

THE UNIVERSITY OF WARWICK

WARWICK GAME DESIGN SOCIETY

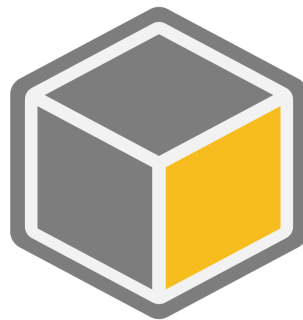
A GODOT BASICS ARTICLE

Designing Flappy Bird in Godot for Beginners

Author
Thomas J. WALSH

Edited by
The Game Design
EXECUTIVE TEAM

September 3, 2026



1 Origins

The Engine The Godot Game Engine was released to the public on the 14th of January 2014, with the ambitious aim “to become the most amazing game engine the world has ever seen!”⁽²⁾ Godot is MIT-licensed; you receive permanent ownership, editing and redistribution rights when you download the engine.

The Game Flappy Bird was developed in under a week by Dong Nguyen, a solo indie developer. By 2014, he was earning over \$50,000 every day in advertisement revenue. On the 9th of February, just five days before the first public release of Godot, Flappy Bird was taken down. The developer cited personal fears over the game’s addictive nature and increased stress over intense media attention.⁽¹⁾ A 2024 reboot was launched to critical failure⁽⁴⁾⁽³⁾ – can you do better than the remake?

Installation There are two main ways to install Godot (with a litany of others including package managers, compilation from source etc.):

- *Steam* (recommended): convenient, fast and easy to update – what more is there to want?
- *Godot’s Website*: useful when Steam’s automatic updates cause problems in big projects.

While you wait for Godot to install, give yourself a reminder of Flappy Bird’s mechanics. There are a few variants of the game out there, play as many as you can find. Ask yourself some questions as you play:

- Why is the game fun / boring?
- What makes each variant better or worse?
- How does the bird move?

Forwards These questions will be key to designing your own game in the future. Rephrased, they might sound like:

- Why is your game fun / boring?
- What makes an iteration of your game better or worse?
- Does the core mechanic feel good to use?

Feedback There will be problems in this document; bad wording, poor explanations, moments of syntactic ineptitude (whether in English or GDScript). Please point them out to us in person, on the discord server, or on the website.

2 Godot Orientation

Projects Godot opens to the project manager which lists your projects and provides project creation / import functionality. You may not have created any projects for the manager to list, so you need a new one to start developing. The project creation window [fig.1] is a form through which you tell Godot some general information about what it should expect from your game.

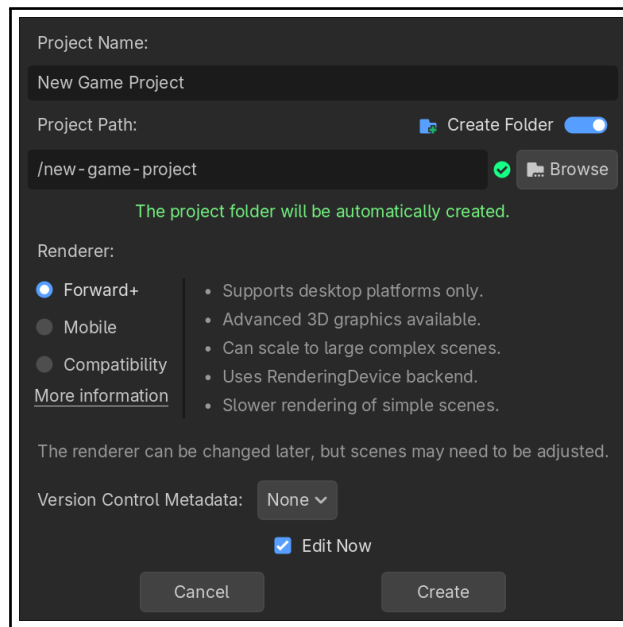


Figure 1: The project creation window

Name the project as you wish and place the project in an empty path (folder) of your choice. I have set the renderer option to ‘Forward+’ mode because my computer is reasonably powerful. Compatibility mode can provide a much better experience on some computers – see here for more detail. If you have experience in Git you can select it in the version control metadata drop-down; Godot will provide you with a preconfigured gitignore file. Make sure that ‘Edit Now’ is ticked and, when you’re ready, hit create.

