Your first game

Adapted from
source: https://docs.godotengine.org/en/3.3/getting_started/step_by_step/your_first_game.html

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.

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 in an empty folder. Then, download dodge_assets.zip from this folder - the images and sounds you'll be using to make the game. Unzip these files to your project folder.

Your view will show a 3d space. Switch to 2d using the button at the top of the screen.

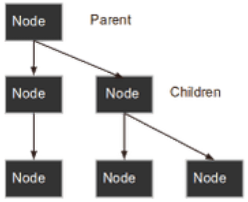
This game will use portrait mode, so we need to adjust the size of the game window. Click on Project -> Project Settings -> Display -> Window and set "Width" to `480` and "Height" to `720`.

Create a "2D Scene" root node using the button on the left hand side. You will see the node in the scene window on the left. Rename it from "Node2D" to "Main". This will be the main game area. You can now run the game by pressing the play button in the top right corner. When asked to select a main scene, choose the scene you just created. The game will then launch in a new window. Naturally, it will be a blank grey window for now. We will add to it soon.

Nodes and Scenes

Nodes are fundamental building blocks for creating a game. As mentioned above, a node can perform a variety of specialised functions. However, any given node always has the following attributes:

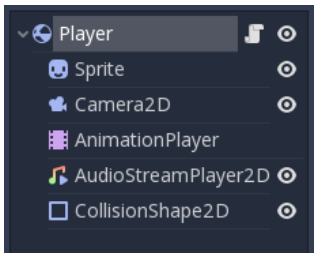
- It has a name.
- It has editable properties.
- It can receive a callback to process every frame.
- It can be extended (to have more functions).
- It can be added to another node as a child.



The last one is important. Nodes can have other nodes as children. When arranged in this way, the nodes become a **tree**.

In Godot, the ability to arrange nodes in this way creates a powerful tool for organising projects. Since different nodes have different functions, combining them allows for the creation of more complex functions.

For example, a character in a game may consist of a number of nodes which represent: their model, the contents of their backpack, the functionality of their weapon and the set of sounds they make. In this way, a developer can work on each building block separately and add them together to create the 'player' node, which is the root of the tree.

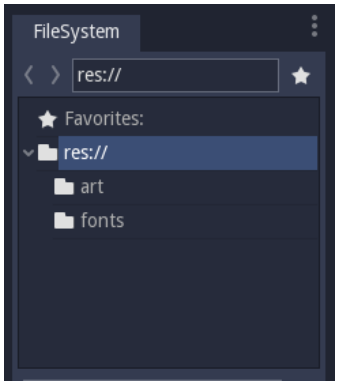


Once a set of nodes are assembled into a tree representing a piece of the game, they become a 'scene'. Examples of scenes in games could include a player, enemies, the world, or the menu system.

Running a game means running a scene. A project can contain several scenes, but for the game to start, one of them must be selected as the main scene (for example, the 'world' scene).

Organising 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:

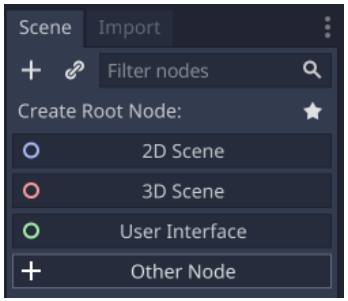


Player scene

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.

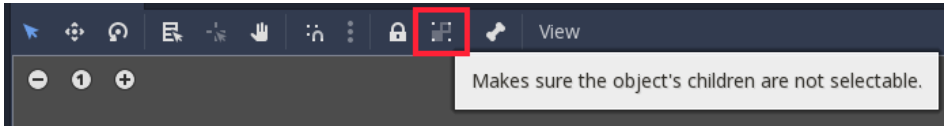
Node structure

To begin, select "New Scene" from the "Scene" menu. Again, we need to create a root node, but this time choose "Other Node" and search for Area2D.



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."



