Your first game

Overview

This tutorial will guide you through making your first Godot project. You will learn how the Godot editor works, how to structure a project, and how to build a 2D game.

> **Note**
>
> This project is an introduction to the Godot engine. It assumes that you have some programming experience already. If you're new to programming entirely, you should start here: Scripting.

The game is called "Dodge the Creeps!". Your character must move and avoid the enemies for as long as possible. Here is a preview of the final result:

Why 2D? 3D games are much more complex than 2D ones. You should stick to 2D until you have a good understanding of the game development process.

Project setup

Launch Godot and create a new project. Then, download dodge_assets.zip - the images and sounds you'll be using to make the game. Unzip these files to your project folder.

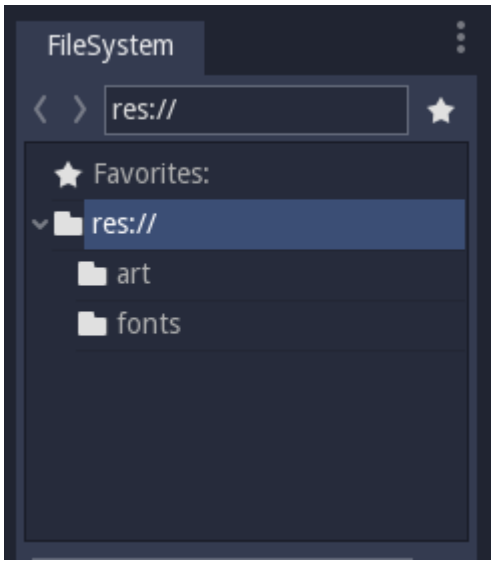
Note

For this tutorial, we will assume you are familiar with the editor. If you haven't read Scenes and nodes, do so now for an explanation of setting up a project and using the editor.

This game will use portrait mode, so we need to adjust the size of the game window. Click on Project -> Project Settings -> Display -> Window and set "Width" to `480` and "Height" to `720`.

Organizing the project

In this project, we will make 3 independent scenes: `Player`, `Mob`, and `HUD`, which we will combine into the game's `Main` scene. In a larger project, it might be useful to make folders to hold the various scenes and their scripts, but for this relatively small game, you can save your scenes and scripts in the project's root folder, referred to as `res://`. You can see your project folders in the FileSystem Dock in the lower left corner:

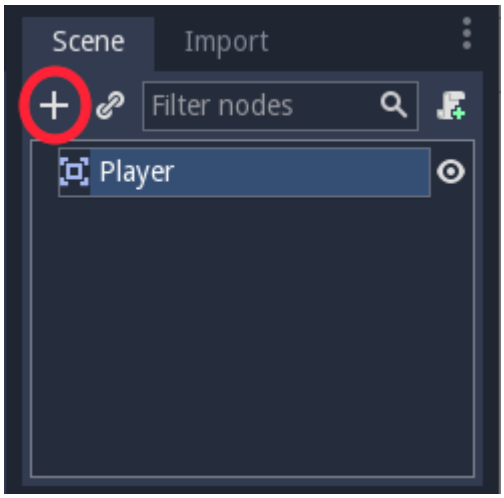


Player scene

The first scene we will make defines the `Player` object. One of the benefits of creating a separate Player scene is that we can test it separately, even before we've created other parts of the game.

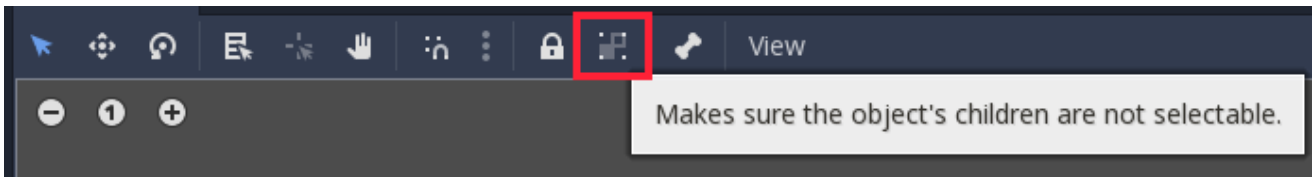
Node structure

To begin, click the "Add/Create a New Node" button and add an Area2D node to the scene.



With `Area2D` we can detect objects that overlap or run into the player. Change its name to `Player` by clicking on the node's name. This is the scene's root node. We can add additional nodes to the player to add functionality.

Before we add any children to the `Player` node, we want to make sure we don't accidentally move or resize them by clicking on them. Select the node and click the icon to the right of the lock; its tooltip says "Makes sure the object's children are not selectable."



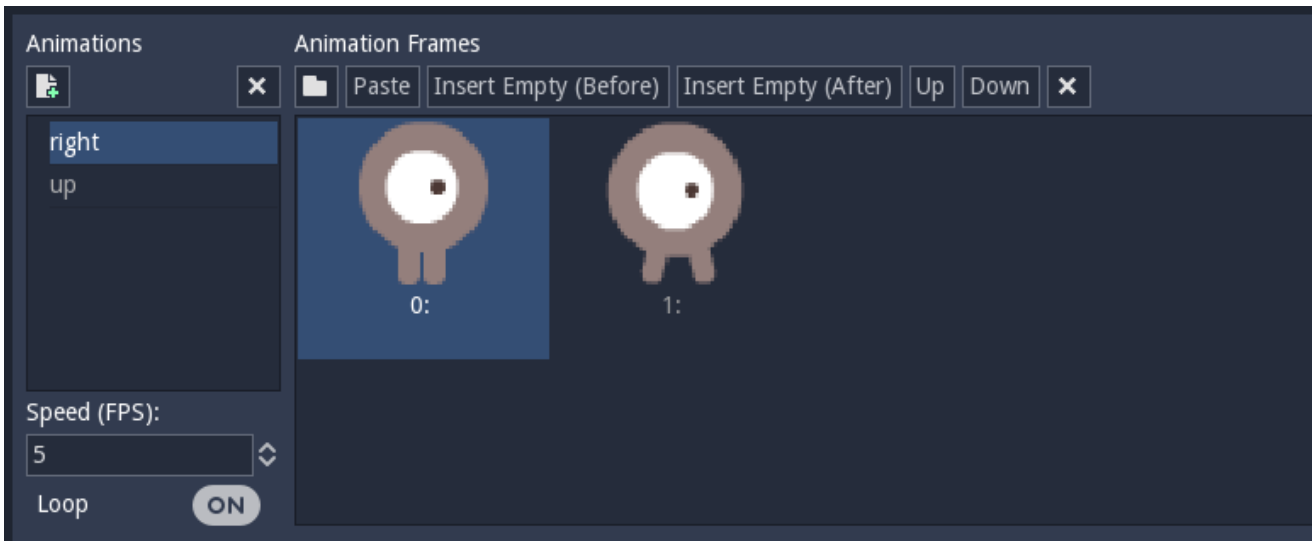
Save the scene. Click Scene -> Save, or press `Ctrl+S` on Windows/Linux or `Command+S` on Mac.

