Lesson #3

Today's Game: Pinball

What we will learn

- Resize our game window
- Creating a ball that has gravity applied to it
- Create Bumpers that will Score points and Emit Sounds
- Create flippers that will rotate around a pivot
- Creating an amount of lives for our player

Pseudocode

We will create a list of objects that will be required for the project and list what they do.

- Flippers (2) ○ Will rotate when action is pressed and stay at maximum level until released.
    - We will have two copies for the left and right sides with the ability to create duplicates if we wish.

- Pinball ○ Will react to gravity using a rigid body property ○ Will have the ability to respawn on an object assigned as it's spawn point.
    - Will have properties added that will make its physics feel like that of the real-world counterpart.

- Bumpers
    - Upon contact with the pinball will change colour and emit a sound.
    - Will contain physics properties that make collision with it bouncier.

- Game Manager
    - Will Keep track of lives, score, and other game properties

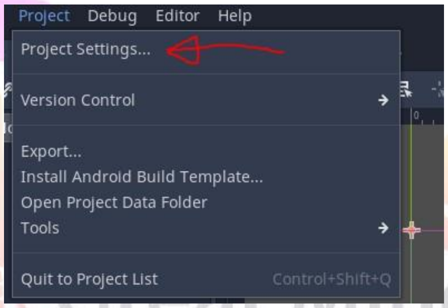
Setting up our Scene

Note

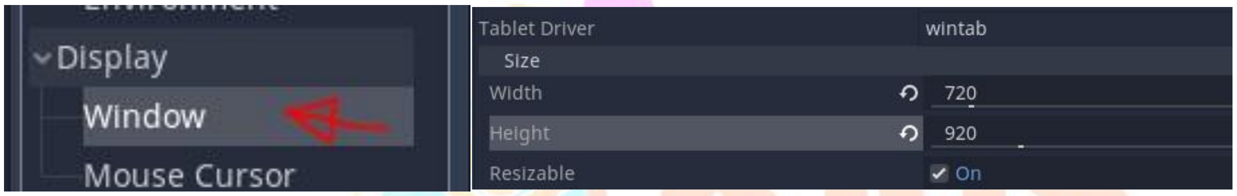
We will be skipping over details for creating the project and scene steps in this step. If you need a refresher as to how to create a new project and scene, please see the first instruction of the Brick Breaker and Air Hockey projects.

Additionally, most concepts that have already been mentioned and explained in previous manuals will be instructed, however with not as much explaination or visuals.

1. First let's set up our Project and name it 'Pinball Project'. After we do this, we will also go ahead by making a 2D scene which we will call 'Main Scene'.
2. As pinball machines are usually vertical in nature, we are going to make a bigger screen for our play area. To do this we will click on **Project** at the top of our window and then go into **Project settings**.



3. Scroll down through the options on the left until you find the **Display** section and then select **Window**. In the table to the right of this we will see an option for size which is likely to be currently a width of 1024 and a height of 600. Let's set our width to be 720 and our height to be 920.

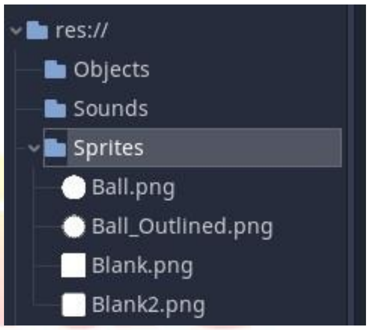


Tablet Driver wintab
Size
Width 720
Height 920
Resizable On

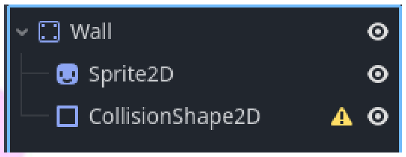
We can now see that the blue outline in our screen space editor has changed its dimensions with what our scene will be. If we were to hit play, we would also notice that our play window is different.

Part 1: Setting (up) the pinball table.

1. We will now want to Create some walls to go around our pinball table and so to do this we must first import some of the images from our folder. While we are at it let's create in Godot a folder for our Objects, Sounds, and Sprites.

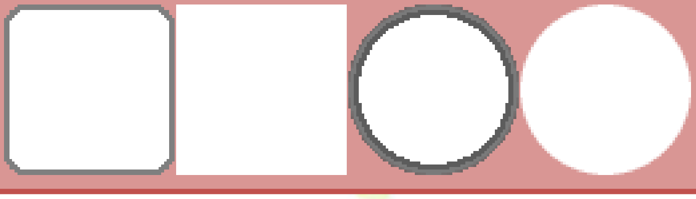

res://
Objects
Sounds
Sprites
Ball.png
Ball_Outlined.png
Blank.png
Blank2.png

2. Now let's create an object scene called wall. For this we will Create a Node with a base being a **Static Body**, and contain a **Sprite2D**, and **CollisionShape2D.** Remember to also rename our main node to be 'Wall'.

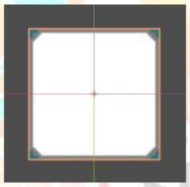

Wall
Sprite2D
CollisionShape2D

Note

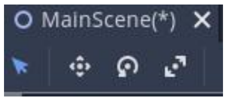
Students will be provided with textures for each class in a folder. Provi ded here as well are the textures that will be used:



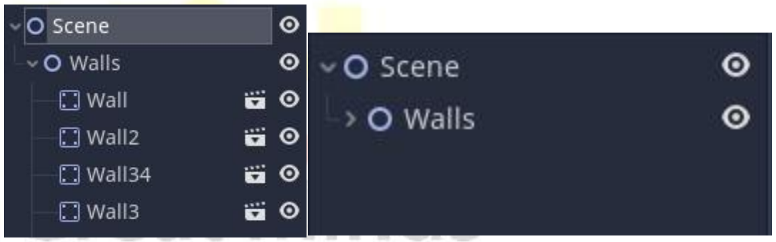
3. Once we have made this let's add a texture to the sprite ('Blank' or 'Blank2') and put a rectangular collision shape around it.