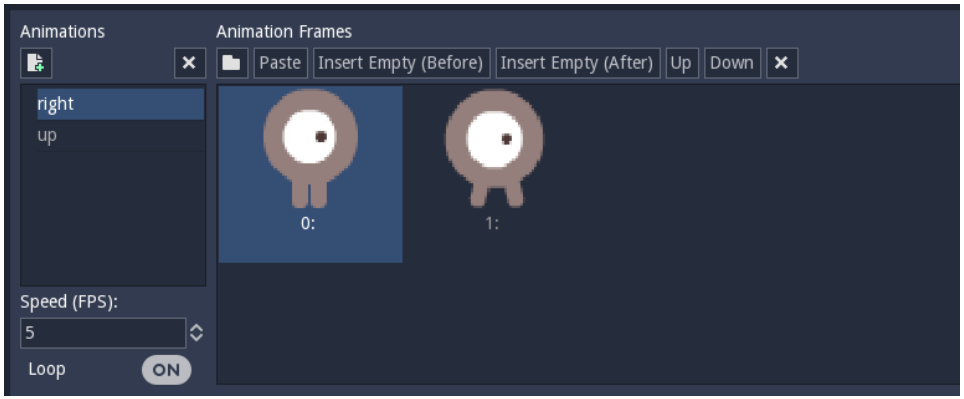
Save the scene as Player.tscn. Click Scene -> Save, or press `Ctrl+S` on Windows/Linux or `Command+S` on Mac.

Sprite animation

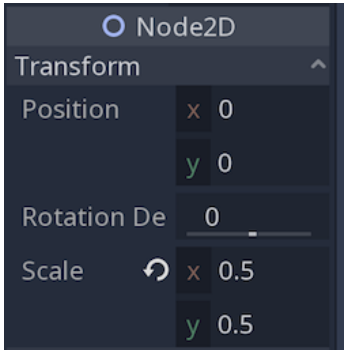
We now have a 'player scene', but it has no functionality. Therefore, we want to add a 2d animation to the player, called a 'sprite'.

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 "<null>" -> "New SpriteFrames". Next, in the same location, click `<SpriteFrames>`, then click "Open Editor" to 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:

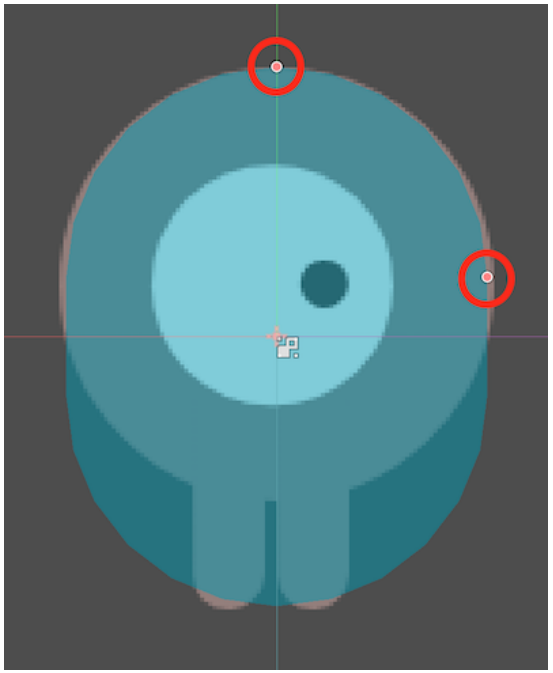


The player images are a bit too large for the game window, so we need to scale them down. Click on the `AnimatedSprite` node and set the `Scale` property to `(0.5, 0.5)`. You can find it in the Inspector under the `Node2D` heading.

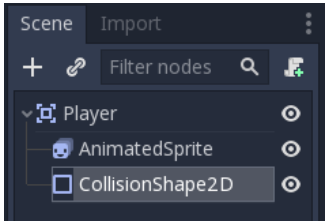


Now the player is visible, however we still need to define its boundaries to the game engine, this allows for the player to interact with the environment.

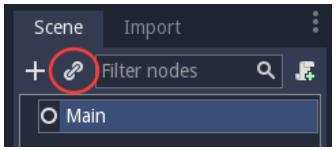
Add a CollisionShape2D as a child of `Player`. This will determine the player's "hitbox", or the bounds of its collision area. For this character, a `CapsuleShape2D` node gives the best fit, so next to "Shape" in the Inspector, click "<null>"" -> "New CapsuleShape2D". Using the two size handles inside the rectangle (not the ones outside), resize the shape to cover the sprite:



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



Finally, we can add the `Player` scene to the `Main` game scene. Select the `Main` scene from the tabs at the top of the screen to go back to the main scene and click on the `Main` node in the left panel. To add the `Player` scene as a node under the `Main` node, select the icon that looks like a chain and is called "Instance Child Scene". Select the `Player` scene and it will now appear as a child of the `Main` node.