The Editor Welcome to Godot's editor [fig.2]! Leave everything as it is for the time being. You will spend much of your time here during the course of making any game in Godot, so it's worth taking a moment to get familiar. The editor's centrepiece initialises to the 3D viewport. For now, switch over to the 2D viewport with the panel at the top of the screen.

Note for macOS. If you're using macOS, you may not see the [Scene, Project, Debug, Editor, Help] buttons at the very top left. They are hiding in the native Apple menu bar.

Everything should look fairly empty at the moment. Just a reminder: if at any point you get lost (e.g., this image is not what you're seeing), feel free to ask anyone for help. With that, we can start building flappy bird!

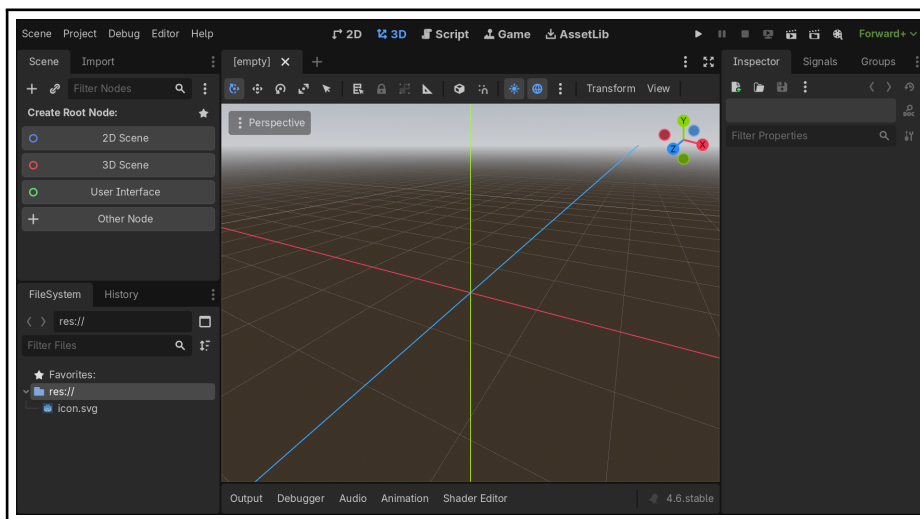


Figure 2: Godot's editor

3 Nodes and Scenes

Definitions In Godot, games are composed of ‘nodes.’ A node might constitute an entire level in your platformer, an enemy in your RPG, or a card in your Balatro clone. Nodes can contain other nodes; the player character in a 2D Minecraft clone might have the subnodes:

- Sprite2D (displays an image)
- Inventory (container of items)
- CollisionShape2D (hitbox for collision)

Scenes exist in the editor to enable structuring and blueprinting. Scenes can be ‘instantiated’ (cloned into different contexts) so that game logic can be reused when necessary. For example, a level in a racing game would need to be instantiated every time the player wanted to drive that track. Instantiation is vital in many circumstances. For example, the Minecraft clone may have the structure:

- Main (contains, manages: the entire game)
 - Mobs (contains, manages: player, enemies and neutrals)
 - Terrain (contains, manages: all blocks)

The ‘mobs’ subnode may itself have subnodes, corresponding to skeletons, zombies and others, each of which will need to be instantiated. These have their own subnodes, as discussed above. Every Godot game is simply a node tree of some sort.

The Bird With that out of the way, we can return to the editor and start with creating the main character of flappy bird, the bird!

A scene needs to be created for the bird so we can instantiate it elsewhere. The scene tree panel in the top-left of the editor re-

quires that we “create [a] root node.” A `CharacterBody2D` node will give the bird some properties we want, including gravity, velocity and physics interactions. Select ‘other node’ and search for this name. For confirmation that everything is working correctly, see [fig.3]. Hit create.