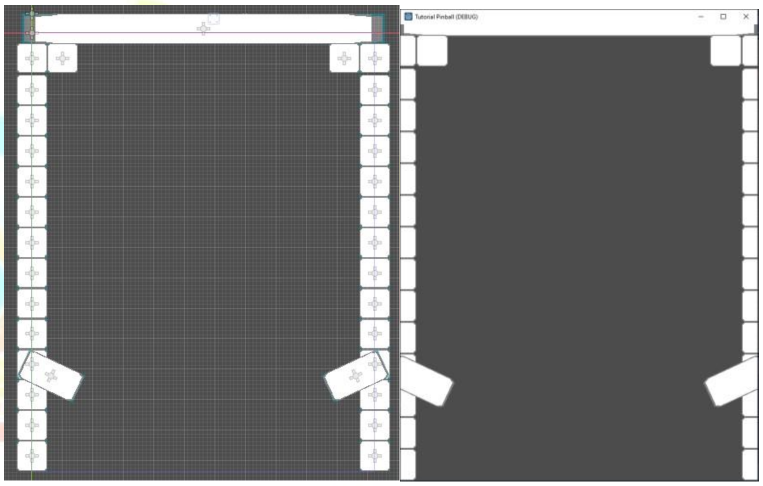
4. Now that we have a static Body Object lets hit save by using **CTRL +S** and then go back to our scene and place a few of these around in our scene to look more like a pinball table. Try stretching and rotating some of the walls using the transform tools found right under the scene's name. Overall, we want to make the sides and top inescapable for a ball.



5. We can also organize our walls by creating an empty Node2D, rename it to 'Walls' and from there we can add our walls inside of it. We can then even collapse the list if we do not wish for them to fill up the scene hierarchy in a visual way.



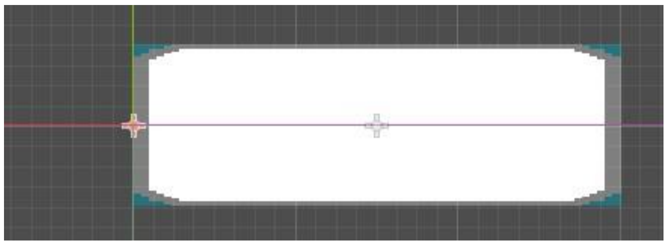
You can change it to your liking but for reference here is one design.



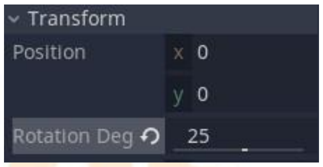
In Editor In Game

Part 2: Setting up Flippers

1. For our next step we need to set up our flippers. First, we will Create a Scene object and rename the base node To be Flipper_L and change the node type to be a **Charcterbody2D**. Make sure as well to set the gravity scale to be 0 so it stays in place. We will do this in the script
2. We will next add a Sprite and ShapeCollider2D. but instead of putting it on top of the center point, we will offset the sprite and collision. This is so that we can pivot the shape around the center. To test this, try rotating the base node and see how the flipper movers around the center point.



3. Because pinball flippers are by default at a lowered position, also make sure to rotate your left flipper facing downward (try 25 degrees)

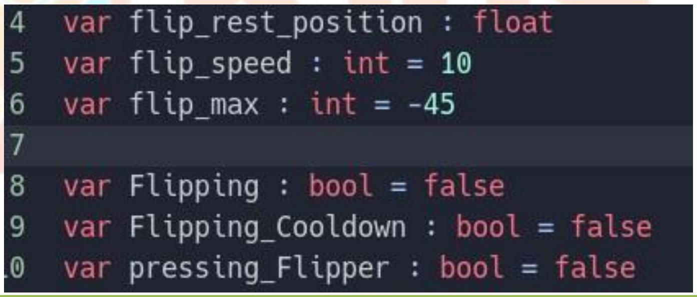


Transform
Position x 0
y 0
Rotation Deg 25

Note

The code section for the flippers will very likely be the hardest part of the lesson for students so make sure to take your time with the explanations in this part.

4. Now let's create a Script and attach it to our Characterbody naming it 'Left_Flipper'. To make our flippers go we need to know a few things. Pinball flippers can only go up by a certain amount. The flippers go up by a certain speed so we will also need to know by what amount that will also be. We will need to save the current rotation we are at as this will be the resting position for our flipper. We also need to check when the player is holding down a particular key. Finally, we will need 2 bools to check if our flipper is currently flipping or if it is receding back to the rest position. Let's start by creating these variables.



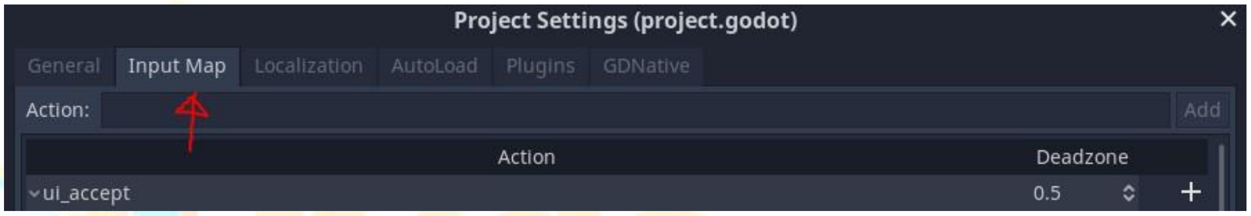
```
var flip_rest_position : float
var flip_speed : int = 10
var flip_max : int = -45

var Flipping : bool = false
var Flipping_Cooldown : bool = false
var pressing_Flipper : bool = false
```

" Why do we always put our numbers as variables?"

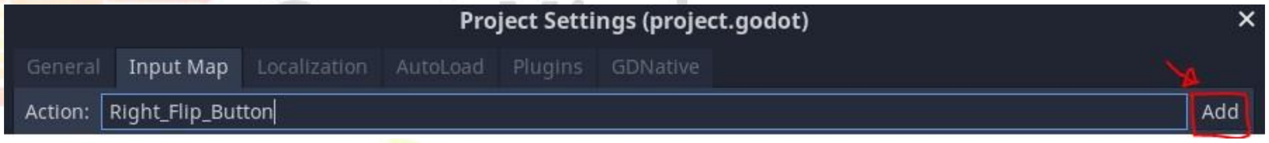
Variables are easier to update and can apply to multiple area such as our flip speed.

5. The next thing we will need to do is create a physics Process function to monitor every frame and check or update our rotation and if we are flipping.
To start off, our first logical though is probably when I press a specific key the flipper will start flipping. This is a great place to start, but thinking ahead we need to know as well when the player released the button. As it stands now, Godot only has When_Key_Pressed and not When_Key_Released. What does have both a press and release function however is **Action**.