> **Note**
>
> For this project, we will be following the Godot naming conventions.
>
> - **GDScript**: Classes (nodes) use PascalCase, variables and functions use snake_case, and constants use ALL_CAPS (See GDScript style guide).
> - **C#**: Classes, export variables and methods use PascalCase, private fields use _camelCase, local variables and parameters use camelCase (See C# style guide). Be careful to type the method names precisely when connecting signals.

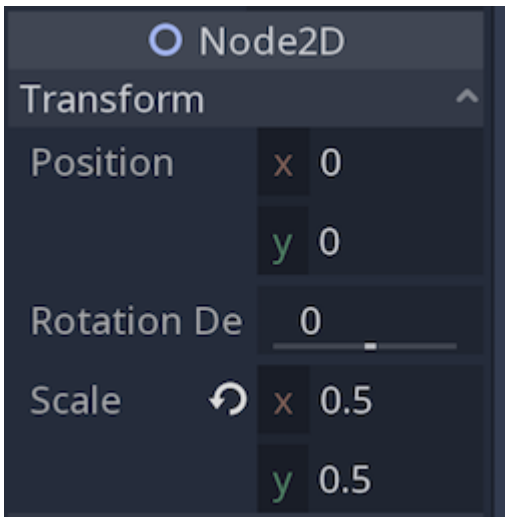
Sprite animation

Click on the `Player` node and add an AnimatedSprite node as a child. The `AnimatedSprite` will handle the appearance and animations for our player. Notice that there is a warning symbol next to the node. An `AnimatedSprite` requires a SpriteFrames resource, which is a list of the animations it can display. To create one, find the `Frames` property in the Inspector and click "[empty]" -> "New SpriteFrames". This should automatically open the SpriteFrames panel.

On the left is a list of animations. Click the “default” one and rename it to “right”. Then click the “Add” button to create a second animation named “up”. Drag the two images for each animation, named `playerGrey_up[1/2]` and `playerGrey_walk[1/2]`, into the “Animation Frames” side of the panel:

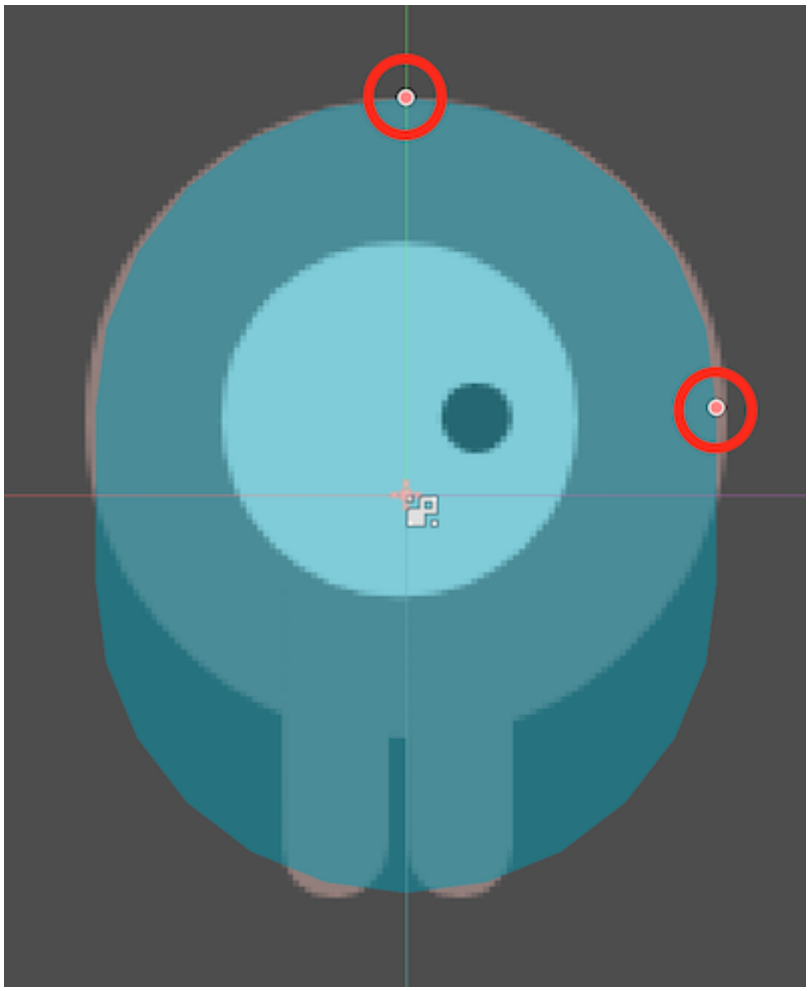


The player images are a bit too large for the game window, so we need to scale them down. Click on the `AnimatedSprite` node and set the `Scale` property to `(0.5, 0.5)`. You can find it in the Inspector under the `Node2D` heading.

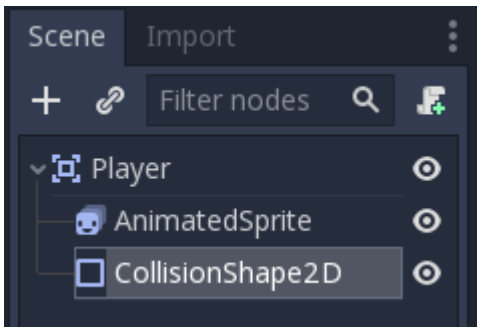


Finally, add a CollisionShape2D as a child of `Player`. This will determine the player’s “hitbox”, or the bounds of its collision area. For this character, a `CapsuleShape2D` node gives the best fit, so next to “Shape” in the Inspector, click “[empty]”” -> “New CapsuleShape2D”. Using the two

size handles, resize the shape to cover the sprite:

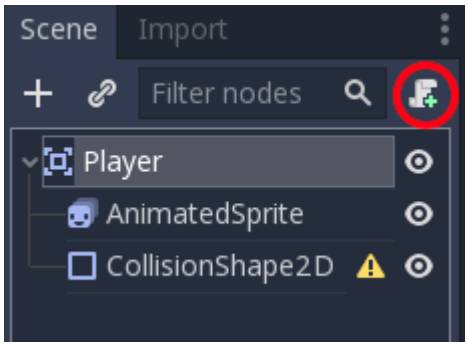


When you're finished, your `Player` scene should look like this:



Moving the player

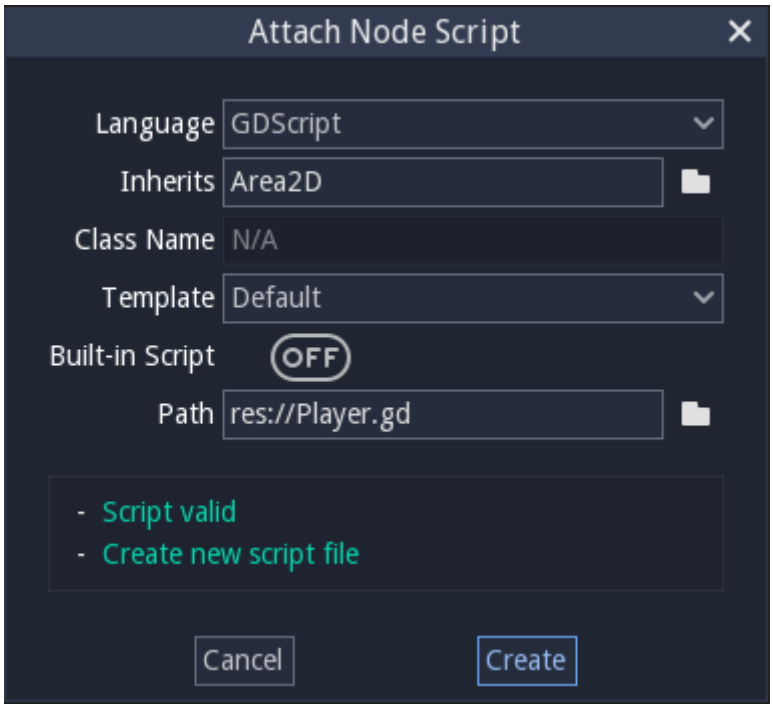
Now we need to add some functionality that we can't get from a built-in node, so we'll add a script. Click the `Player` node and click the "Add Script" button:



In the script settings window, you can leave the default settings alone. Just click “Create”:

Note

If you’re creating a C# script or other languages, select the language from the *language* drop down menu before hitting create.



Note

If this is your first time encountering GDScript, please read Scripting before continuing.

Start by declaring the member variables this object will need:

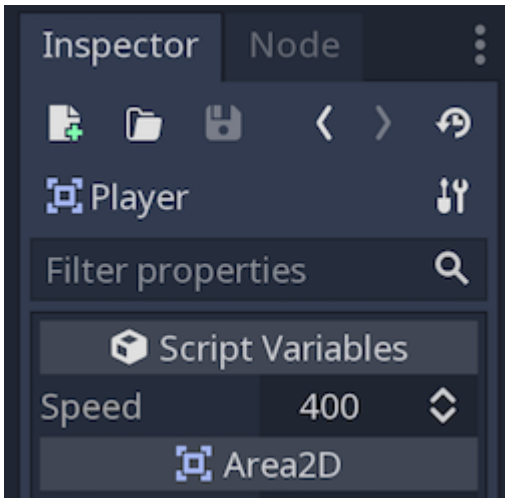
GDScript C#

```
extends Area2D

export var speed = 400  # How fast the player will move (pixels/sec).
var screen_size  # Size of the game window.
```

Using the `export` keyword on the first variable `speed` allows us to set its value in the Inspector. This can be handy for values that you want to be able to adjust just like a node's built-in properties. Click on the `Player` node and you'll see the property now appears in the "Script Variables" section of the Inspector. Remember, if you change the value here, it will override the value written in the script.

> **Warning**
>
> If you're using C#, you need to (re)build the project assemblies whenever you want to see new export variables or signals. This build can be manually triggered by clicking the word "Mono" at the bottom of the editor window to reveal the Mono Panel, then clicking the "Build Project" button.



The `_ready()` function is called when a node enters the scene tree, which is a good time to find the size of the game window:

GDScript | C#

```
func _ready():
    screen_size = get_viewport_rect().size
```

Now we can use the `_process()` function to define what the player will do. `_process()` is called every frame, so we'll use it to update elements of our game, which we expect will change often. For the player, we need to do the following:

- Check for input.
- Move in the given direction.
- Play the appropriate animation.

First, we need to check for input - is the player pressing a key? For this game, we have 4 direction inputs to check. Input actions are defined in the Project Settings under "Input Map". Here, you can define custom events and assign different keys, mouse events, or other inputs to them. For this demo, we will use the default events that are assigned to the arrow keys on the keyboard.

You can detect whether a key is pressed using `Input.is_action_pressed()`, which returns `true` if it is pressed or `false` if it isn't.

GDScript C#

```
func _process(delta):
    var velocity = Vector2()  # The player's movement vector.
    if Input.is_action_pressed("ui_right"):
        velocity.x += 1
    if Input.is_action_pressed("ui_left"):
        velocity.x -= 1
    if Input.is_action_pressed("ui_down"):
        velocity.y += 1
    if Input.is_action_pressed("ui_up"):
        velocity.y -= 1
    if velocity.length() > 0:
        velocity = velocity.normalized() * speed
        $AnimatedSprite.play()
    else:
        $AnimatedSprite.stop()
```

We start by setting the `velocity` to `(0, 0)` - by default the player should not be moving. Then we check each input and add/subtract from the `velocity` to obtain a total direction. For example, if you hold `right` and `down` at the same time, the resulting `velocity` vector will be `(1, 1)`. In this case, since we're adding a horizontal and a vertical movement, the player would move *faster* than if it just moved horizontally.

We can prevent that if we *normalize* the velocity, which means we set its *length* to `1`, and multiply by the desired speed. This means no more fast diagonal movement.

> **Tip**
>
> If you've never used vector math before, or need a refresher, you can see an explanation of vector usage in Godot at Vector math. It's good to know but won't be necessary for the rest of this tutorial.

We also check whether the player is moving so we can start or stop the AnimatedSprite animation.

> **Tip**

In GDScript, `$` returns the node at the relative path from the current node, or returns `null` if the node is not found. Since AnimatedSprite is a child of the current node, we can use `$AnimatedSprite`.

`$` is shorthand for `get_node()`. So in the code above, `$AnimatedSprite.play()` is the same as `get_node("AnimatedSprite").play()`.