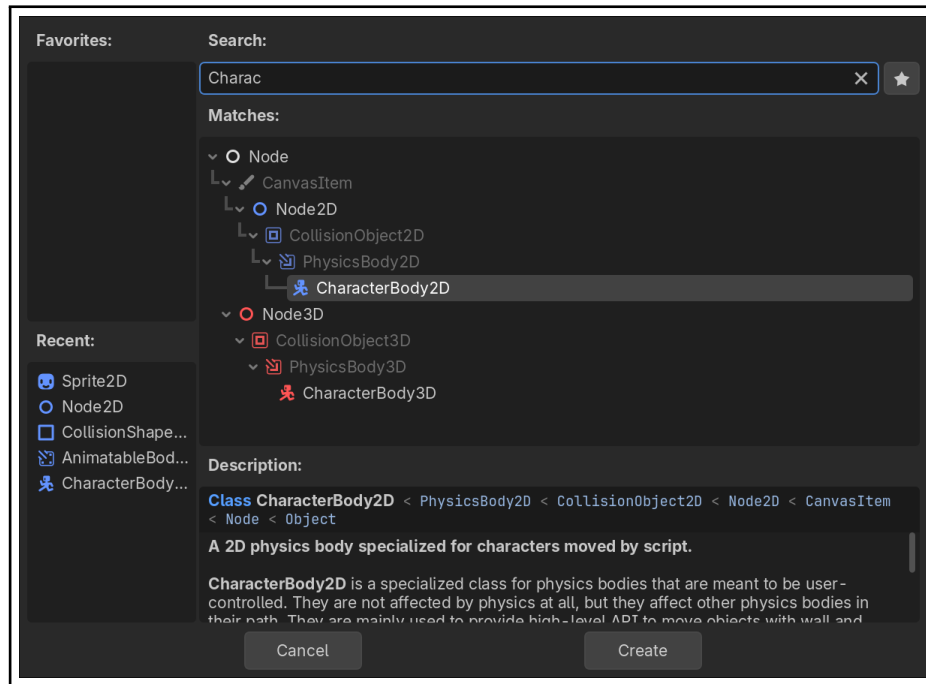


Figure 3: Selecting the `CharacterBody2D`

Possibly the first error of your Godot career has popped-up next to the `CharacterBody2D` node. Hover over the yellow warning symbol to the right of the node to take a look at what Godot is complaining about.

CharacterBody2D Warning. This node has no shape, so it can’t collide or interact with other objects. Consider adding a `CollisionShape2D` or `CollisionPolygon2D` as a child to define its shape.

Read it and weep! The joys of game design have only just begun. Fortunately, Godot is being rather transparent about what the problem is in this situation: the `CharacterBody2D` wants to know the size of its collision box, and you haven’t even

given it one. Right-click on the node and hit ‘add child node.’ Search for a CollisionShape2D, and add it. Unfortunately, like a slightly irritating game of whack-a-mole (can whack-a-mole be not-irritating?), a new warning has appeared next to the new node.

CollisionShape2D Warning. A shape must be provided for CollisionShape2D to function. Please create a shape resource for it!

This error description is a little less verbose, but the underlying problem is easy enough to solve. Select the CollisionShape2D. The node property inspector [fig.4], on the right side of the editor, is now telling us about the internals of the CollisionShape2D node. Unsurprisingly, the ‘shape’ property is empty.

Click on ‘<empty>’ (not the folder-lightning symbol, that does something else), and set the shape to whichever one of [circle, rectangle, capsule] you think best represents a bird. If you change your mind, hit the anti-clockwise ‘recycle’ button which pops up once the property has been set.

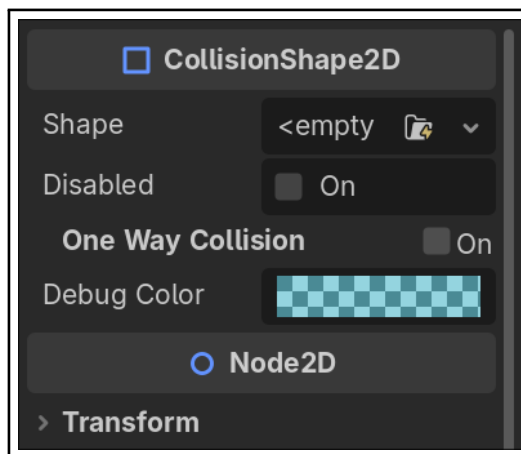


Figure 4: The collision shape property tab

The central editor window should show a tiny blue shape. For now, so that the bird is visible in-game without a sprite (image),

open the ‘debug’ menu at the top-left, and toggle on ‘visible collision shapes.’ Save the scene with ctrl+s (cmd+s for macOS) and name it bird.tscn or something similar. Run the current scene with the ‘clapperboard’ button at the top-right of the editor. A popup window should appear with a grey screen, except for a little blue shape in the top-left corner. That’s the bird!

There is still much to implement for the game to even remotely resemble flappy bird:

- Gravity
- Obstacles
- Movement
- The eponymous flapping

4 Character Controller

Definition What is a character controller? A character controller takes input from the user and translates it into action in the game. Flappy bird has exactly one input: flap. When the player presses flap the bird ‘jumps’ in midair; its vertical speed is set to some value. Intricate character controllers would need an elaborate graph, but we simply require [fig.5].