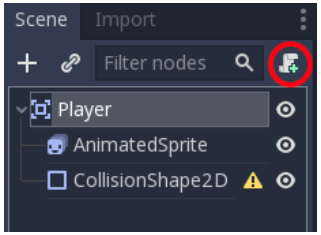


The player sprite should now appear in the top corner of the game window. Don't worry about the positioning yet. Try running the game again using the play button to confirm the sprite now appears in the corner. In the next section, we will add movement to the character.

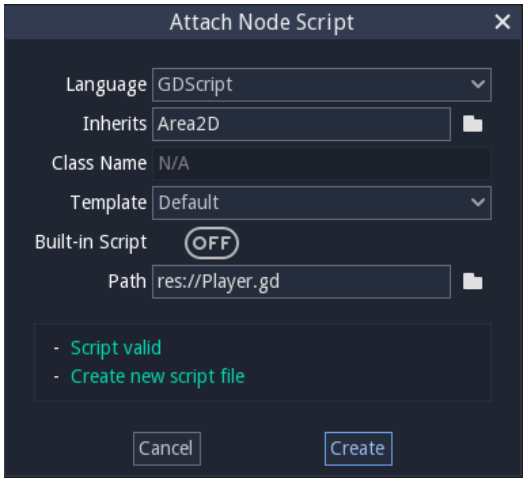
Remember to save your project.

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. Navigate back to the Player scene, 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":



We can now view the script, which has been filled out with a template already. Lines beginning with a hash symbol (`#`) are 'comments' and are ignored by Godot.

The first line `extends Area2D` brings in all the settings of an Area2D object, we can ignore this for now.

The `_ready()` function is called when a node enters the scene tree (i.e. when the Player is first added to the game). We don't need to do anything here yet so we tell Godot to `pass` (i.e. move on).

We can use the `_process()` function to define what the player will do. `_process()` is called every frame. In other words, the game continually asks our 'player' scene what it wants to do, and the player scene determines this by running through the `_process()` command. The command is currently commented out, therefore the player object does nothing as the game runs. We need it 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 - the arrow keys.

We 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. We add the key into the command.

```
func _process(delta):
  if Input.is_action_pressed("ui_right"):
      print("right key pressed")
  if Input.is_action_pressed("ui_left"):
      print("left key pressed")
  if Input.is_action_pressed("ui_down"):
      print("down key pressed")
  if Input.is_action_pressed("ui_up"):
      print("up key pressed")
```

Now we are detecting the correct keys, we want them to change the direction the player is moving in. We can store the player's direction as a vector (i.e. a line) which has an x and y value, and have each key adjust the direction. Note that in this environment, the Y scale decreases in value as it moves up on the screen.

```
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
```

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 *normalise* the velocity, which means we set its *length* to `1`.

```
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"):
```

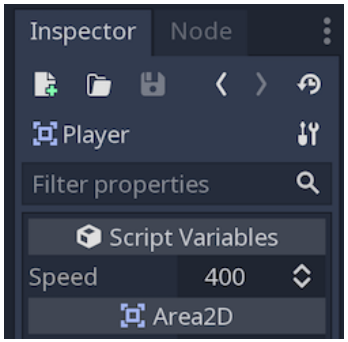
```
        velocity.y -= 1
    velocity = velocity.normalized()  # Note: American spelling
```

Now we can set the character to be travelling at a speed of 1 in one of 8 directions, but we'd like to set a speed. We can store this as a variable at the beginning of the script.

```
extends Area2D

export var speed = 400  # How fast the player will move (pixels/sec).
```

We use the `export` flag which allows us to set an override to the variable from the Inspector (i.e. without changing the code). 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.



Now we can multiply the velocity by this speed value.

```
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
    velocity = velocity.normalized() * speed
```

Finally, we can add a check for whether the player is moving so we can start or stop the AnimatedSprite animation. Remember the velocity is set to 0 at the beginning of each frame (each time `_process()` runs) so velocity will be 0 unless a key is being pressed.

```
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()
```

Note: 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 we are setting a movement direction and speed for the player object based on the keys being pressed, but the player object doesn't know what to do with those values. Therefore, we need to explain to player object how to update its position based on the direction we give it.

