i>check</i>	Has a line of a dialogue been said this game? Kept by Save Game. See 6.15.
Start Dialogue	Play a dialogue made in the dialogue graph in the scene's Dialogue Box , line by line, with the player's answers as buttons, until it ends. The state waits until then. Go on with a transition on FINISHED, or name a Finished Event to send. With more than one Dialogue Box in the game, pick which in Box Path . See 6.15.

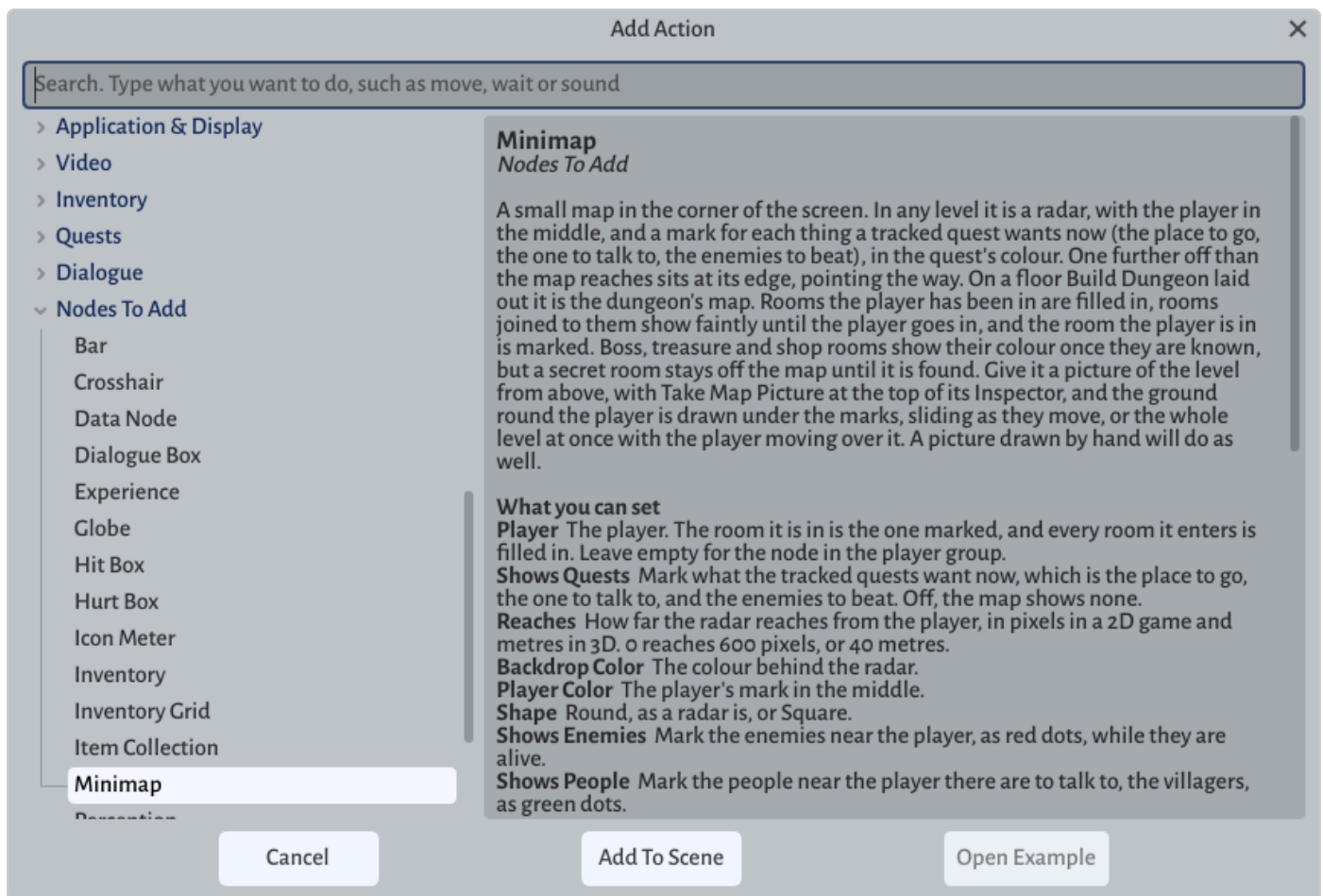
10.35 Inventory, Inventory Grid, Crosshair, Minimap, Item Collection, Icon Meter, Stat Panel, Globe, Bar, Skill Bar, Dialogue Box, Quest List, Quest Log, Hurt Box, Hit Box, Perception, Experience, Data Node and Performance Overlay, the nodes to add