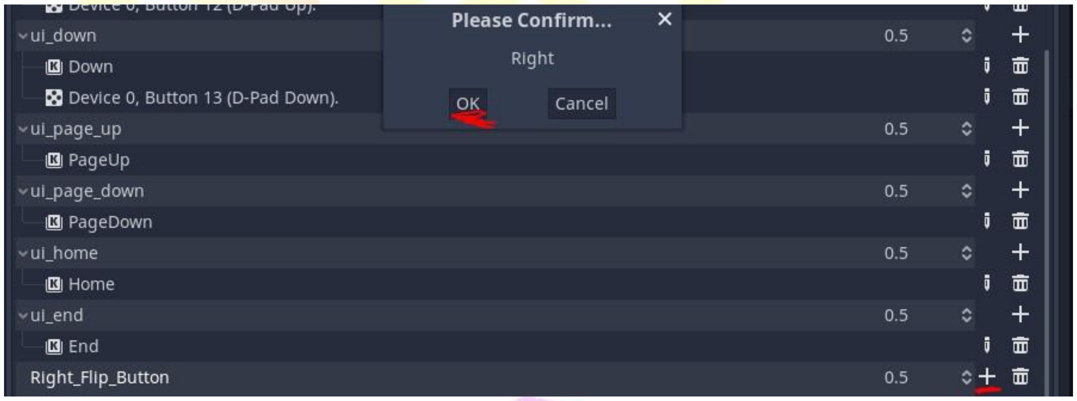
To create an action we need to **Project Settings** and go into to the **Input map**.



6. In the Section that says **Actions** we will write and add 2 new actions named **Left_Flip_Button** and **Right_Flip_Button**.

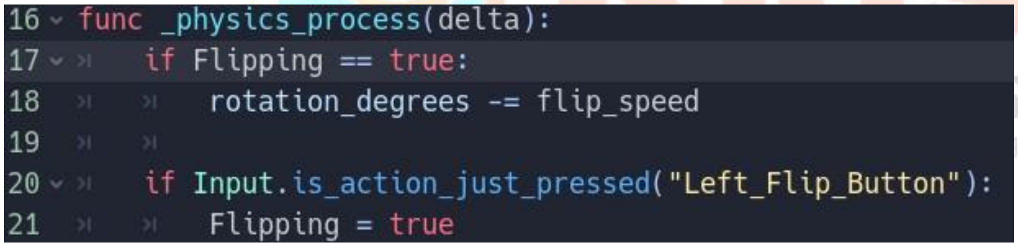


7. Now we need to map these actions to a particular key. Scroll down to where your particular action is and hit the **+** button is. You will next be promoted as to what type of input the action is. Hit **Key** and then input the key that you will want the input to be. Once that has been done, hit **OK**.



8. Repeat the previous step for your other flip button.

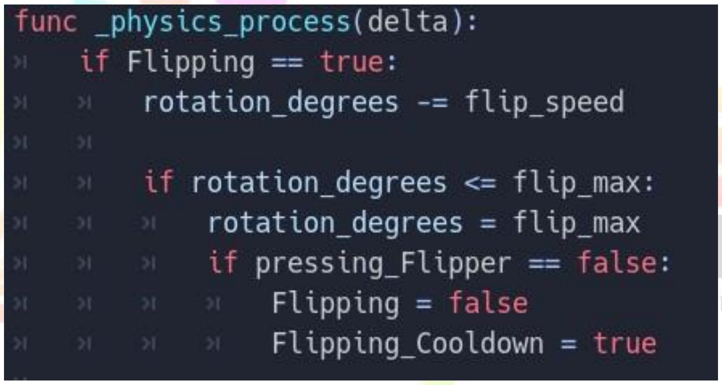
9. After you have done the previous step, lets return to our code. We can now use the is_action_pressed function and we will say that when we press a particular action, we will set flipping to be true.



```
func _physics_process(delta):
	if Flipping == true:
		rotation_degrees -= flip_speed

	if Input.is_action_just_pressed("Left_Flip_Button"):
		Flipping = true
```

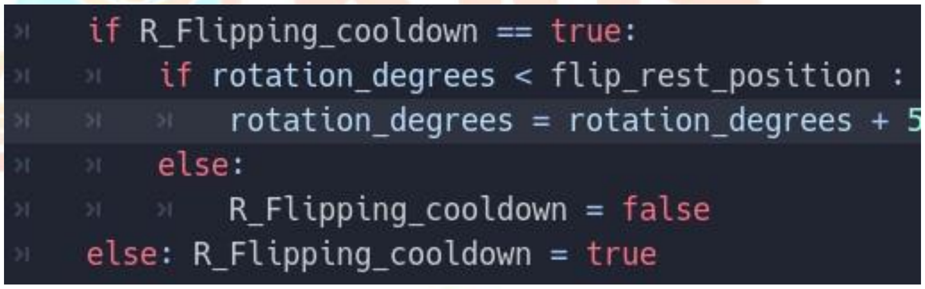
10. If we now go into our scene and add in our flipper. We will see that when we press our button the flipper will start to rotate. However, at the moment, the flipper moves nonstop and so we will need to add in some more logic to stop it once it has reached the end of its threshold.

What we do first is check that if the rotation surpasses the flip max, we will set it to be the flip_max value. Next, we will see if the key is being pressed or not by saying if the pressing_flipper bool is false (which we will go more into in a second), we will set flipping to be false and start our cooldown



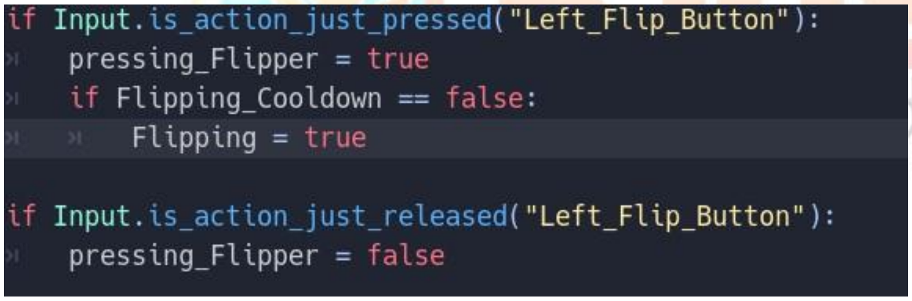
```
func _physics_process(delta):
	if Flipping == true:
		rotation_degrees -= flip_speed
		
		if rotation_degrees <= flip_max:
			rotation_degrees = flip_max
			if pressing_Flipper == false:
				Flipping = false
				Flipping_Cooldown = true
```

11. What we will do underneath the code above is now bring back our rotation to it's original state by checking if our cooldown is on (which it will be if flipping = false). To do this we check if the rotation is less than the resting position and if it is under the mark, we will just set it to it's resting position.