Remember we set the speed of player to 400 pixels per second. Therefore, if we pass the player the instructions to move left at a speed of 400 pixels/sec, we need to divide the value of 400 by the fraction of a second that the frame represents. This value is given to us as `delta`, so we can multiply our speed of 400 by the value `delta` to decide how far the player needs to move for this frame.

Our player object has an attribute `position`, so to represent movement, we simply need to tell it to exist at a new position for this frame, much like how an animation works. We calculate this new position by taking the current position and adding the velocity (which could be a negative value):

```
position += velocity * delta
```

Finally, we would like to prevent the player leaving the screen, as we'd have no idea where it was. We can do this easily by constraining the position value to be within the window size. First we can store the window size in a variable we define at the beginning of the script, and we get the window size and store it to this variable when we initialise the object -- in the `_ready()` function.

```
extends Area2D

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


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

Then we can update our `_process()` function to force the `position` to be within these boundaries using `clamp()`:

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

In the above code, clamp returns a value based on the following logic: `clamp(value_to_consider, minimum_value_acceptable, maximum_value_acceptable)` Therefore if the `position.x` is between 0 and 480, `clamp()` will return `position.x` , if it is smaller than 0, `clamp()` will return 0, and if it is higher, `clamp()` will return 480. Thus we cannot set the position of our character outside the window.

Click "Play Scene" (`F6`) and confirm you can move the player around the screen in all directions.

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:

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

When we use an expression such as `velocity.x < 0`, the returned value will either be `true` or `false`. Therefore, we can streamline the above code into a single line as: `$AnimatedSprite.flip_h = velocity.x < 0` Finally, we want to choose the correct animation from the set we created earlier -- "right" if we're moving left or right, and "up" if we're moving up or down (in this implementation, diagonal movement will use the "right" animation).

```
if velocity.x != 0:
    $AnimatedSprite.animation = "right"
elif velocity.y != 0:
    $AnimatedSprite.animation = "up"

$AnimatedSprite.flip_h = velocity.x < 0
$AnimatedSprite.flip_v = velocity.y > 0
```

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:

```
hide()
```

Preparing for collisions

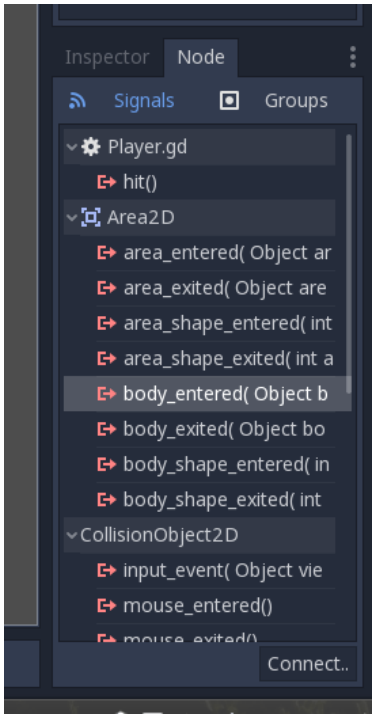
We want `Player` to detect when it's hit by an enemy, but we haven't made any enemies yet! That's OK, because we're going to use Godot's *signal* functionality to make it work.

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

```
signal hit
```

This defines a custom signal called "hit" that we will have our player emit (send out) when it collides with an enemy. This is how our player will tell the game that is has been hit, and what should happen. We will define this later.

We will use `Area2D` features 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 another body (e.g. of an enemy) enters (contacts) the body of 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 (i.e. whenever the player is hit).

Add this code to the function:

```
func _on_Player_body_entered(body):
    hide()  # Player disappears after being hit.
    emit_signal("hit")  # Player tells the game it has been hit.
```

Each time an enemy hits the player, the `hit` signal will be emitted. We need to disable the player's collision after the first collision so that we don't trigger the `hit` signal more than once. We can do that by disabling the `CollisionShape2D` node of the player. Typically, this would be done with the command `$CollisionShape2D.disabled = true`, however, disabling the area's collision shape can cause an error if it happens in the middle of the engine's collision processing. Using `call_deferred()` allows us to have Godot wait to disable the shape until it's safe to do so.