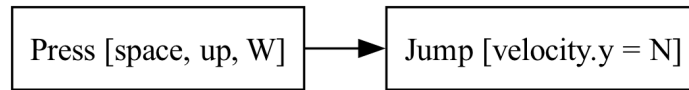


Figure 5: A flapping flowchart

Application To realise this flowchart, you’ll need some code. In Godot, a script can be attached to a node so that it can take actions based on inputs (from the player or other scripts). In our instance, we want to attach a script to the CharacterBody2D node. Right-click the node in the scene tree and hit ‘attach script’ in the menu. Ensure that the ‘template’ box is checked and the drop-down set to ‘CharacterBody2D: Basic Movement.’ You should see code as in [lst.3].

```
1 extends CharacterBody2D
2
3
4 const SPEED = 300.0
5 const JUMP_VELOCITY = -400.0
6
7
8 func _physics_process(delta: float) -> void:
9     # Add the gravity.
10    if not is_on_floor():
11        velocity += get_gravity() * delta
12
13    # Handle jump.
14    if Input.is_action_just_pressed("ui_accept")
```

```

15         and is_on_floor():
16             velocity.y = JUMP_VELOCITY
17
18         # Get the input direction and handle
19         # the movement/deceleration.
20         # As good practice, you should replace
21         # UI actions with custom gameplay actions.
22         var direction := Input.get_axis("ui_left", "ui_right")
23         if direction:
24             velocity.x = direction * SPEED
25         else:
26             velocity.x = move_toward(velocity.x, 0, SPEED)
27
28     move_and_slide()

```

Listing 1: The template script

As you might note once you’ve run the game a few times, the controls don’t exactly match the original’s. You can’t flap, you can move side-to-side relative to the camera with the arrow keys, and the bird drops rather quickly. It’s worth analysing the code line-by-line. Note that all code in `_physics_process` is executed sixty times a second.

The first few lines of code are acceptable. Line ⑩ asks if the bird *is not* on the floor. If this statement evaluates to true, then the indented line ⑪ is carried out, applying gravity to the bird. These lines are acceptable because the only places we might expect the bird to ‘land’ are the ground or a pipe, which would end the game anyway.

The next lines present the first of our gameplay difficulties. The ‘jump’ requires both that `ui_accept` (spacebar) has just been pressed ⑭, and that the bird was colliding with the floor at the same time ⑮. This is the expected behaviour of a platformer, but not of flappy bird. The bird should be able to ‘jump’ at any time. To solve this, simply remove the requirement that the `CharacterBody2D` `is_on_floor`.

To disallow sideways movement, simply remove all code between lines ⑮ and ⑳ inclusive. These lines apply horizontal velocity based on `ui_left` and `ui_right`, left- and right-arrow.

```
1 # ---- // SNIP // ---- #
2
3 func _physics_process(delta: float) -> void:
4     # Add the gravity.
5     if not is_on_floor():
6         velocity += get_gravity() * delta
7
8     # Handle jump.
9     if Input.is_action_just_pressed("ui_accept"):
10         velocity.y = JUMP_VELOCITY
11
12     move_and_slide()
```

Listing 2: Inputs to suit flappy bird

An aside on ‘delta’. Delta is how much time has passed since `_physics_process` was last called. On low-spec computers or in complex games, the program may not be able to execute at sixty frames-per-second. To account for this, delta increases and reduces as such: $delta = \frac{1}{FPS}$. Then, as in ⑥, delta can be multiplied by variables which contribute to movement to.

Parameters There are two parameters which define the movement of the bird. They are Default Gravity and JUMP_VELOCITY. Adjust jumping in your code, and the gravity parameter in Project → Project Settings → search ‘gravity.’ Test how these affect each other, optimise them as you wish for fun gameplay!

5 Obstacles

The Aim The obstacles in flappy bird move across the screen right-to-left. They are composed of two vertical strips (the pipes) with a gap in-between. The game ends if the bird hits either collision box. When the bird passes through the gap successfully, the player gains a point.

Setting an aim. It is often very helpful to set strict aims when designing subsystems in games. The above paragraph is a good example.