These are nodes rather than actions. **Inventory** is the odd one out, because it draws nothing at all. It is what something is carrying. Put one under the player, a chest, a shop shelf or a guard, fill in **Starts With** in the Inspector, and the item actions work on it. Nothing draws it on the screen until an Inventory Grid does. The Inspector says so, and **Show It On Screen** there adds one. Fill in **Slot Names** (Weapon, Armour, Trinket) and it wears one thing in each slot and holds no stacks, and Equip Item puts things into those slots. A character that both carries and wears things has two of them, and the actions find the right one on their own. Because those are the node's own settings, one chest scene dropped into a level twenty times can hold something different in every copy.

Inventory Grid is what draws an inventory on screen. It is a grid of slots with a count on any stack holding more than one, moved about with the arrows or a gamepad and chosen with the accept button, and with the mouse. At an edge of the grid the arrows move the focus on to whatever is beside it, as they do from Godot's own lists, and the accept button on an empty slot is left for anything else waiting for it, such as a Dialogue Box. What choosing does is set by its **When Chosen. Use Or Wear**, as a new grid is, drinks a potion, puts on a sword (or takes it off again, chosen where it is worn) and uses an item used with a button, with no template at all. Set it to **Nothing** for a grid whose choosing your template decides, such as a shop. It sends **Selection Changed** and **Chosen** carrying the item either way, and an Event On Signal catching one carries the item straight into a variable for Equip Item, Remove Item or Move Item to be pointed at. Added from NodeWorthy, it sits in the bottom left corner of the screen, and a second one just above the first. A worn inventory draws a slot for each of its Slot Names, filled or not, so an empty weapon hand still has a place on screen.

The quickest way to add one of these nodes is from the **Add Action** window, where they are listed under **Nodes To Add** after the actions, and a search for "map" or "hearts" finds them there. **Add To Scene** puts the node where it belongs and selects it, ready to set up in the Inspector. Something drawn on the screen goes under the scene's CanvasLayer (one called HUD is made when the scene has none), **so in the editor you see it in the 2D view** (the **2D** button at the top of the editor), even in a 3D game, and over the game when you press Run. Perception goes under the node the machine drives, and the Performance Overlay under the scene's root. It adds nothing to the state, and one Undo takes it out again. All but the Performance Overlay are also in the Scene dock's right-click menu, with the parts of the screen under **NodeWorthy** → **Add To Screen** on the level, and what a character carries under **NodeWorthy** → **Give It** on the character, where a Hurt Box comes with **Health**. One that would not work under the node you right-clicked is greyed out and says what it needs, and so is a second Perception, Experience or Data Node on a character that has one. And they are in Godot's own **Add Child Node** window, where you can search for the name.

Setting one up. Pick one in the Scene dock and the Inspector shows it, with a line at the top saying what it does for the thing it sits under. A setting that names a variable is a list of that character's own variables of the right kind, with **New variable...** to make one in their template. One that names an event lists the events their template knows. What a Hit Box hurts and what Perception look for are picked in the scene (2.8). A Stat Panel's stats are ticked from the character's numbers, and Experience's gains are rows, like an item's changes. Picking one drawn on the screen, such as a Stat Panel, leaves the 2D view up for you to place it while you set it up.



Nodes To Add, at the end of the Add Action list.

In the Scene dock each of these shows a small script icon beside it, as the **Machine** does. That is how Godot marks any node that comes from an add-on. It is NodeWorthy's own, and there is nothing in it to open or change.