Now that we have a movement direction, we can update the player's position and use `clamp()` to prevent it from leaving the screen by adding the following to the bottom of the `_process` function:

GDScript C#

```
position += velocity * delta
position.x = clamp(position.x, 0, screen_size.x)
position.y = clamp(position.y, 0, screen_size.y)
```

Tip

Clamping a value means restricting it to a given range.

Click "Play Scene" (`F6`) and confirm you can move the player around the screen in all directions. The console output that opens upon playing the scene can be closed by clicking `Output` (which should be highlighted in blue) in the lower left of the Bottom Panel.

Warning

If you get an error in the "Debugger" panel that refers to a "null instance", this likely means you spelled the node name wrong. Node names are case-sensitive and `$NodeName` or `get_node("NodeName")` must match the name you see in the scene tree.

Choosing animations

Now that the player can move, we need to change which animation the AnimatedSprite is playing based on direction. We have a "right" animation, which should be flipped horizontally using the `flip_h` property for left movement, and an "up" animation, which should be flipped vertically with `flip_v` for downward movement. Let's place this code at the end of our `_process()` function:

GDScript C#

```
if velocity.x != 0:
    $AnimatedSprite.animation = "right"
    $AnimatedSprite.flip_v = false
    # See the note below about boolean assignment
    $AnimatedSprite.flip_h = velocity.x < 0
elif velocity.y != 0:
    $AnimatedSprite.animation = "up"
    $AnimatedSprite.flip_v = velocity.y > 0
```

Note

The boolean assignments in the code above are a common shorthand for programmers. Consider this code versus the shortened boolean assignment above:

GDScript C#

```
if velocity.x < 0:
    $AnimatedSprite.flip_h = true
else:
    $AnimatedSprite.flip_h = false
```

Play the scene again and check that the animations are correct in each of the directions. When you're sure the movement is working correctly, add this line to `_ready()`, so the player will be hidden when the game starts:

GDScript C#

```
hide()
```

Preparing for collisions

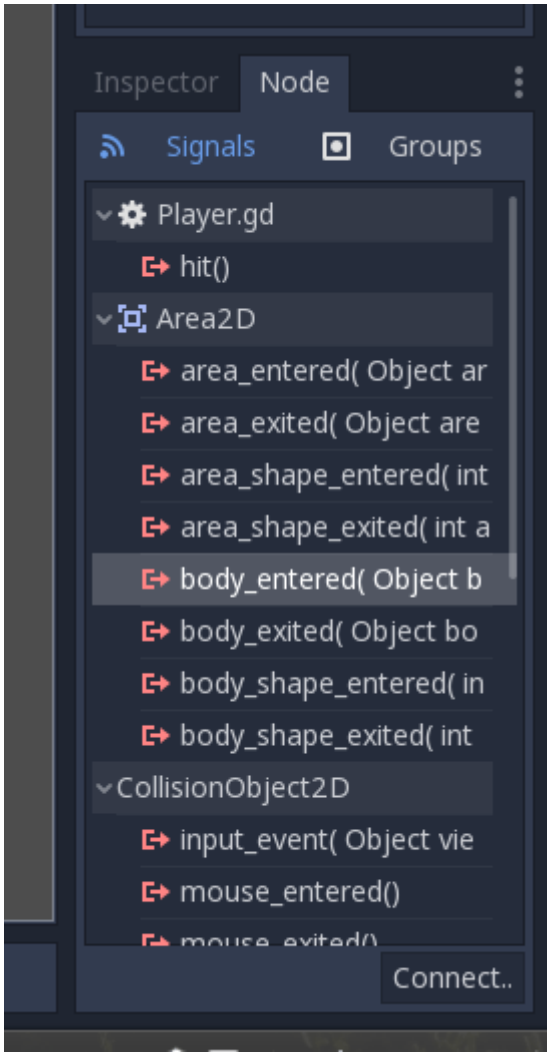
We want `Player` to detect when it's hit by an enemy, but we haven't made any enemies yet! That's OK, because we're going to use Godot's *signal* functionality to make it work.

Add the following at the top of the script, after `extends Area2d`:

GDScript C#

```
signal hit
```

This defines a custom signal called “hit” that we will have our player emit (send out) when it collides with an enemy. We will use `Area2D` to detect the collision. Select the `Player` node and click the “Node” tab next to the Inspector tab to see the list of signals the player can emit:



Notice our custom “hit” signal is there as well! Since our enemies are going to be `RigidBody2D` nodes, we want the `body_entered( Object body )` signal; this will be emitted when a body contacts the player. Click “Connect..” and then “Connect” again on the “Connecting Signal” window. We don’t need to change any of these settings - Godot will automatically create a function in your player’s script. This function will be called whenever the signal is emitted - it *handles* the signal.

> **Tip**
>
> When connecting a signal, instead of having Godot create a function for you, you can also give the name of an existing function that you want to link the signal to.

Add this code to the function:

GDScript C#

```
func _on_Player_body_entered(body):
    hide()  # Player disappears after being hit.
    emit_signal("hit")
    $CollisionShape2D.set_deferred("disabled", true)
```

Each time an enemy hits the player, the signal is going to be emitted. We need to disable the player's collision so that we don't trigger the `hit` signal more than once.

> **Note**
>
> Disabling the area's collision shape can cause an error if it happens in the middle of the engine's collision processing. Using `set_deferred()` allows us to have Godot wait to disable the shape until it's safe to do so.

The last piece for our player is to add a function we can call to reset the player when starting a new game.

GDScript C#

```
func start(pos):
    position = pos
    show()
    $CollisionShape2D.disabled = false
```

Enemy scene

Now it's time to make the enemies our player will have to dodge. Their behavior will not be very complex: mobs will spawn randomly at the edges of the screen and move in a random direction in a straight line, then despawn when they go offscreen.

We will build this into a `Mob` scene, which we can then *instance* to create any number of independent mobs in the game.

Node setup

Click Scene -> New Scene and we'll create the Mob.

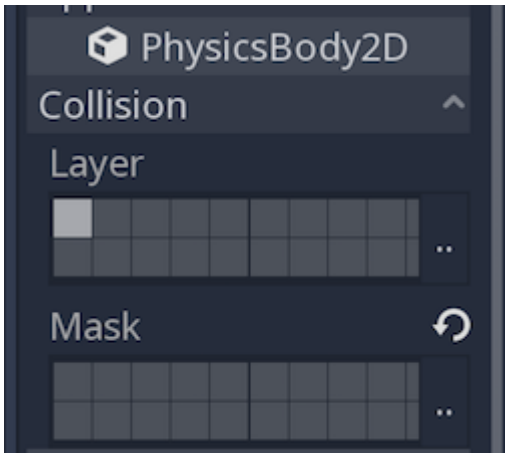
The Mob scene will use the following nodes:

- RigidBody2D (named `Mob`)

- AnimatedSprite
- CollisionShape2D
- VisibilityNotifier2D (named `Visibility`)

Don't forget to set the children so they can't be selected, like you did with the Player scene.