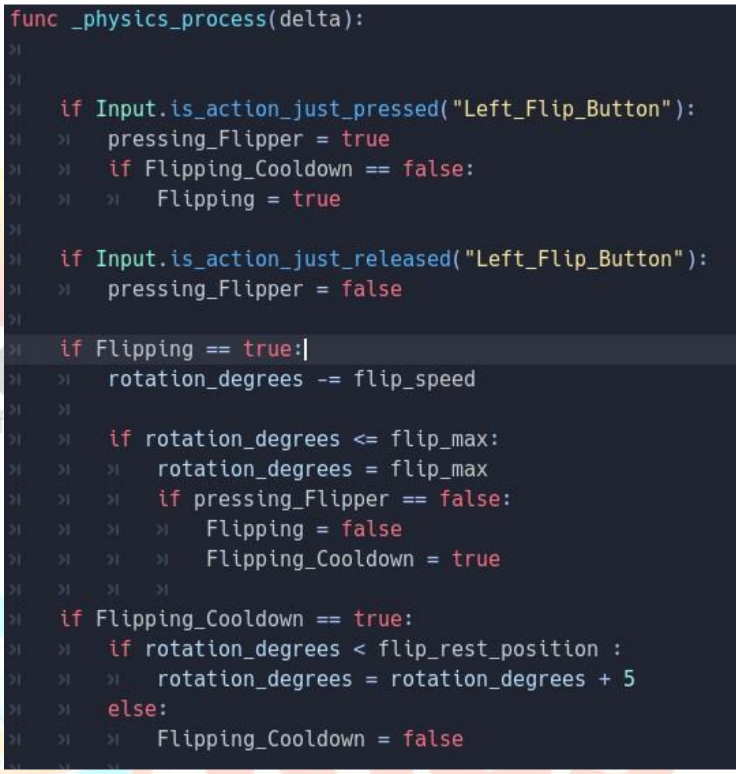
```
	if R_Flipping_cooldown == true:
		if rotation_degrees < flip_rest_position :
			rotation_degrees = rotation_degrees + 5
		else:
			R_Flipping_cooldown = false
	else: R_Flipping_cooldown = true
```

12. We will finally make some adjustments to our Input functions in order to set our pressing_flippers correctly and also making a check to see if the cooldown is occurring. We will also cut and paste our input to be at the top of the function.



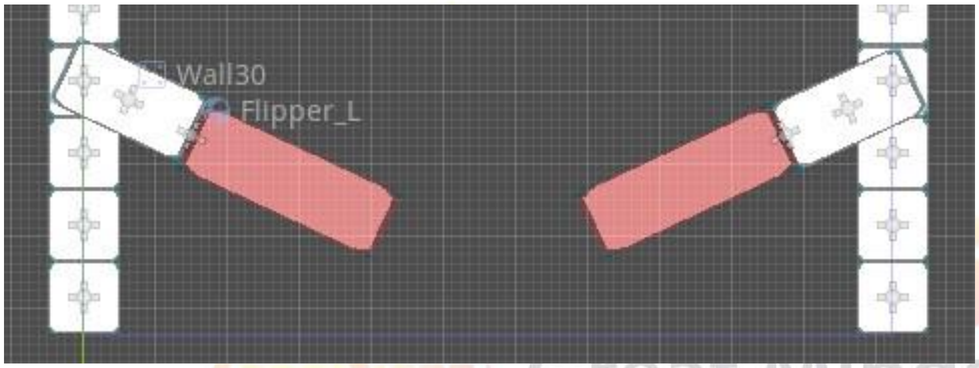
```
if Input.is_action_just_pressed("Left_Flip_Button"):
	pressing_Flipper = true
	if Flipping_Cooldown == false:
		Flipping = true

if Input.is_action_just_released("Left_Flip_Button"):
	pressing_Flipper = false
```

Overall, our physics process function should look something like this now:

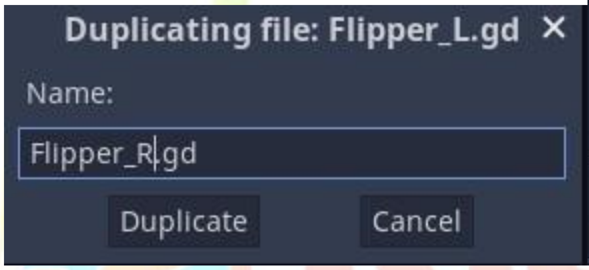


```
func _physics_process(delta):


	if Input.is_action_just_pressed("Left_Flip_Button"):
		pressing_Flipper = true
		if Flipping_Cooldown == false:
			Flipping = true

	if Input.is_action_just_released("Left_Flip_Button"):
		pressing_Flipper = false

	if Flipping == true:
		rotation_degrees -= flip_speed

		if rotation_degrees <= flip_max:
			rotation_degrees = flip_max
			if pressing_Flipper == false:
				Flipping = false
				Flipping_Cooldown = true

	if Flipping_Cooldown == true:
		if rotation_degrees < flip_rest_position :
			rotation_degrees = rotation_degrees + 5
		else:
			Flipping_Cooldown = false
```

13. Now that our left flipper has been made, we will need to make our right flipper. We can actually **Duplicate our object scenes** by right clicking on them in the 'filesystem' tab, hitting **Duplicate** and changing the name of the object (such as Flipper_R). Try your best to make a flipper that will look like a mirrored version of your left flipper when they are both placed in the scene somewhat like this:

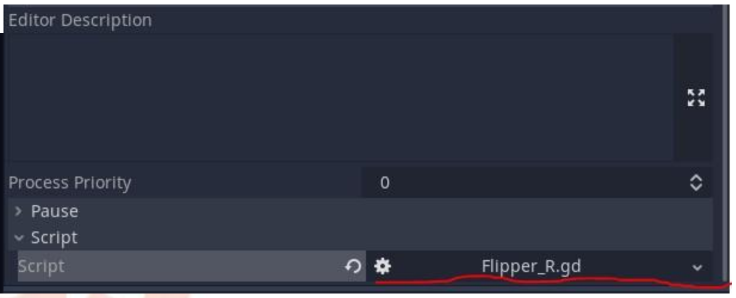


14. We can duplicate objects without much of a problem but for our pinball game we will need to modify the code in a couple of places for the right flipper. To do this let's go into our filesystem and duplicate the script for the left flipper and attach it to our right flipper object.



Duplicating file: Flipper_L.gd

Name: Flipper_R.gd

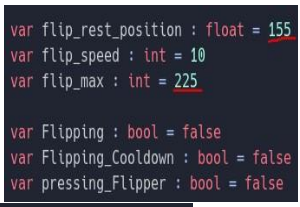


15. Because it's in another the right flipper is rotating in another direction, we will need to change some variables and also input to our right flipper script. Here are the main things we will change (assuming we have changed the rotation to 155):

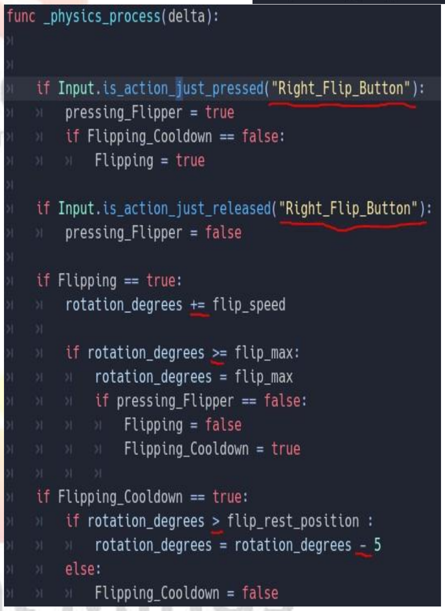
Partners

If there are partners this is a great time for them to switch up so that they can both do a flipper.