Node	Does
Crosshair	The aiming mark in the middle of a shooter's screen. Put it under a CanvasLayer and it fills the screen and draws itself in the centre. Machines make it say things by setting its properties with Set Property . Spread opens the marks for running and closes them for standing still (they ease there on their own), Spread Scale keeps them tight while aiming without touching the spread, Bloom kicks them open for a shot and closes again at Bloom Recovery , Color turns it red over an enemy, and Hit shows the marks that say a shot landed, for that many seconds. Read Opening with Get Property to know how far open it is drawn.
Minimap	A small map in the bottom right corner. In any level it is a radar with the player in the middle. It marks the enemies near them as red dots, the people to talk to in green, what there is to pick up or open in gold, and the way out as a white square, each with a tick under Marks to leave it off. Right-click it and choose NodeWorthy → Take Map Picture to draw the level under the marks, its ground, walls, trees and buildings, pictured from above. It marks in each quest's colour what the tracked quests want now, such as the place to go, the one to talk to and the enemies to beat (6.16). A quest's mark or the way out further off than Reaches (0 means 600 pixels, or 40 metres in 3D) sits at the edge pointing the way, and an

	enemy, person or item that far off is not marked. Shows Quests turns off only the quests' marks. A picture of the level from above can go under the marks. Press Take Map Picture at the top of its Inspector, and the ground round the player slides under the arrowhead as they walk, leaving out the player, enemies and anything that moves. Under Map Picture, Shows picks the ground round the player or The Whole Level at once, and Turns with Player turns the picture so the way the player faces is always up. Use My Own Picture... puts a map of your own under it, and Shape makes it round or square. On a floor made by Build Dungeon it is the dungeon's map instead, centred on the room the player is in. Rooms the player has been in are filled in, rooms joined to them show faintly, and the boss, treasure, shop and secret rooms carry a coloured mark once found. A secret room stays off the map until it is entered. Under Dungeon Floor, Room Size and Gap set how big each room is drawn.
Item Collection	Every item in a grid for a collection page, with the ones ever found in full colour and the rest as dark shapes still to find. Hover over one to read its name and line. An item is found once it goes into the player's Inventory, or Give Item hands it over. List the items, or name a Folder and every item file in it and the folders under it is shown. Cell Size and Gap set how big each one is drawn. Set Show to Held This Run for a pause menu. It then shows only the items given to the player this run, in the order they were given, with nothing to list.
Icon Meter	A row of icons showing a number, such as hearts for health, souls for lives or shells for ammo. Give it a Full Icon, a Half Icon and an Empty Icon. Per Icon says how much of the number one icon stands for, so with 2 each heart is two halves and 5 out of 6 shows two and a half hearts. Pick the number in Shows, such as health, and it follows the player's by itself (another character's with Character), its Maximum too when the variable has a Max. Or a machine sets Value and Maximum with Set Property. With no icons of its own it draws plain hearts. Per Row wraps a long row.
Stat Panel	A character's stats on the screen, such as Strength 15, Defence 6, Health 24 / 30. It reads them straight from the character's template, with nothing to keep up to date. Add it from the right-click menu and it shows the player's numbers at once, in the editor too. Leave Character empty for the player, and Stats empty for every number the template has that is not one of its settings, or tick the ones to show. Gear shows where it came from, as 15 (10 + 5), and a change flashes green going up and red going down. Health that is Max max_health shows as one line, 24 / 30. It sits on the HUD, so in the editor look for it in the 2D view, even in a 3D game. It keeps up as you add, remove or change the template's variables.
Globe	A glass globe that fills with a number, the way action role-playing games show health and mana. It is red for health and blue for mana, draining smoothly as it goes down, with the number written across it. Pick the number in Shows, such as health, and it follows the player's by itself (another character's with Character), its Maximum too when the variable has a Max. Added from the right-click menu the first goes in the bottom left corner, and a second goes in the bottom right, blue, showing mana. Liquid Color, Diameter and Shows Number are on its page.
Bar	A bar that follows a character's number, with a name over it, such as a boss's health along the top of the screen. Pick the Character in the scene and the number in Shows. Its Max is a full bar. A hit drains it smoothly, with a pale strip showing the bite just taken. With Appears Within set it comes up only once the player is that close, as they walk into the boss's hall,