```
func _on_Player_body_entered(body):
    hide()  # Player disappears after being hit.
```

```
    emit_signal("hit")  # Player tells the game it has been hit.
    $CollisionShape2D.call_deferred("set_disabled", true)
```

Finally, we need to add a function we can call to reset the player when starting a new game. We show the player, move them to the middle of the screen and turn their CollisionShape2D on.

```
func start():
    position.x = screen_size.x / 2
    position.y = screen_size.y / 2
    show()
    $CollisionShape2D.disabled = false
```

We now have a completed player scene which we can add to the main game. Our player will spawn when we run it's `start()` function, can move around the screen, and will disappear and emit a `hit` signal when it collides with another body.

Enemy scene

Now it's time to make the enemies our player will have to dodge. Their behaviour 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. Instancing is when we use multiple copies of the same scene in the game, each of which may have their own unique characteristics.

Node setup

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

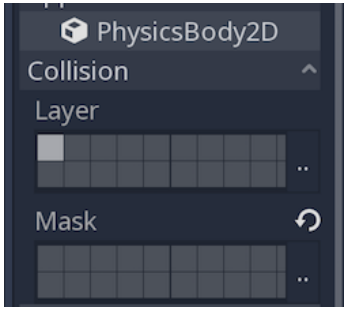
The Mob scene will use the following nodes which you can add:

- ***RigidBody2D (named `Mob`)***
  - AnimatedSprite
  - CollisionShape2D
  - VisibilityNotifier2D (named `Visibility`)

This is very similar to our player object. The main difference is the addition of the VisibilityNotifier2D node, which is able to check whether the scene (mob) is visible, and send signals based on the result. We can use this to create a signal when a mob moves off the screen (to delete them).

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)`. Check the 'playing' box so the animation is playing by default.

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. We want our mobs to be different, so unlike the player object, we set a max and min speed and we will choose a value between these each time a new mob is created. We also make a list of mob types (as per the animations we added) so we can choose between them later. Make sure they are spelled the same as in the AnimatedSprite node.

```
extends RigidBody2D

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

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

```
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 to the Mob script, and add this code:

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

This completes the Mob scene.

Main scene

Now it's time to bring it all together. Navigate back to the Main scene, 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

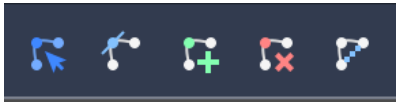
Set the `Wait Time` property of each of the `Timer` nodes as follows:

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

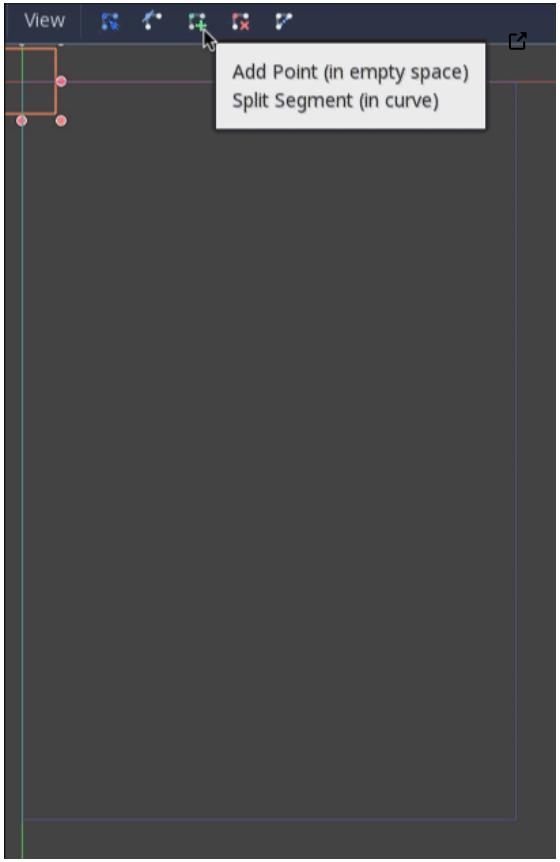
In addition, set the `One Shot` property of `StartTimer` to "On". This will cause it to run once, rather than repeatedly on a loop.

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.



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, like a car, so we can tell it to travel a random distance along the path, then ask it to tell us where it is, and use that position as the point at which to spawn the mob.

Main script

Add a script to `Main`. At the top of the script, we use `export (PackedScene) var Mob` to allow us to choose the Mob scene we want to instance -- we are declaring a new variable, Mob, of the type (PackedScene) which refers to another scene we have already created. The variable is exported so we can define it from the inspector.

We also declare the score variable and run `randomize()` at startup, which generates a new random seed each time the game is run.