Test it out to see if everything is working properly. If done correctly, we will now be able to move our flippers with our input of the left and right arrow keys.



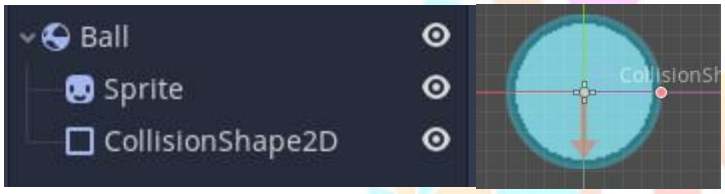
```
var flip_rest_position : float = 155
var flip_speed : int = 10
var flip_max : int = 225

var Flipping : bool = false
var Flipping_Cooldown : bool = false
var pressing_Flipper : bool = false
```



```
func _physics_process(delta):


	if Input.is_action_just_pressed("Right_Flip_Button"):
		pressing_Flipper = true
		if Flipping_Cooldown == false:
			Flipping = true

	if Input.is_action_just_released("Right_Flip_Button"):
		pressing_Flipper = false

	if Flipping == true:
		rotation_degrees += flip_speed

		if rotation_degrees >= flip_max:
			rotation_degrees = flip_max
			if pressing_Flipper == false:
				Flipping = false
				Flipping_Cooldown = true

	if Flipping_Cooldown == true:
		if rotation_degrees > flip_rest_position :
			rotation_degrees = rotation_degrees - 5
		else:
			Flipping_Cooldown = false
```

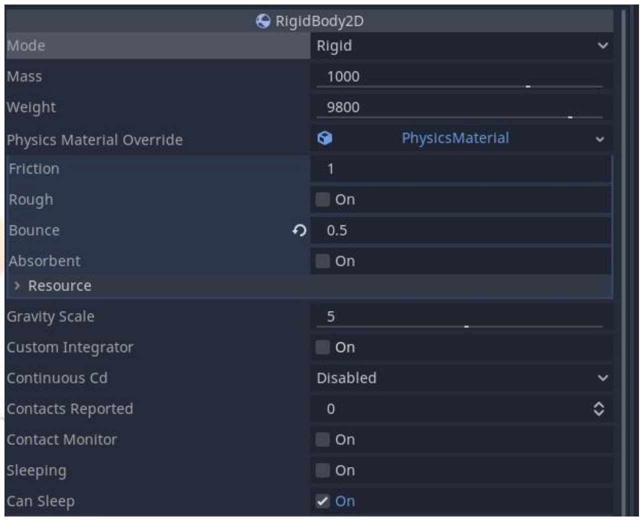
Part 3: Creating the ball

1. We will now create a new Scene object and name it 'Ball'. We will make our main node a **Rigidbody2D** and also add on a collisionShape2D along with a Sprite as well (which we can use either of the ball images provided to be our texture). Once we have applied a texture to the sprite, we will add a circle to be our shape collision and have it bordering our image.



Another thing we will also have to do is in our **CollisionShape2D** node is to turn off the feature labelled '**Oneway collision**'. If we do not do this our ball will fall through our walls and flippers.

In our rigidbody section of our inspector, we will want to change **Mass** to be <u>1000</u> and **Weight** to be <u>9800</u>. In the **physics Material Override** section, we will click where it says empty and create a new physics property with a **Bounce** value that equals <u>0.5</u>. Finally, we will set our **Gravity Scale** to be 5. Overall, the ridigbody inspector should look like this:

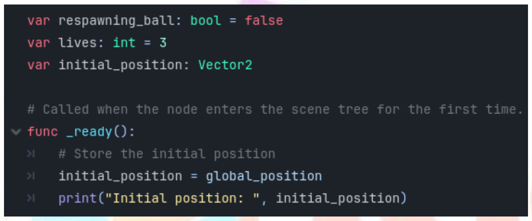


Mode	Rigid
Mass	1000
Weight	9800
Physics Material Override	PhysicsMaterial
Friction	1
Rough	On
Bounce	0.5
Absorbent	On
Gravity Scale	5
Custom Integrator	On
Continuous Cd	Disabled
Contacts Reported	0
Contact Monitor	On
Sleeping	On
Can Sleep	✓ On

2. If we place the ball in the scene now, we will be able to have it collide with the objects within it and even be able to launch it with our flippers. However, before we move to our next section let's add a script to our ball so we can have it respawn on a to a particular location.

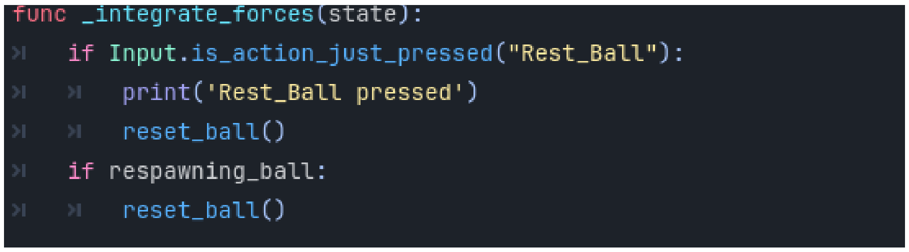
   Once we have made our script we will start off by creating a variable that checks if the ball is currently respawning which we will set to false. We will also create a variable for how many lives/tries our pinball has before we get a game over.

   The way we will create our respawning system is by making the ball a child of another object that we create and so throughout this script we will be referencing that spawn point using global_position So in our ready function, we will make our objects position to initial_position = global_position.