	and it goes when the character is gone. Empty, it stays on screen, so an empty mana bar is seen to be empty. Tick Over Its Head and it leaves its place on the screen and rides just above its character in the game, in 2D and 3D, which is how an enemy gets a health bar. NodeWorthy → Give It → Health Bar on a character that has health adds a small one with the tick on. Title, Color, Width, Thickness and Shows Number are on its page.
Skill Bar	A row of skills along the bottom of the screen, each with its picture and its key. Press the key, or click the slot, and when the skill is ready and the character has the mana, the mana is paid, the skill's event is sent to the player's machine, and the slot darkens and sweeps back as it comes ready again. What a skill does is up to your template. A state waiting for its event, such as Cleaving on CLEAVE, swings, blasts or heals, as any state does. Add skills in the Inspector with Add Skill. Each has a Skill Name, a Description shown when the pointer rests on it, an Icon, a Key, a Cooldown, a Mana Cost, its Event Name, and Uses Item for a skill that takes an item, such as a potion. The slot then shows how many are carried, and it cannot be used with none. Mana Variable is the number skills are paid from, and Mana Back Per Second fills it again by itself up to its Max. A skill the mana cannot pay for is dimmed.
Dialogue Box	Where a dialogue is shown. It holds who is speaking, their picture, their words a letter at a time, and the player's answers as buttons under them. Added from the right-click menu, it goes along the bottom of the screen, and you can drag it anywhere. Letters Per Second sets how fast the words appear, 0 for all at once, and Portrait Size how big the picture is. It is hidden until Start Dialogue uses it, and it takes its panel, font and buttons from the game's look. See 6.15.
Quest List	The quests under way, and their steps with a tick against each one done, as in <i>Find red mushrooms 2 / 3</i>. It reads them from the game, needs nothing to keep it up to date, and hides while there are none. Show picks Active for only the quests under way, as a tracker in a corner, Active And Done to list those done after them, or Every One for those failed as well, as a journal. Show Descriptions adds each quest's line. It takes its panel and font from the game's look. See 6.16.
Quest Log	The player's quest log, a separate screen opened with a key (Open Key is Q, or Select on a pad) and closed with it again or Escape. It lists the quests down one side and sets out the one picked beside them, with its description, each step ticked with its count and description, and its reward. Tabs show the quests Under Way, those Done, or Every Quest. Track and Stop Tracking choose which quests the Minimap marks, and Abandon (asking twice first, and you can untick Can Abandon to leave it out) gives a quest up. It stops the game while it is open (Pauses The Game), works with the mouse, the keyboard and a gamepad, and never shows a hidden quest. It sends Opened, Closed and Quest Picked for an Event On Signal.
Hurt Box	Makes something able to be hurt. A Hit Box touching it takes its damage, less the defence, off its health, and its machine is sent HURT, or DIED at 0. It is knocked back and safe for a moment, blinking, and flashes in its Flash Color. Shows Damage floats each hit's number up over it. Hurt Sound and Died Sound play when it is hit and when it dies. Give It → Health on the right-click menu adds one with a health and a Dead state. It takes its shape from its character's body. See 6.17.
Hit Box	The part that does the hurting, such as a slime's body, a sword's edge or spikes. It hurts whatever it touches with a Hurt Box in its Hurts Group, never its own character. It hurts

	always, or once per swing. See 6.17.
Perception	What a character notices. That is what it can see and hear, and where it last saw what it watches. The AI actions and checks read it, so range, view cone and memory are set once, here. Adding one of those actions brings one, and Give It → Perception adds one too. See 6.13.
Experience	Experience and levels for the character it is under. Enough XP in its <code>xp</code> variable takes it up a level, its stats grow as Gains Each Level says, health fills up, and every machine is sent LEVEL_UP. It works before any number is changed. See 6.18.
Data Node	Values of one thing's own from a Data Definition, such as this goblin's Health and Damage. Pick its Definition and the fields show at the top of the thing's Inspector, and Starts As starts it from a Data record. Get Data, Set Data and the Data check reach it with From set to Data On A Node. See 6.20.
Performance Overlay	A small panel showing how fast the game runs. It shows frames a second, the 1% low (the slowest frames, which is where stutter shows), the worst frame, how much of each frame the machines take, and how many machines are running. F3 shows and hides it, and so does the backtick key (` , under Escape). On a Mac the F keys work the brightness and the windows unless the keyboard is set otherwise, so F3 alone does not reach the game there. The numbers are gathered either way. Take it out before you ship.

11. Where things live

Everything under `addons/nodeworthy` and `addons/nodeworthy_editor` is NodeWorthy's, and an update replaces it whole. Keep every file of your own somewhere else. Your templates go in `res://nodeworthy/`, and your scenes and scripts in folders you make yourself.