In the RigidBody2D properties, set `Gravity Scale` to `0`, so the mob will not fall downward. In addition, under the `PhysicsBody2D` section, click the `Mask` property and uncheck the first box. This will ensure the mobs do not collide with each other.



Set up the AnimatedSprite like you did for the player. This time, we have 3 animations: `fly`, `swim`, and `walk`. Set the `Playing` property in the Inspector to "On" and adjust the "Speed (FPS)" setting as shown below. We'll select one of these animations randomly so that the mobs will have some variety.

`fly` should be set to 3 FPS, with `swim` and `walk` set to 4 FPS.

Like the player images, these mob images need to be scaled down. Set the `AnimatedSprite`'s `Scale` property to `(0.75, 0.75)`.

As in the `Player` scene, add a `CapsuleShape2D` for the collision. To align the shape with the image, you'll need to set the `Rotation Degrees` property to `90` under `Node2D`.

Enemy script

Add a script to the `Mob` and add the following member variables:

GDScript C#

```
extends RigidBody2D

export var min_speed = 150  # Minimum speed range.
export var max_speed = 250  # Maximum speed range.
var mob_types = ["walk", "swim", "fly"]
```

When we spawn a mob, we'll pick a random value between `min_speed` and `max_speed` for how fast each mob will move (it would be boring if they were all moving at the same speed). We also have an array containing the names of the three animations, which we'll use to select a random one. Make sure you've spelled these the same in the script and in the SpriteFrames resource.

Now let's look at the rest of the script. In `_ready()` we randomly choose one of the three animation types:

GDScript C#

```
func _ready():
    $AnimatedSprite.animation = mob_types[randi() % mob_types.size()]
```

Note

You must use `randomize()` if you want your sequence of "random" numbers to be different every time you run the scene. We're going to use `randomize()` in our `Main` scene, so we won't need it here. `randi() % n` is the standard way to get a random integer between `0` and `n-1`.

The last piece is to make the mobs delete themselves when they leave the screen. Connect the `screen_exited()` signal of the `Visibility` node and add this code:

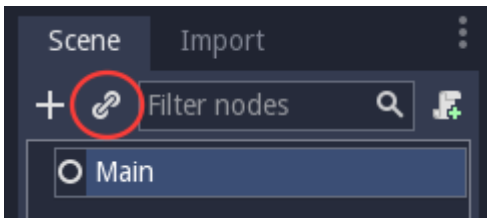
GDScript C#

```
func _on_Visibility_screen_exited():
    queue_free()
```

This completes the *Mob* scene.

Main scene

Now it's time to bring it all together. Create a new scene and add a Node named `Main`. Click the "Instance" button and select your saved `Player.tscn`.



> **Note**
>
> See Instancing to learn more about instancing.

Now, add the following nodes as children of `Main`, and name them as shown (values are in seconds):

- Timer (named `MobTimer`) - to control how often mobs spawn
- Timer (named `ScoreTimer`) - to increment the score every second
- Timer (named `StartTimer`) - to give a delay before starting
- Position2D (named `StartPosition`) - to indicate the player's start position

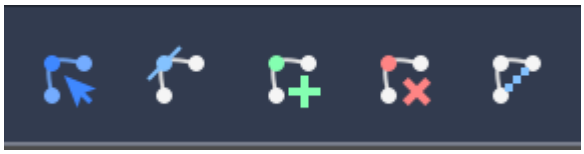
Set the `Wait Time` property of each of the `Timer` nodes as follows:

- `MobTimer`: `0.5`
- `ScoreTimer`: `1`
- `StartTimer`: `2`

In addition, set the `One Shot` property of `StartTimer` to "On" and set `Position` of the `StartPosition` node to `(240, 450)`.

Spawning mobs

The Main node will be spawning new mobs, and we want them to appear at a random location on the edge of the screen. Add a Path2D node named `MobPath` as a child of `Main`. When you select `Path2D`, you will see some new buttons at the top of the editor:



Select the middle one (“Add Point”) and draw the path by clicking to add the points at the corners shown. To have the points snap to the grid, make sure “Snap to Grid” is checked. This option can be found under the “Snapping options” button to the left of the “Lock” button, appearing as a series of three vertical dots.

Important

Draw the path in *clockwise* order, or your mobs will spawn pointing *outwards* instead of *inwards*!

After placing point `4` in the image, click the “Close Curve” button and your curve will be complete.

Now that the path is defined, add a PathFollow2D node as a child of `MobPath` and name it `MobSpawnLocation`. This node will automatically rotate and follow the path as it moves, so we can use it to select a random position and direction along the path.

Main script

Add a script to `Main`. At the top of the script, we use `export (PackedScene)` to allow us to choose the Mob scene we want to instance.

GDScript C#

```
extends Node

export (PackedScene) var Mob
var score

func _ready():
    randomize()
```

Drag `Mob.tscn` from the "FileSystem" panel and drop it in the `Mob` property under the Script Variables of the `Main` node.

Next, click on the Player and connect the `hit` signal. We want to make a new function named `game_over`, which will handle what needs to happen when a game ends. Type "game_over" in the "Method In Node" box at the bottom of the "Connecting Signal" window. Add the following code, as well as a `new_game` function to set everything up for a new game:

GDScript C#

```
func game_over():
    $ScoreTimer.stop()
    $MobTimer.stop()

func new_game():
    score = 0
    $Player.start($StartPosition.position)
    $StartTimer.start()
```

Now connect the `timeout()` signal of each of the Timer nodes (`StartTimer`, `ScoreTimer`, and `MobTimer`) to the main script. `StartTimer` will start the other two timers. `ScoreTimer` will increment the score by 1.

GDScript C#

```
func _on_StartTimer_timeout():
    $MobTimer.start()
    $ScoreTimer.start()

func _on_ScoreTimer_timeout():
    score += 1
```

In `_on_MobTimer_timeout()`, we will create a mob instance, pick a random starting location along the `Path2D`, and set the mob in motion. The `PathFollow2D` node will automatically rotate as it follows the path, so we will use that to select the mob's direction as well as its position.

Note that a new instance must be added to the scene using `add_child()`.

Now click on `MobTimer` in the scene window then head to inspector window, switch to node view then click on `timeout()` and connect the signal.