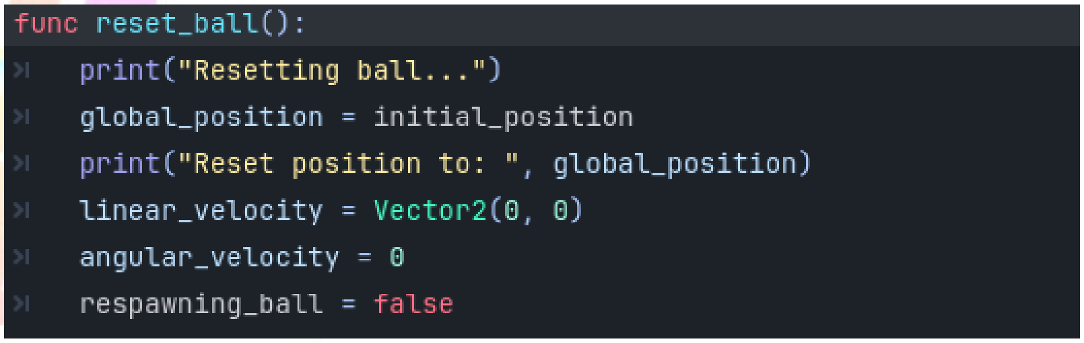
```
var respawning_ball: bool = false
var lives: int = 3
var initial_position: Vector2

# Called when the node enters the scene tree for the first time.
func _ready():
	# Store the initial position
	initial_position = global_position
	print("Initial position: ", initial_position)
```

3. Because this is a rigidbody we will need to use the **Integrate Forces** function in order to move it. Inside the integrate forces we will put 2 separate conditions on how to reset our object to the respawn. One will be a simple key input and the other will be for if we wish to use function to signal that the ball should respawn.



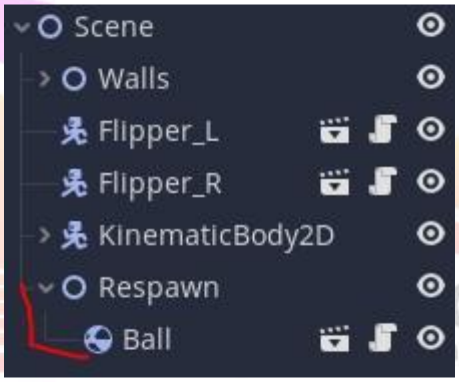
```
func _integrate_forces(state):
	if Input.is_action_just_pressed("Rest_Ball"):
		print('Rest_Ball pressed')
		reset_ball()
	if respawning_ball:
		reset_ball()
```

4. When we respawn our object, we will also want to reset the velocity that it is moving in so we will reset the linear and angular velocity to be 0. We will create a function called Rest_ball() for this action.



```
func reset_ball():
	print("Resetting ball...")
	global_position = initial_position
	print("Reset position to: ", global_position)
	linear_velocity = Vector2(0, 0)
	angular_velocity = 0
	respawning_ball = false
```

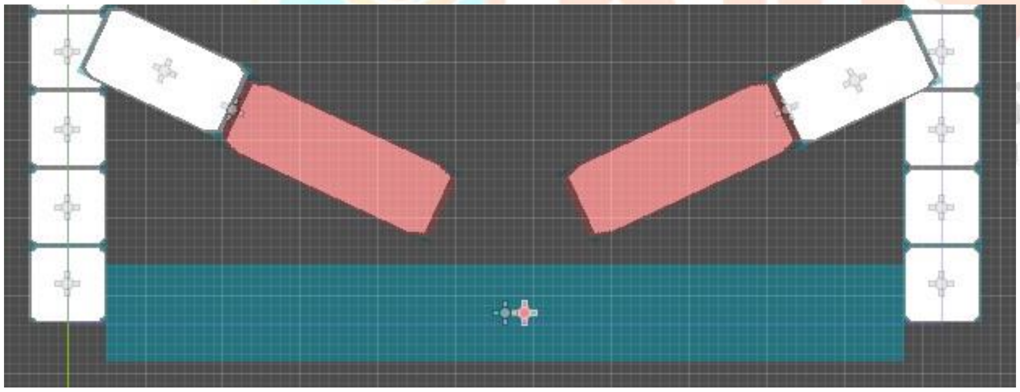
5. Now in our main scene lets add a new **Node 2D** and rename it 'Respawn' and to make the ball a child of it simply drag it overtop of the node in the scene hierarchy.



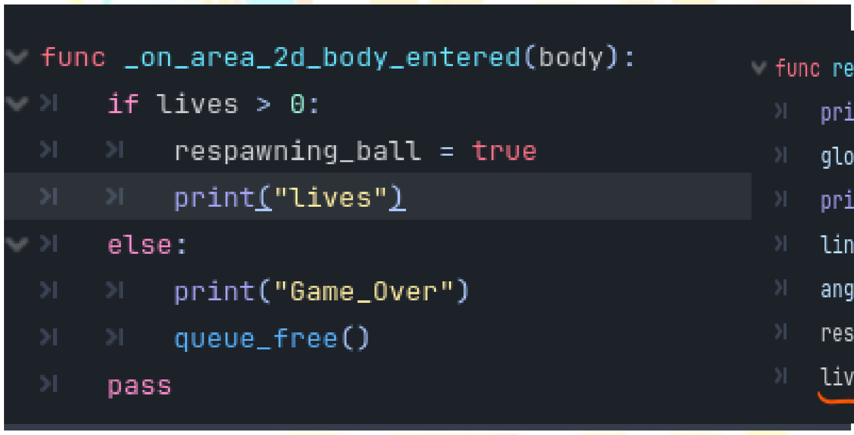
```
Scene
  Walls
  Flipper_L
  Flipper_R
  KinematicBody2D
  Respawn
    Ball
```

If we test it out now in our scene, the ball should respawn now wherever we put the respawn whenever we hit our selected key.

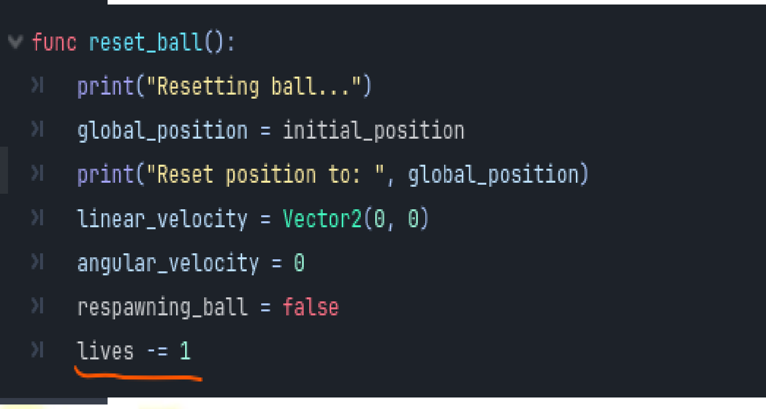
6. For the ball, we will make one last change in the form of an area 2D underneath board so if the ball goes out of play, it will respawn if it has enough lives. Once we make the area 2D we will add a rectangular collision shape at the bottom of the screen.



7. In the node tab of the Area 2D we will use the **Signal** 'body_entered' and attach that to the ball object. In the function now created in the ball script, we will say that if the score is more than 0 we will respawn the character. If that is not the case we will print 'Game Over' in the debug screen and also use queue_free() which will delete our ball (the deleting part is optional).