Where a new file goes. Every window that makes a new file, whether an item, a quest, a dialogue, an actor, Data, a template or a level, says where it is saved, as in *Saved in conversations/*, with **Folder...** beside it. Folder... opens Godot's own folder window, where you can also make a new folder. The folder you pick is where the file is made, and the next file of that kind goes there too, in this project, until you pick another. Make It Mine copies into that folder as well, and says where. Folders inside NodeWorthy's own are refused, with the reason, because an update would wipe them. Until you pick a folder, each kind goes in the folder below.

Path	Holds
<code>res://nodeworthy/</code>	Your templates and their states. Change it with the project setting <code>nodeworthy/template_folder</code> , or pick another folder with Folder... when you make a template. A template keeps its states beside it.
<code>res://items/</code> , <code>quests/</code> , <code>conversations/</code> , <code>actors/</code> , <code>data/</code>	Your items, quests, dialogues, actors, and Data Tables and Data Definitions, made from the NodeWorthy panel or a setting that holds one.
<code>res://levels/</code>	Levels made with New Level.
<code>addons/nodeworthy/templates/</code>	The ready-made templates Hello World, Platform Hero (2D), Platform Hero (3D), Side-On Platform Hero (3D), Top-Down Hero (2D), Top-Down Hero (3D), Third Person Hero (3D), AI Enemy, Guard, Chaser, Skittish Villager and Pack Hunter.
<code>addons/nodeworthy/recipes/</code>	Small scenes that each show one thing, with what to press written on screen. See chapter 9.
<code>addons/nodeworthy/games/</code>	Whole games made with NodeWorthy and nothing else. They are the first person shooter (see 8.4), Hollow Spire, a 2D roguelike (8.5), Handbrake Rally, a rally game against the clock (8.6), Swarm Break, an arena shooter (8.7), and Lantern Hop, a platformer (8.8). There are also two smaller worlds to play and take apart. Mossbrook is a village adventure with a shop, quests, someone to talk to

	and a cave (open <code>mossbrook/village.tscn</code>), and Emberdeep is a dungeon seen at an angle, where you click to walk and fight, gather loot and face the Bone King (open <code>emberdeep/levels/title.tscn</code>).
<code>addons/nodeworthy/NodeWorthy_Manual.pdf</code>	This manual.
<code>addons/nodeworthy/extensions/</code>	NodeWorthy's engine, built for Mac and Windows. It holds the Machine and every action and check that comes with it. There is nothing in it to open or change. Actions and checks you write go in a folder of your own (see chapter 13).
<code>addons/nodeworthy_editor/</code>	The NodeWorthy panel itself, with the graph, the tabs and the boxes in them. A game does not need it to run, so it can be left out of an export.

12. Glossary

Plain definitions of the words this manual uses. Words in **bold** inside a definition have their own entry. NodeWorthy's names are written exactly as they appear in the editor.

Action

One job a state does while it is the current state, such as waiting, moving, playing a sound or setting a property. See 3.2 and 6.3. Every action is listed in chapter 10.

AnimatableBody2D / AnimatableBody3D

A solid physics body that you move, such as a moving platform. It carries whatever stands on it. See 2.6.

AnimationPlayer

A Godot node that plays animations made by hand in the Animation tab. See 6.12.

AnimationTree

A Godot node that blends a character's animations together. A machine tells it what is happening with **Travel To** and **Set Tree Parameter**. See 6.12.

Area2D / Area3D

An invisible zone that things pass through. It notices what comes in and goes out. A trigger, a pickup's reach and a pressure plate are all made this way. See 2.6.

Argument

A value a signal hands over when it fires, such as the body that came in or a slider's new value. See 2.9.

Array

A kind of variable (Godot's Array) that holds several values in order. See 6.5.

Autostart

A Machine setting, on by default. When it is on, the machine starts by itself. When it is off, it waits until something starts it. See 6.1.

Body

A node that takes part in physics. It is a StaticBody, RigidBody, CharacterBody or AnimatableBody. See 2.6.

Breadcrumb

The trail of state names above the graph that shows which level of a template you are looking at. Click a name, or **Up**, to go back out. See 4.4.

Bus

A channel of Godot's sound mixer, in the Audio tab. **Master** is the main one, and NodeWorthy adds **Music**, **Effects** and **Voice** when a game uses them. Volume sliders in an Options menu set buses. See 10.15.