Obstacles need to be instantiated (spawned) when they enter the screen and freed (despawned) when they leave. Thus, we need a new scene. Hit the plus icon on the top tab bar next to the bird scene and switch back to the 2D view.

The Pipes The most appropriate node for this purpose is the ‘Animatable-Body2D,’ because it is not affected by gravity or external collisions. Add it as the root node of the new scene. Ensure that it’s ‘Sync to Physics’ property is switched *off*.

CharacterBody2D Warning. This node has no shape, so it can’t collide or interact with other objects. Consider adding a CollisionShape2D or CollisionPolygon2D as a child to define its shape.

The warning returns! It’s up to you now. Consider this a miniature test. Don’t worry if you can’t work it out, my work is in the figures below. Hints: multiple CollisionShape2Ds can be attached to one Body2D. They can be moved in the transform tab in the properties inspector, [fig.4].

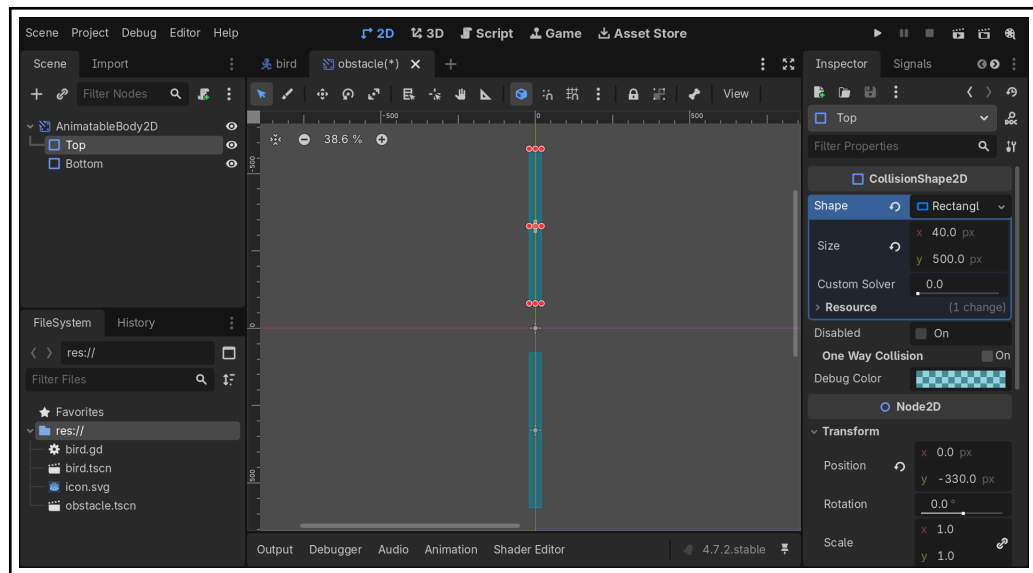


Figure 6: The ‘top’ CollisionShape2D

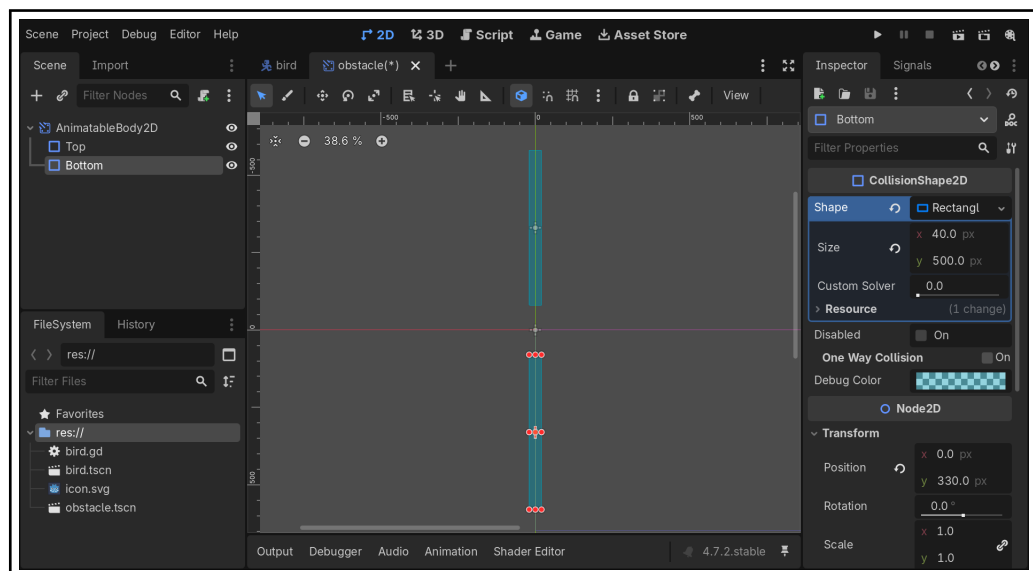


Figure 7: The ‘bottom’ CollisionShape2D

Copy-pasting nodes. When you copy a node and paste it elsewhere, nodal properties are conveniently carried across. However, if you subsequently modify resources within the new node (e.g., change the size of a Shape2D), any change will be replicated in the original because they share this resource. Make the resource ‘unique’ to the new node by hitting the toolbar icon (top-left of the property inspector) → Make Sub-Resources Unique. Now the nodes won’t affect each other.