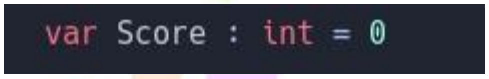
```
func _on_area_2d_body_entered(body):
	if lives > 0:
		respawning_ball = true
		print("lives")
	else:
		print("Game_Over")
		queue_free()
	pass
```



```
func reset_ball():
	print("Resetting ball...")
	global_position = initial_position
	print("Reset position to: ", global_position)
	linear_velocity = Vector2(0, 0)
	angular_velocity = 0
	respawning_ball = false
	lives -= 1
```

Part 4: Creating a game manager/Score

1. Now we will want to make a part of our game that controls our score. In our scene node, we will create a new script called 'Game_Manager'. For this lesson our Game Manager is very small, as we will only be keeping track of our score and updating the display. We will start by declaring a variable for score at and setting it to be 0.



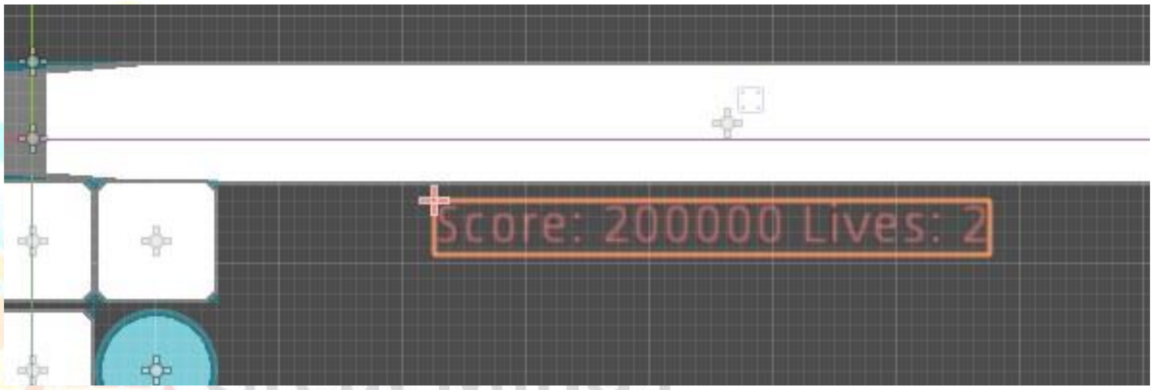
```
var Score : int = 0
```

2. Next, we will add a **label node** to our scene game object that will print our score. You can place it anywhere you want on the screen if it's clearly visible with a string of 20 characters. If you want, you can also scale the label for larger text. If you wish to change the colour of the text you can also use the modulate option in the inspector.

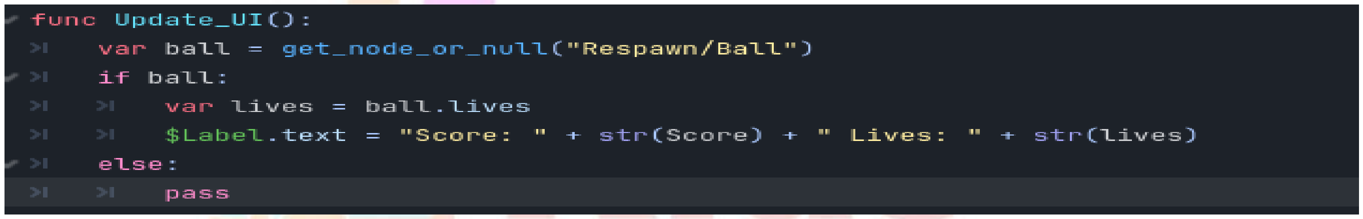
" Why might we want to make room for 20 characters/letters?"

If our score increases, we will want to be able to still have space without the UI going our of bounds.

.

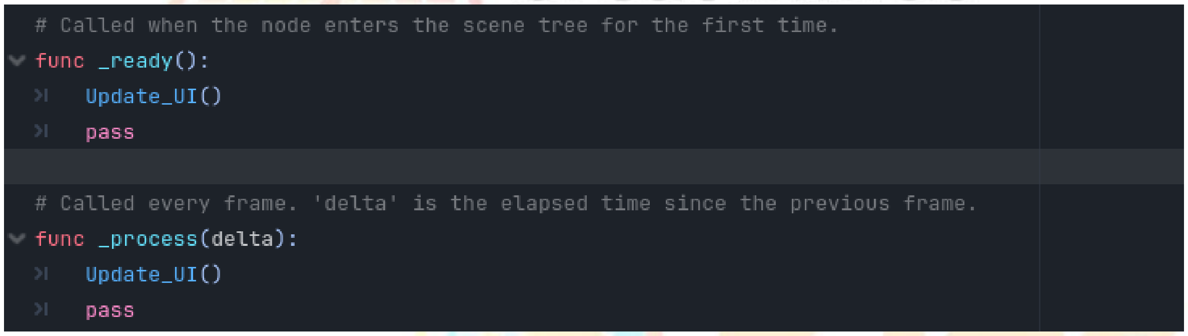


3. Next we will create a function that updates our score and lives every time it is called. We will call this function Update_UI and it will change the label text. To get our lives variable we will also use the **GetNode()** function which can come in handy for more of our projects in the future. **We will also add an if statement if there are no more lives left.**



```
func Update_UI():
	var ball = get_node_or_null("Respawn/Ball")
	if ball:
		var lives = ball.lives
		$Label.text = "Score: " + str(Score) + " Lives: " + str(lives)
	else:
		pass
```

Make sure to also call this function in our **Ready() and _ process()** function as well so we start with the proper data displayed.