Add the following code:

GDScript C#

```
func _on_MobTimer_timeout():
    # Choose a random location on Path2D.
    $MobPath/MobSpawnLocation.set_offset(randi())
    # Create a Mob instance and add it to the scene.
    var mob = Mob.instance()
    add_child(mob)
    # Set the mob's direction perpendicular to the path direction.
    var direction = $MobPath/MobSpawnLocation.rotation + PI / 2
    # Set the mob's position to a random location.
    mob.position = $MobPath/MobSpawnLocation.position
    # Add some randomness to the direction.
    direction += rand_range(-PI / 4, PI / 4)
    mob.rotation = direction
    # Set the velocity (speed & direction).
    mob.linear_velocity = Vector2(rand_range(mob.min_speed, mob.max_speed), 0)
    mob.linear_velocity = mob.linear_velocity.rotated(direction)
```

Important

In functions requiring angles, GDScript uses *radians*, not degrees. If you're more comfortable working with degrees, you'll need to use the `deg2rad()` and `rad2deg()` functions to convert between the two.

HUD

The final piece our game needs is a UI: an interface to display things like score, a "game over" message, and a restart button. Create a new scene, and add a CanvasLayer node named `HUD`. "HUD" stands for "heads-up display", an informational display that appears as an overlay on top of the game view.

The CanvasLayer node lets us draw our UI elements on a layer above the rest of the game, so that the information it displays isn't covered up by any game elements like the player or mobs.

The HUD displays the following information:

- Score, changed by `ScoreTimer`.
- A message, such as "Game Over" or "Get Ready!"
- A "Start" button to begin the game.

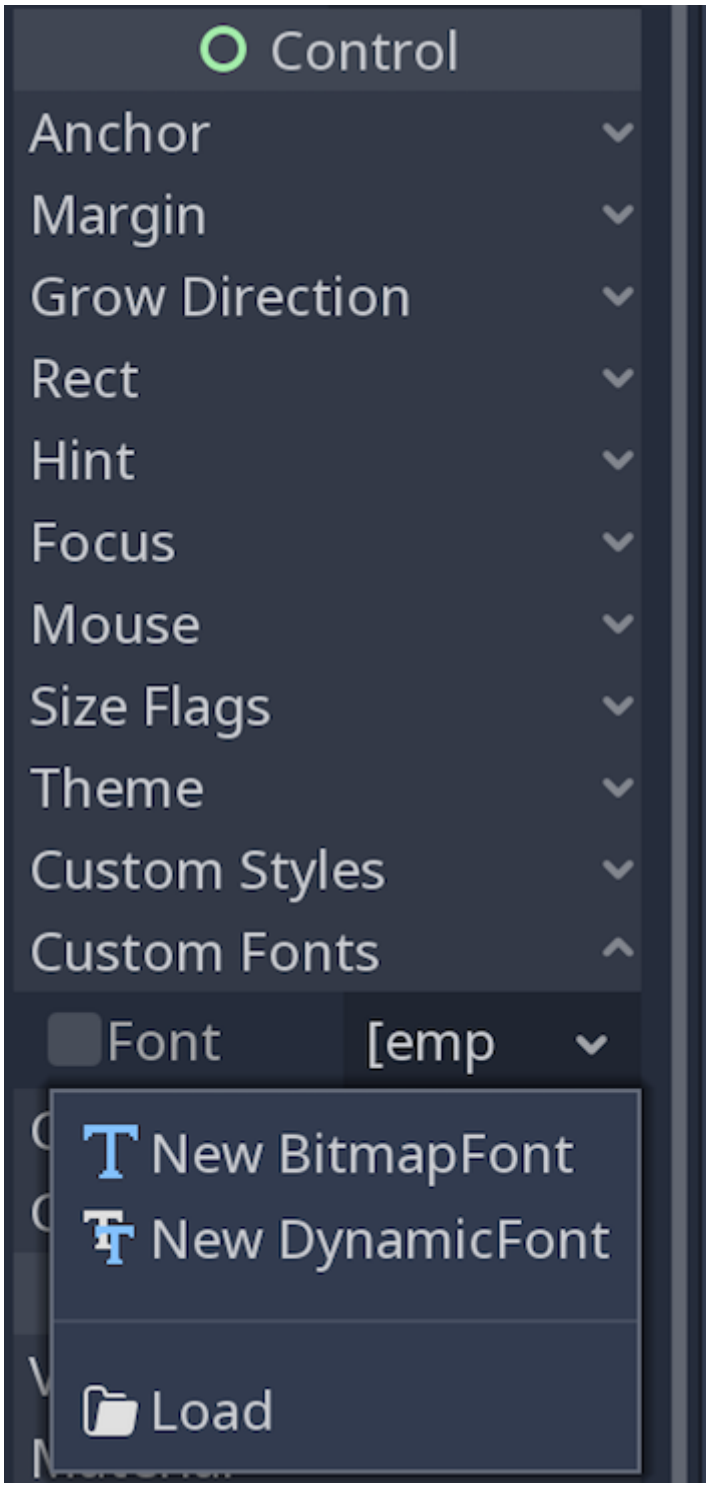
The basic node for UI elements is Control. To create our UI, we'll use two types of Control nodes: Label and Button.

Create the following as children of the `HUD` node:

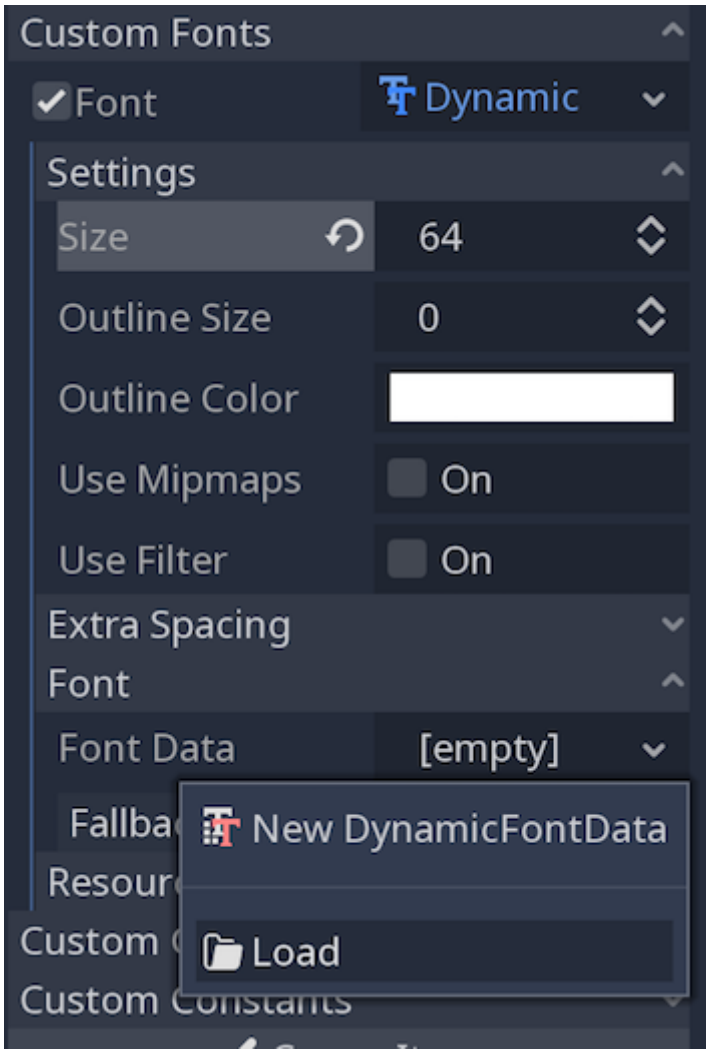
- Label named `ScoreLabel`.
- Label named `MessageLabel`.
- Button named `StartButton`.
- Timer named `MessageTimer`.