CanvasLayer

A Godot node that draws what is under it on top of the game, not moving with the camera. Scores and menus go under one. See 2.10.

CharacterBody2D / CharacterBody3D

A physics body for a walking character. It only moves when something walks it, which in NodeWorthy is **Move Character**. See 2.6 and 10.12.

Check

An action that asks a question, such as "is the player within 4 metres?", and sends one event for yes and another for no. See 6.6 and 10.2.

Child, parent

A node placed under another node in the Scene dock is its child, and the node above is its parent. A child moves with its parent. See 2.2.

Collision layer and mask

Which layers a body or area stands on (its layer) and which it bumps into or notices (its mask). NodeWorthy names them World, Player, Enemies and Pickups and shows them by name. Collision Layer, on the right-click menu, puts something on one. See 2.6.

CollisionShape2D / CollisionShape3D

The node under a body or area that gives it a size and shape for physics. See 2.6.

Control

Any Godot node for screens and menus, such as a Label, a Button, a slider or a picture. See 2.10.

Custom data

Names and values the maker of a tile set gives its tiles, such as `surface` set to "ice". See 10.22.

Debug

The list in the NodeWorthy toolbar, while the game runs, for picking a machine to watch. See 7.1.

Dictionary

A kind of variable (Godot's Dictionary) that keeps a value for each of several names, such as an inventory. See 6.5.

Dock

One of the panels of the Godot editor, such as the Scene dock, FileSystem dock or Inspector. See 2.1.

Editable Children

A right-click choice on a copy of a scene in the Scene dock that lets you see and change the nodes inside it. See 2.2.

End State

A state inside another state, marked to say "reaching this one means the state holding it has finished its work". See 3.5 and 6.8.

Event

A named message, such as `Opened` or `FINISHED`. It moves nothing by itself, but a transition waiting for it does. See 3.4 and 6.4.

Events tab

The tab beside Actions, Transitions and Variables where events are written down, described, given a value to carry, or made global. See 6.7.

Every Frame

A **Run Mode** that keeps an action working every frame for as long as its state lasts. A state holding one never sends `FINISHED`. See 6.3.

FileSystem dock

The Godot panel listing the files in your project. See 2.1.

FINISHED

The event a state sends itself when all of its actions are done. A transition on `FINISHED` means "when this state is finished, move on". See 3.4 and 6.4.

Frame

One redraw of the screen. A game draws many a second. See 2.6 and 6.11.

Global event

An event heard by every machine in the game, not only the one that sent it. See 6.7.

Global variable

A variable with **Is Global** ticked. It is one value shared by the whole game, kept when the scene changes. See 6.5.

Godot view

The editor with everything Godot has, the NodeWorthy panel as a tab in the bottom panel. The button at the top right of the editor switches to it and back to the NodeWorthy view. See 2.1.

Group (Godot)

A label given to nodes, such as `enemies`, that lets an action work on all of them. NodeWorthy sets them up when you pick who a setting is about. See 2.8.

Group (graph)

A note on the NodeWorthy graph with states dragged onto it, so they move together. Nothing to do with Godot's groups. See 6.10.

Impulse

A push given all at once, as opposed to a Force. A kind of **Apply Force**. See 10.11.

Input Map

The page under Project Settings where named input actions, such as *jump*, are given keys and buttons. NodeWorthy works without it. See 2.7.

Inspector

The Godot panel that shows the settings (properties) of whatever is selected. See 2.3.

Machine

The NodeWorthy node, **SCMachine** in the Add Node window, that runs a template on the node it sits under. See 4.2 and 6.1.

Marker2D / Marker3D

An empty node that marks a spot, such as a muzzle or a patrol point. See 2.10.

Navigation mesh (navmesh)

The walkable area of a level, which characters use to find their way round walls. Adding Navigate To makes it, and NodeWorthy → Bake Navigation Mesh bakes it again. See 10.21.

Node

One building block of a Godot game, with one job. Scenes are made of nodes. See 2.2. On an action, **Node** is the setting that says which node it works on. Left empty, it works on **Self** (6.3).

Node dock

The Godot panel beside the Inspector listing a node's signals and groups. See 2.1.

Normal

The direction a surface faces. **Raycast** can store it, so a mark can be stood on a wall. See 10.11.

Once

A **Run Mode** that does an action's work the moment the state is entered, and no more. See 6.3.