Note that the top and bottom CollisionShape2Ds are placed relative to the AnimatableBody2D (which is at 0, 0). This means that obstacles can be spawned anywhere to challenge the player.

6 The Main Scene

Bird Placement Create a new scene with a simple Node2D as the root. The viewport is the purple rectangle. Drag and drop the ‘bird.tscn’ file from the FileSystem, [fig.8]. The bird should be placed at the center-left of the viewport to give the player some time to react to the obstacles.

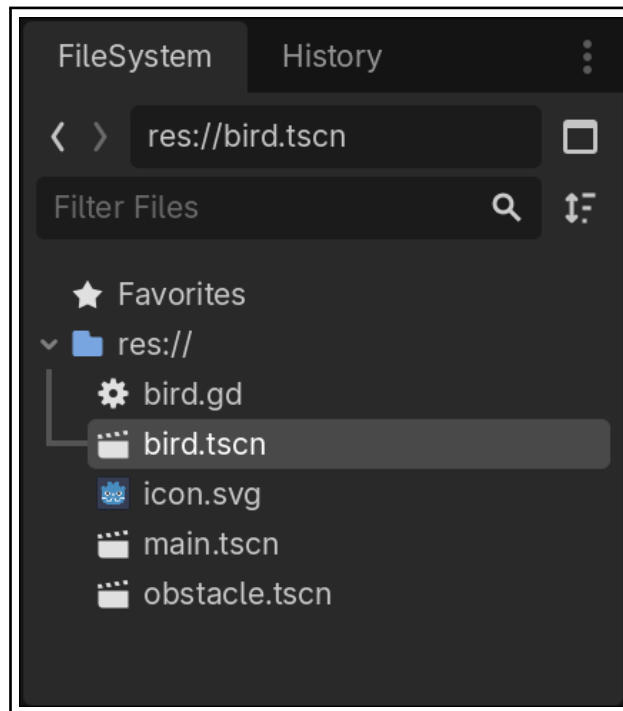


Figure 8: See bird.tscn in the FileSystem

Add the obstacles to the main scene, ... and nothing moves. But what should move? This is an interesting question; think about it for yourself.

Movement The bird seems obvious, right? If you thought so, don't worry, you've fallen for a classic game design trap. If the player and camera move rightwards, and the obstacles are static, we need to write movement code for both. Instead, we can just move the obstacles left. Thus, like a conveyor belt across the screen, we add and remove them at both ends. See [fig.9] for a more detailed plan. Always keep in mind that an intuitive real-world answer may not provide a better programming solution.

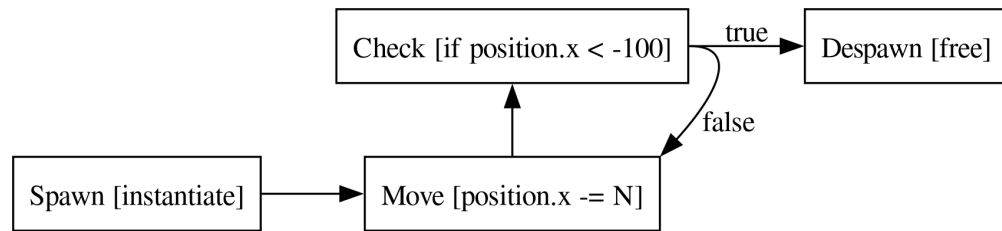


Figure 9: A logic flowchart for obstacle behaviour

We need just two more scripts for a fully functional game! The first is the actual obstacle movement. It will be instantiated by the main scene; it only needs to move and delete itself once off-screen. Return to the obstacle scene in the top tab bar and attach a new script to the `AnimatableBody2D`.