Click on the `ScoreLabel` and type a number into the *Text* field in the Inspector. The default font for `Control` nodes is small and doesn't scale well. There is a font file included in the game assets called "Xolonium-Regular.ttf". To use this font, do the following for each of the three `Control` nodes:

1. Under "Custom Fonts", choose "New DynamicFont"



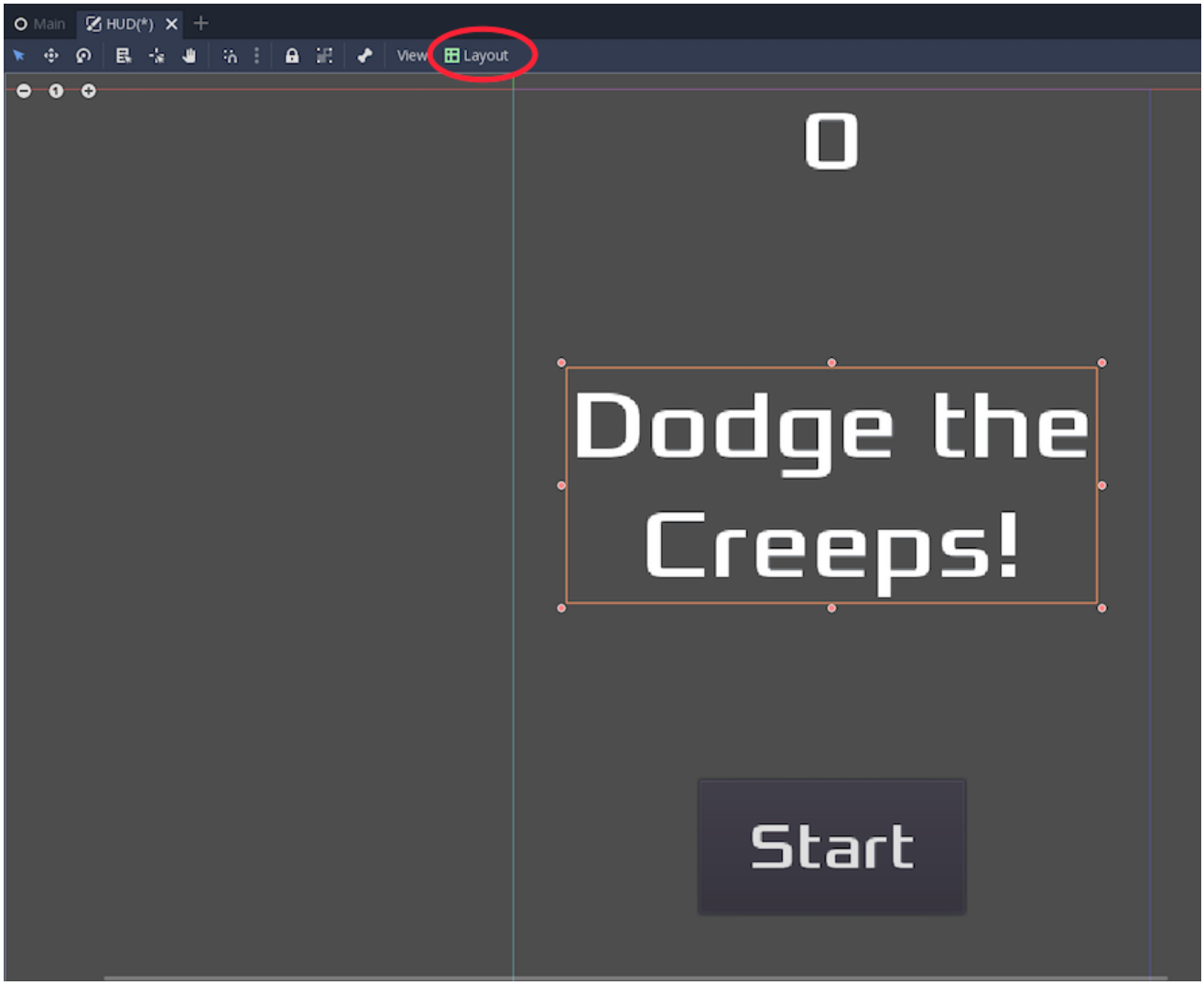
2. Click on the “DynamicFont” you added, and under “Font/Font Data”, choose “Load” and select the “Xolonium-Regular.ttf” file. You must also set the font’s `Size`. A setting of `64` works well.



Note

Anchors and Margins: `Control` nodes have a position and size, but they also have anchors and margins. Anchors define the origin - the reference point for the edges of the node. Margins update automatically when you move or resize a control node. They represent the distance from the control node's edges to its anchor. See Design interfaces with the Control nodes for more details.

Arrange the nodes as shown below. Click the "Anchor" button to set a Control node's anchor:



You can drag the nodes to place them manually, or for more precise placement, use the following settings:

ScoreLabel

- *Text* : `0`
- *Layout* : “Top Wide”
- *Align* : “Center”

MessageLabel

- *Text* : `Dodge the Creeps!`
- *Layout* : “HCenter Wide”
- *Align* : “Center”

StartButton

- *Text* : `Start`
- *Layout* : “Center Bottom”
- *Margin* :
  - Top: `-200`
  - Bottom: `-100`

Now add this script to `HUD`:

GDScript C#

```
extends CanvasLayer

signal start_game
```

The `start_game` signal tells the `Main` node that the button has been pressed.