Output panel

The tab in the editor's bottom panel where messages and warnings appear. See 2.1.

Parallel States

Several sets of states inside one state that run side by side, each at its own pace. See 3.6 and 6.9.

Perception

The NodeWorthy node, **SCPerception** in the Add Node window, that gives a character sight, hearing, memory and a target. See 6.13.

Physics step

The fixed beat on which Godot's physics works, separate from frames. See 6.11.

Plugin

An add-on to the Godot editor, turned on under Project Settings, Plugins. NodeWorthy is one. See 4.1.

Problems button

The button in the NodeWorthy toolbar that shows a tick, or lists what is wrong with a template. See 7.2.

Process Mode

A Godot setting on every node that decides whether it runs while the game is paused, or at all. See 2.3.

Property

One setting of a node as shown in the Inspector, such as its position, its text or its colour. See 2.3.

Raycast

Looking along a straight line to find what is there. See 10.11.

res://

How Godot writes "the project folder" at the start of a file's path. See 2.2.

RigidBody2D / RigidBody3D

A physics body that Godot's physics moves, so it falls, bounces and can be pushed. See 2.6 and 8.1.

Run Mode

The setting on an action that chooses **Once** or **Every Frame**. See 6.3.

Scene

A group of nodes saved as one `.tscn` file. A level, a player, a bullet and a menu are all scenes. See 2.2.

Scene dock

The Godot panel showing the nodes of the open scene as a tree. See 2.1.

Self

The node a Machine works on. It is the node the Machine sits under, unless its **Target** says otherwise. See 6.1 and 6.3.

Signal

A node's announcement that something has happened to it, such as a button's `pressed`. See 2.9. A transition can wait for one directly, because the **Signals** submenu of its **When** list offers the ones the machine's node sends, such as `body_entered`. See 6.4.

Start state

The state a machine, or a state holding sub states, begins in. Its title bar is green. Set it with **Set as Start**. See 3.1.

Starting Values

A machine's own starting value for a template's variables, set in the Variables tab with the Machine picked in the Scene dock, or at the bottom of its Inspector. See 6.1 and 6.5.

State

A situation a machine can be in, such as Closed or Open. Only one runs at a time, except where parallel states run side by side. See 3.1 and 6.2.

NodeWorthy view

How the NodeWorthy app opens your games, with the NodeWorthy panel in a tab of its own beside 2D, 3D, Script and Game. See 2.1 and 4.1.

StaticBody2D / StaticBody3D

A solid physics body that never moves, such as a floor. See 2.6.

Sub States

States inside a state that take turns, one at a time. See 3.5 and 6.8.

Target

A Machine setting naming the node its actions work on. Empty means the node the Machine sits under. See 6.1. (A Perception node also has a target, meaning the one thing a character is dealing with. See 6.13.)

Template

A file of states, actions, transitions, variables and events, drawn in the NodeWorthy panel. Many machines can run one template. See 4.2.

Transform

A node's position, rotation and scale, under the Transform heading in the Inspector. See 2.3.

Transition

A way out of a state, taken when a named event arrives. See 3.3 and 6.4.

Tween

Easing a value smoothly from one number to another over time, as **Tween Property** does. See 10.10.

Update Interval

The setting on an action for how many seconds pass between repeats. See 6.3.

Var

The button beside a setting that makes it read a variable while the game runs instead of a typed value. See 6.5.

Variable

A named value a machine keeps, such as a score, a count, a flag or a node. See 6.5.

Wait For Previous

A setting on an action that makes it wait until the action above has finished, for doing things one after another. See 6.3.

13. Writing your own action, an optional extra for programmers only

You do not need this chapter. Every game in this manual, including the whole first person shooter, is made without a line of code. This chapter is for a team that has a programmer and wants to add an action or a check of its own, or to talk to a machine from a script. Skip it if that is not you.

Everything here is written in GDScript. NodeWorthy's own core is compiled C++, and your scripts run on top of it, but writing C++ of your own against that core is not supported.

13.1 Talking to a machine from a script

A team with a programmer can send an event from a script with

```
SCMachine.machine_of(self).send_event("&0opened") . Nobody needs to.
```

A Machine node offers `send_event(name, data)`, `go_to("StateName")`, `get_value(name)`, `set_value(name, value)`, `start()`, `stop()`, and the signals `state_changed`, `event_sent` and `state_finished`.