Much like the bird script, our code should go in the function we used before: `_physics_process`. If you are confident enough, try it yourself. Investigate the `queue_free` and `move_and_collide` functions for advice on implementation details.

Find my version below.

```

1 extends AnimatableBody2D
2
3
4 const SPEED = 100
5
6
7 func _physics_process(delta: float) -> void:
8     if position.x < -100:
9         print("Obstacle despawned.")
10        queue_free()
11
12    move_and_collide(Vector2(-SPEED, 0) * delta)

```

Listing 3: The template script

Line ⑧ does the check as per the flowchart. Lines ⑨ and ⑩ are indented and thus carried out if the query returns true. Line ⑨ prints a message to the console, which pops up below the editor viewport when you run the game. ⑪ runs every physics tick and tells the AnimatableBody2D to move_and_collide at SPEED. Drag and drop a couple of obstacles into the main scene. The game should work!

To Infinity Unfortunately the game only works for a little while; you eventually run out of obstacles. Also, you can't really lose. Since we've just fixed up obstacle movement, we can sort out obstacle generation. Delete the obstacles you put down as placeholders.

Every obstacle should be spawned on the right and should start entirely off-screen. This means we need one final script to place the obstacles appropriately. Return to the main scene and create a new script on the root node.

To instantiate an obstacle, the main script must contain a reference to the obstacle scene. Godot has a neat trick to automate this. Drag obstacle.tscn (in the bottom-left FileSystem) into the template script on line three. Before you drop it, hold down

ctrl+shift (cmd+shift on macOS). Godot does the rest!

```
1 extends Node2D
2
3 const OBSTACLE = preload("res://obstacle.tscn")
```

Listing 4: Preloading the obstacle

Preload takes in a file path and provides ‘OBSTACLE’ with a reference which can be accessed at any point later. The ‘const’ keyword tells Godot that this variable will never change.

‘Preloading’ vs. ‘loading’. A note to more experienced programmers: Godot’s preload function is executed at compile time. The load function is executed at runtime. This is a typical storage-speed trade-off.

There is no need to do anything in the `_ready` function (for now). This time, we have a use for process. Every few seconds, a new obstacle should spawn. Consider how you might go about this. Read through the documentation on instantiation and children for more details. If you feel confident, try it out yourself. My code is below.

```

1  const OBSTACLE = preload("res://obstacle.tscn")
2
3  var await_next_obstacle: float = 0.0
4
5  func _process(delta: float) -> void:
6      await_next_obstacle -= delta
7
8      if await_next_obstacle < 0.0:
9          var next_obstacle: Node2D = OBSTACLE.instantiate()
10         next_obstacle.position.x = 1300
11         next_obstacle.position.y = 324
12         add_child(next_obstacle)
13
14         await_next_obstacle = 4.0

```

Listing 5: Preloading the obstacle

The keystone of this code is `await_next_obstacle`, a timer for introducing obstacles. The ‘var’ keyword tells Godot to expect that this property will be changed. This variable is set once at the start of the game at 0.0, line ③. Then, every game-tick (`_process` execution ⑤) `await_next_obstacle` is reduced by the time since the last game-tick ⑥. If it is less than zero ⑧, the obstacle is setup ⑨, placed ⑩ ⑪ and added to the scene ⑫. The timer is reset to 4.0 seconds.

Mess around with the variables and properties across all three scripts and nodes. The numbers that I have assigned result in plodding gameplay – find ones that provide the kind of game you want. There are just two mechanics to go.

Randomness As it stands, the game is a little monotonous. The vertical position of the obstacles ought to vary randomly over time. Random obstacle placement can be achieved with Godot’s `RandomNumberGenerator`. Check out the documentation and see if you can manage it yourself. Below, as always, is my code.

```

1 var rng = RandomNumberGenerator.new()
2
3 func _ready() -> void:
4     rng.randomize()
5
6 func _process(delta: float) -> void:
7     await_next_obstacle -= delta
8
9     if await_next_obstacle < 0.0:
10        # ---- // SNIP // ---- #
11
12        next_obstacle.position.y = 324
13                                + rng.randf_range(-128, 128)
14
15        # ---- // SNIP // ---- #

```

Listing 6: Preloading the obstacle

There are three additions here. First is line ①, which tells Godot to cook up some random numbers for us. The single-line `_ready` function ④ ensures that these random numbers are different each time you play the game (look into pseudorandomness for more detail). Line ⑬'s `randf_range` takes in two numbers between which the random number is generated.