GDScript C#

```
func show_message(text):
    $MessageLabel.text = text
    $MessageLabel.show()
    $MessageTimer.start()
```

This function is called when we want to display a message temporarily, such as “Get Ready”. On the `MessageTimer`, set the `Wait Time` to `2` and set the `One Shot` property to “On”.

GDScript C#

```
func show_game_over():
    show_message("Game Over")
    yield($MessageTimer, "timeout")
    $MessageLabel.text = "Dodge the\nCreeps!"
    $MessageLabel.show()
    yield(get_tree().create_timer(1), 'timeout')
    $StartButton.show()
```

This function is called when the player loses. It will show “Game Over” for 2 seconds, then return to the title screen and, after a brief pause, show the “Start” button.

Note

When you need to pause for a brief time, an alternative to using a Timer node is to use the SceneTree’s `create_timer()` function. This can be very useful to delay, such as in the above code, where we want to wait a little bit of time before showing the “Start” button.

GDScript C#

```
func update_score(score):
    $ScoreLabel.text = str(score)
```

This function is called by `Main` whenever the score changes.

Connect the `timeout()` signal of `MessageTimer` and the `pressed()` signal of `StartButton`.

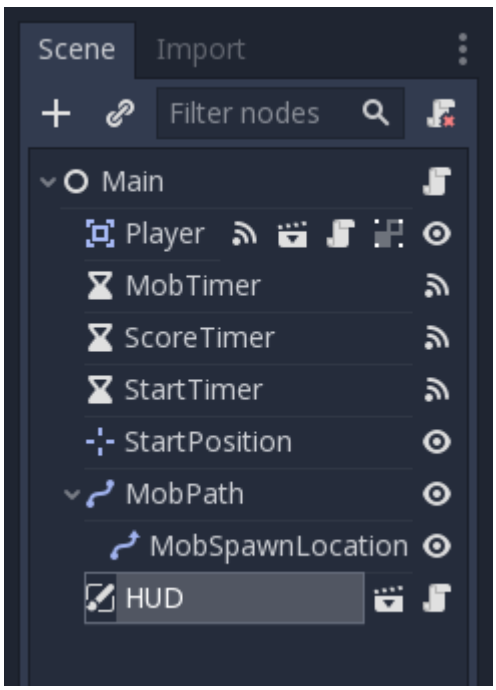
GDScript C#

```
func _on_StartButton_pressed():
    $StartButton.hide()
    emit_signal("start_game")

func _on_MessageTimer_timeout():
    $MessageLabel.hide()
```

Connecting HUD to Main

Now that we're done creating the `HUD` scene, save it and go back to `Main`. Instance the `HUD` scene in `Main` like you did the `Player` scene, and place it at the bottom of the tree. The full tree should look like this, so make sure you didn't miss anything:



Now we need to connect the `HUD` functionality to our `Main` script. This requires a few additions to the `Main` scene:

In the Node tab, connect the HUD's `start_game` signal to the `new_game()` function.

In `new_game()`, update the score display and show the “Get Ready” message:

GDScript C#

```
$HUD.update_score(score)
$HUD.show_message("Get Ready")
```

In `game_over()` we need to call the corresponding `HUD` function:

GDScript C#

```
$HUD.show_game_over()
```

Finally, add this to `_on_ScoreTimer_timeout()` to keep the display in sync with the changing score:

GDScript C#

```
$HUD.update_score(score)
```

Now you’re ready to play! Click the “Play the Project” button. You will be asked to select a main scene, so choose `Main.tscn`.

Removing old creeps

If you play until “Game Over” and then start a new game the creeps from the previous game are still on screen. It would be better if they all disappeared at the start of a new game.

We’ll use the `start_game` signal that’s already being emitted by the `HUD` node to remove the remaining creeps. We can’t use the editor to connect the signal to the mobs in the way we need because there are no `Mob` nodes in the `Main` scene tree until we run the game. Instead we’ll use code.

Start by adding a new function to `Mob.gd`. `queue_free()` will delete the current node at the end of the current frame.

GDScript C#

```
func _on_start_game():
    queue_free()
```

Then in `Main.gd` add a new line inside the `_on_MobTimer_timeout()` function, at the end.

GDScript C#

```
$HUD.connect("start_game", mob, "_on_start_game")
```

This line tells the new Mob node (referenced by the `mob` variable) to respond to any `start_game` signal emitted by the `HUD` node by running its `_on_start_game()` function.

Finishing up

We have now completed all the functionality for our game. Below are some remaining steps to add a bit more "juice" to improve the game experience. Feel free to expand the gameplay with your own ideas.

Background

The default gray background is not very appealing, so let's change its color. One way to do this is to use a ColorRect node. Make it the first node under `Main` so that it will be drawn behind the other nodes. `ColorRect` only has one property: `Color`. Choose a color you like and drag the size of the `ColorRect` so that it covers the screen.

You could also add a background image, if you have one, by using a `Sprite` node.

Sound effects

Sound and music can be the single most effective way to add appeal to the game experience. In your game assets folder, you have two sound files: "House In a Forest Loop.ogg" for background music, and "gameover.wav" for when the player loses.

Add two AudioStreamPlayer nodes as children of `Main`. Name one of them `Music` and the other `DeathSound`. On each one, click on the `Stream` property, select "Load", and choose the corresponding audio file.

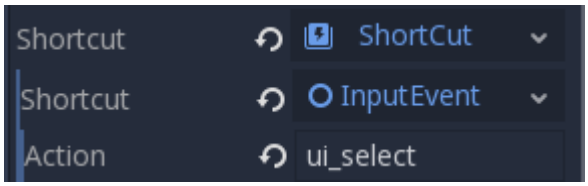
To play the music, add `$Music.play()` in the `new_game()` function and `$Music.stop()` in the `game_over()` function.

Finally, add `$DeathSound.play()` in the `game_over()` function.

Keyboard Shortcut

Since the game is played with keyboard controls, it would be convenient if we could also start the game by pressing a key on the keyboard. One way to do this is using the "Shortcut" property of the `Button` node.

In the `HUD` scene, select the `StartButton` and find its *Shortcut* property in the Inspector. Select "New Shortcut" and click on the "Shortcut" item. A second *Shortcut* property will appear. Select "New InputEventAction" and click the new "InputEvent". Finally, in the *Action* property, type the name `ui_select`. This is the default input event associated with the spacebar.



Now when the start button appears, you can either click it or press the spacebar to start the game.

Project files

You can find a completed version of this project at these locations:

- https://github.com/kidscancode/Godot3_dodge/releases
- https://github.com/godotengine/godot-demo-projects