A machine's variables are read and written with `machine.get_value("&score")`, `machine.set_value("&score", 3)` and `machine.has_value("&score")`.

13.2 An action is one small script

An action is one small script. The quickest way to start one is to right-click a folder of your own in the FileSystem dock and choose **Create New > NodeWorthy Action...** (or **NodeWorthy Check...**). Give it a name and NodeWorthy writes the script, ready to fill in, and opens it. It is in the Add Action list straight away.

Behind the scenes that folder is added to the project setting `nodeworthy/action_folders`, the list of folders of your own that actions and checks are looked for in. A script written by hand works the same way once its folder is in that setting.

Never put a script of your own in `addons/nodeworthy`. It would be lost at the next update. The Create New menu refuses those folders for exactly that reason.

```
@tool
class_name OpenDoorAction
extends SCMachineAction
## Slide the door open.
```

```
## How far it slides, in metres.
@export var distance: float = 2.0

func _enter(m: SCMachine, index: int) -> void:
    var door: Node3D = m.target as Node3D
    door.position.x += distance
    m.finish(index)

func _summary() -> String:
    return "Open door %s m" % number(distance)
```

- `@tool` is required, so the editor can make a real object from your script.
- Your own class names need no prefix. The SC in front of NodeWorthy's is there so they stay clear of yours.
- Every `@export` becomes a field in the panel, and the `##` line above it becomes its hover tooltip. Write one for every setting in plain English. Say what it does and when you would change it, with the unit, and what leaving it empty means. Someone who has never opened the code should be able to work your action out from the panel alone.
- The parts your action fills in begin with an underscore, as a node's `_ready` does. `_enter` runs once when the state starts, `_update` on the action's interval, `_exit` when the state is left however it is left.
- Once your action has called `m.finish(index)`, `_update` goes on being called only while `_works_after_finishing()` says so. By default that means the action is set to Every Frame. An action that hides Run Mode, by returning false from `_uses_run_mode()`, is taken to keep working, the way steering a body carries on after its state has finished. Return false from `_works_after_finishing()` when your `_update` only waits for something to end. A machine whose running actions have nothing left to update is not called at all, so it costs nothing while it waits.
- `m.finish(index)` reports the action done, and FINISHED and Wait For Previous both go by this. It works from `_enter` and `_update`, and from a signal your action connected. In a state with sub states or parallel states, connect the signal to `m.finisher(index)` instead, which you ask for in `_enter`. It always finishes the action in the right state.
- `m.slot(index)` is this action's private scratch record on this machine. Use it for anything that changes as it runs, because one action object is shared by every machine running the state.
- `_summary()` is the line shown in the panel. It is worth writing, because it is what makes a template readable.

If you write `_validate_property`, to hide or show a setting depending on another. Start it with `validate_setting(property)`, which applies NodeWorthy's own rules to your

settings. With it, a setting that holds any type shows the field for what it holds, and a row of physics layers carries the 2D layer names on a 2D object.

```
func _validate_property(property: Dictionary) -> void:
    validate_setting(property)
    # your own hiding here
```

13.3 A question for a Check

A question for a Check is the same shape with one part to fill in, `_check`, and goes in the same folders as your actions.

```
@tool
class_name DoorLockedCondition
extends SCMachineCondition
## True while the door is locked.

func _check(m: SCMachine, _index: int) -> bool:
    return bool(m.target.get(&"locked"))

func _summary() -> String:
    return "Door is locked"
```

13.4 Changing how an action that ships behaves

Never edit an action that ships, because the change would be lost at the next update. Extend it instead, in a folder of your own, and replace only the part you want. Everything you leave alone still comes from the original. To use the original's version of the part you replaced, call it by the same name with `_default` on the end and no underscore in front. That gives `summary_default()` for `_summary`, `enter_default(m, index)` for `_enter`, and so on for each part.

```
@tool
class_name LongWaitAction
extends SCWaitAction
## Wait, and say so loudly in the Actions tab.

func _summary() -> String:
    return "LONG " + summary_default()
```

The new action appears in the Add Action list beside the original, with all of the original's settings. GDScript has no overloading, a method with the same name but different settings, so replace a method with the same name and the same settings as the original's.