```
extends Node2D

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 node and connect the `hit` signal by selecting the Node tab on the right side as before. 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, which will stop `ScoreTimer` timer increasing, and stop `MobTimer`, which is what will spawn new mobs, when the player object tells the main scene that it has been hit.

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

Now we create a function to start a new game. We set the score to 0, tell the player object to run its start function (which we defined earlier), then run the `StartTimer`, which will count down for two seconds then start the mobs spawning.

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

Add the new_game function to _ready().

```
func _ready():
  randomize()
  new_game()
```

Timer nodes have a signal called `timeout()`, which is what will tell the game that the timer has completed. So now we 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. Remember that we set `StartTimer` to run once (one shot) whereas the others continue to repeat and send signals each time they expire.

```
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:

```
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)
```

Note: 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.

The game should now run. You may need to change the main scene from Player.tscn to Main.tscn in Project -> Project Settings -> Application -> Run -> Main Scene

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.

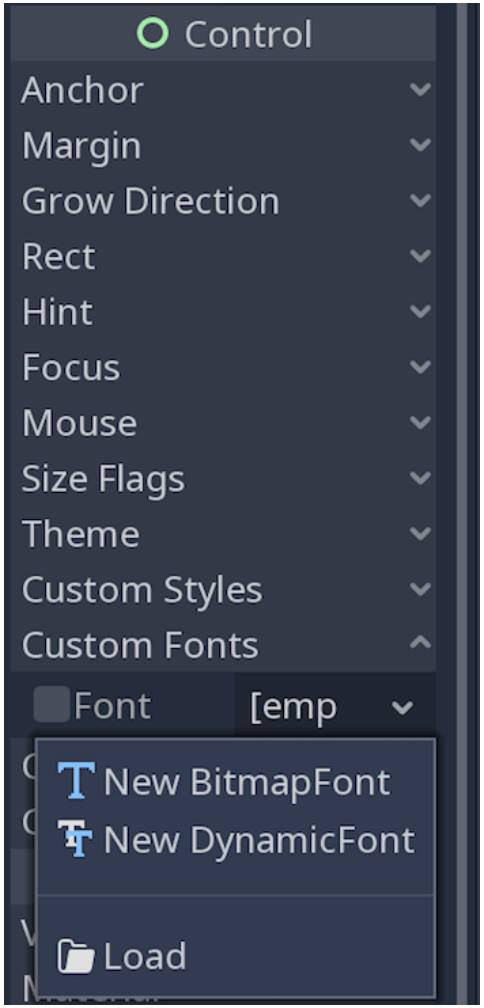
The basic node for UI elements is Control. To create our UI, we'll use two types of Control nodes: Label and Button.

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

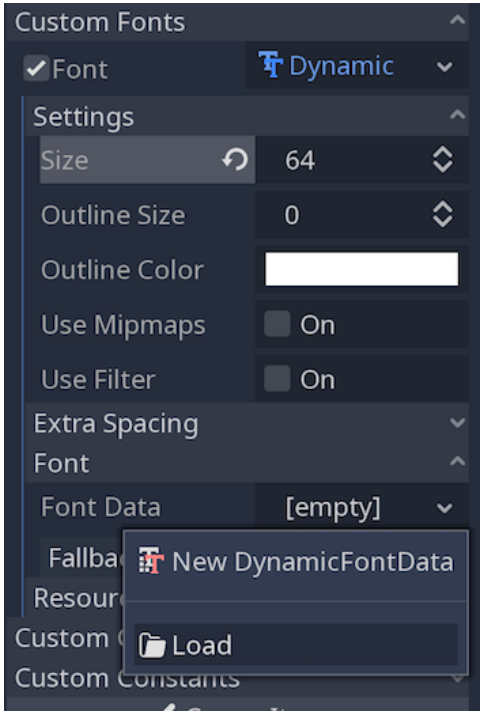
- Label named `ScoreLabel`.
- Label named `MessageLabel`.
- Button named `StartButton`.
- Timer named `MessageTimer`.

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:

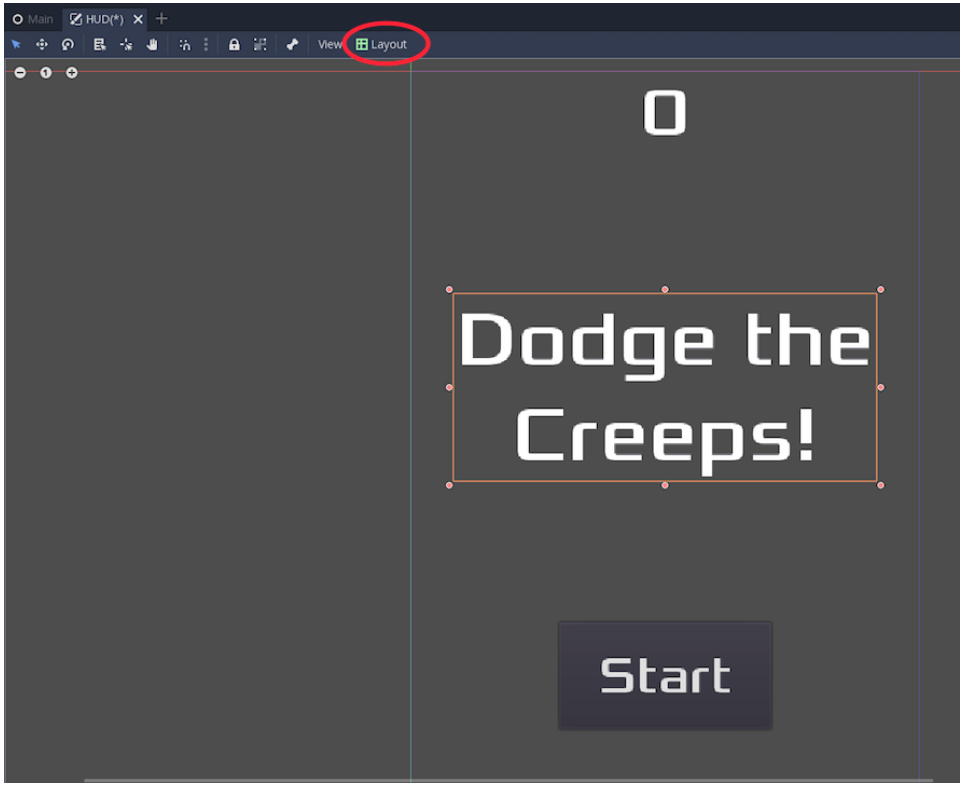
1. Under "Custom Fonts", choose "New DynamicFont"



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.



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`:

```
extends CanvasLayer

signal start_game
```

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

```
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".

```
func show_game_over():
  show_message("Game Over")
  yield($MessageTimer, "timeout")
  $MessageLabel.text = "Dodge the\nCreeps!"
  $MessageLabel.show()
  $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 show the "Start" button.

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

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

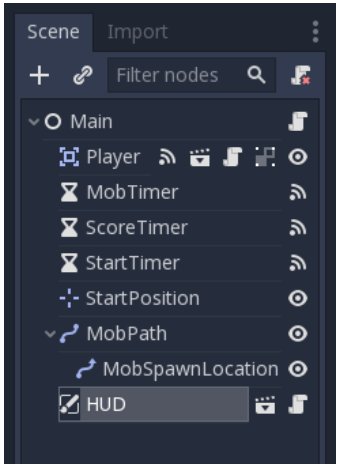
Connect the `timeout()` signal of `MessageTimer` and the `pressed()` signal of `StartButton`.

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

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

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. Remove `new_game()` from the `_ready()` function in `Main.gd` as we will start a new game with the button now.

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

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

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

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

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

```
$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`.

Finishing up

We have now completed all the functionality for our game. Feel free to expand the gameplay with your own ideas.

Further Enhancements

Background

The default gray background is not very appealing, so let's change its color. One way to do this is to use a :ref:`ColorRect <class_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 :ref:`AudioStreamPlayer <class_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.

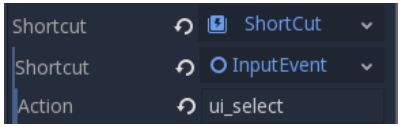
To play the music, add `$Music.play()` in the `new_game()` function and `$Music.stop()` in the `game_over()` function.

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.

ToDo:

Despawn mobs on death Stop score proceeding after fast game over -- see original build Power ups