7 Game Over

We are all out of mechanics: the game should work... until you hit an obstacle. We need to know when the game ends and we should display a screen afterwards.

Signals So far, all information sent between nodes has gone *down* the scene tree; the main node instructs the obstacles where to start. This time, we need information to travel *up* the tree; the bird should alert the main node when it hits an obstacle. However, nodes should never know who their parents are. Signals allow nodes to fire information up the tree without knowing who might handle or react to their emitting.

We can know when the bird hits an obstacle if it collides with anything. Collision detection occurs in `move_and_slide`: it returns true when a collision has occurred. Below are the lines which need to be changed and added.

```
1 signal gameover
2
3 # ---- // SNIP // ---- #
4
5 func _physics_process(delta: float) -> void:
6     # ---- // SNIP // ---- #
7
8     var collision: bool = move_and_slide()
9     if collision:
10         gameover.emit()
```

Listing 7: Preloading the obstacle

We need to tell the main node how to detect and react to this signal. Go back to the main scene and ensure the CharacterBody2D (bird) is selected in the scene tree (upper left). Switch from the inspector to the signals tab as in [fig.10].

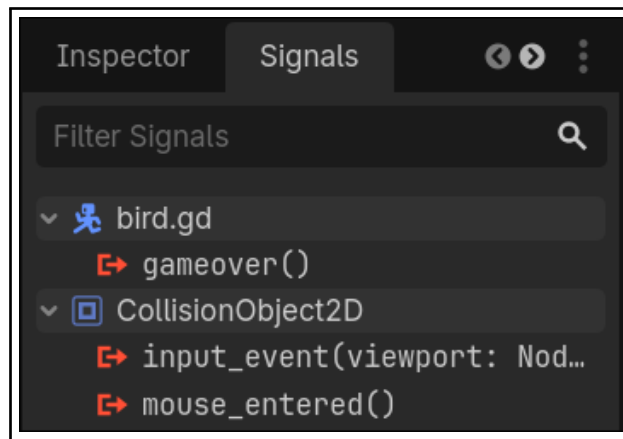


Figure 10: The CharacterBody2D signals tab

Double click the ‘gameover’ signal and connect it to the root node with an appropriate receiver name (e.g., `_on_gameover`). The new function will appear at the bottom of the main node’s script. The simplest way to end the game is ‘`get_tree().quit()`.’ Try out the game!

Congratulations! Flappy bird is functioning. The official tutorial ends here. See the next section for less guided tasks and other ideas for implementation. Thanks for reading!

8 And Beyond

These extension activities are strictly in order of implementation difficulty (from least to most tricky). I have put some research advice next to each idea:

1. *Sprites for everyone*: Sprite2Ds on every node
2. *Game Over screen*: Sprite2Ds, node process_mode, visibility and queue_free
3. *Flying off the screen*: StaticBody2D node, WorldBoundaryShape2D shape
4. *Different obstacle types*: Use other games for inspiration, apply what you've learnt
5. *Custom game-over animations*: AnimatedSprite2D, tween
6. *Earning points*: UI labels
7. *Parallax background*: Parallax2D
8. *Menu & restart screens*: If you've got this far... go it alone!

References

- [1] David Kushner. The flight of the birdman: Flappy bird creator dong nguyen speaks out. *Rolling Stone*, March 2014. URL: <https://www.rollingstone.com/culture/rs-gaming/the-flight-of-the-birdman-flappy-bird-creator-dong-nguyen-speaks-out-112457/>.
- [2] Juan Linietsky. First public release!, January 2014. URL: <https://godotengine.org/article/first-public-release/>.
- [3] Rich Stanton. Flappy bird makes an unlikely comeback after its creator apparently let the trademark lapse and the vultures swept right in. *PCGamer*, September 2024. URL: <https://www.pcgamer.com/games/action/flappy-bird-makes-an-unlikely-comeback-after-its-creator-apparently-let-the-trademark-lapse-and-the-vultures-swept-right-in/>.
- [4] Lance Ulanoff. Flappy bird reboot will never match the awfulness of the original and that's a problem. *techradar*, September 2024. URL: <https://www.techradar.com/computing/websites-apps/flappy-bird-reboot-will-never-match-the-awfulness-of-the-original-and-thats-a-problem>.