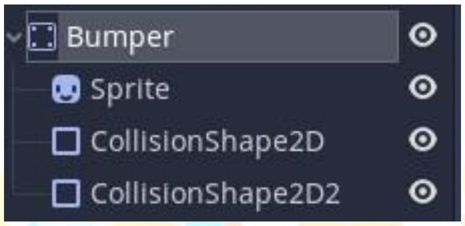


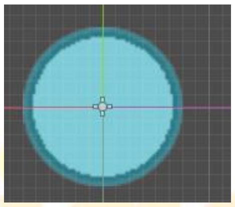
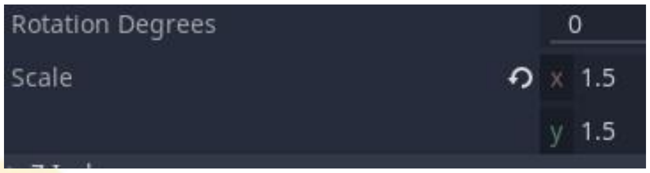
```
# Called when the node enters the scene tree for the first time.
func _ready():
	Update_UI()
	pass

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

4. Now we will need to update our Ball Script, in the rest_ball function lets add update_ui as well.

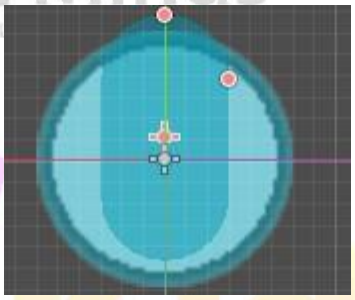
Part 5: Creating Bumpers

1. Now that we have a system that makes room for points let's create objects that score points once our pinball has collided with them. To do this we will create a new scene object called Bumper and make it a StaticBody2D.

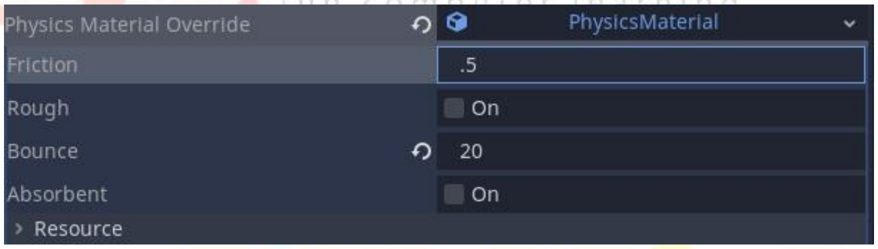




Rotation Degrees 0
Scale x 1.5 y 1.5

2. We will use a collision shape and sprite almost identical to the pinball but to make it different make sure to modulate the colour and scale. To make sure our ball does not get stuck to the roof of the bumper add in another round collider to the top so it will slide down. Try something like a capsule and have it ever so slightly above our circle collider.

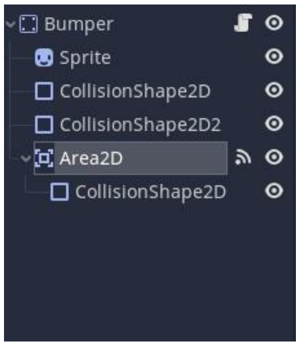


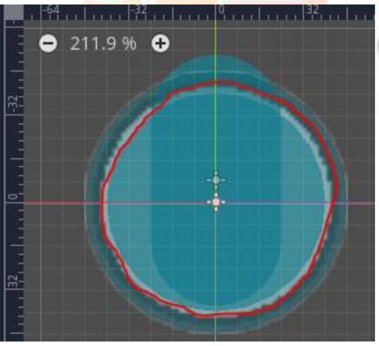
3. We also want our bumpers to be particularly bouncy, and so for this, we will create a physics profile in the **Physics Material Override** section of our inspector for the static body. We will change our friction to be 0.5 and our bounce to be 20.

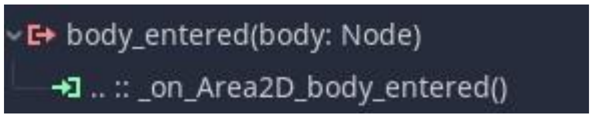


Physics Material Override	PhysicsMaterial
Friction	.5
Rough	On
Bounce	20
Absorbent	On
› Resource	

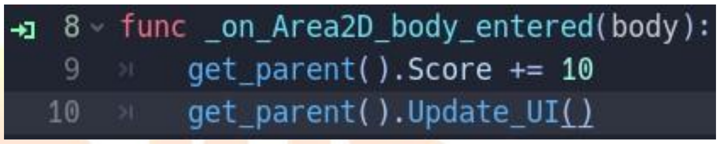
4. Now we want to play an effect when the ball hits and so we will make a Area2D and make it slightly bigger than the collision on our static body. This is because we want to use a body_entered function. This means we will also make a script on our main object and have the signal from the Area2D and of course name our script 'Bumper'.






body_entered(body: Node)
.. :: _on_Area2D_body_entered()

5. Because our Bumper objects will be children of our main scene object we can simply reference them to get our UI to update whenever we collide so we will add this as a line while also updating the points by 10.

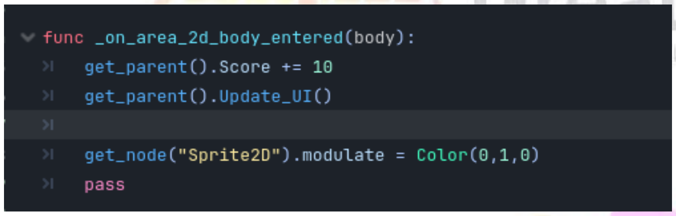


```
func _on_Area2D_body_entered(body):
	get_parent().Score += 10
	get_parent().Update_UI()
```

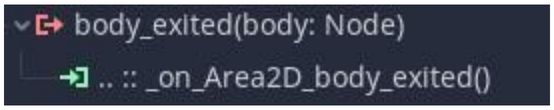
" Why is it important to update the score before using the update function?"

If we don't update after we update the score, the previous result will show up instead of the current one.

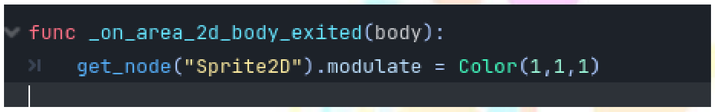
6. Our game is now functionally complete, but we might want to add some visual effects to our bumpers to look more like a collision happened. What we will do in our body entered function is change our modulation to switch the colour and then use a body_exit to revert our colour back.



```
func _on_area_2d_body_entered(body):
	get_parent().Score += 10
	get_parent().Update_UI()

	get_node("Sprite2D").modulate = Color(0,1,0)
	pass
```



```
body_exited(body: Node)
    .. :: _on_Area2D_body_exited()
```



```
func _on_area_2d_body_exited(body):
	get_node("Sprite2D").modulate = Color(1,1,1)
```

Our game should now have bumpers that change colour when the pinball collides with them.

Throughout this manual, there will be several different coloured text boxes and icons. Each of them corresponds to extra information about the steps at hand.

Changing table setups.

Encourage the students to make different types of pinball tables using different placements of bumpers, flippers, and walls.

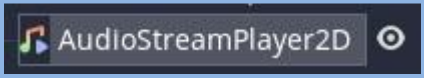
Adding sounds.

Pinball tables usually have an iconic sound to them most of the time and so what we can do is create sounds that will play once the ball hits a bumper.

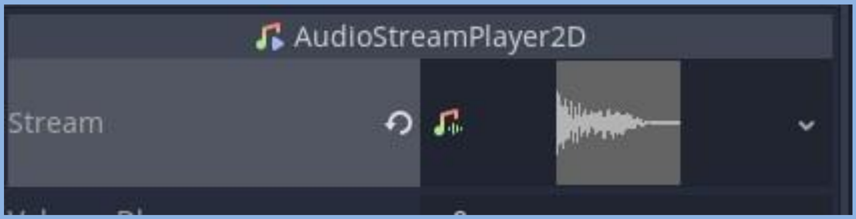
To do this let's import Some sounds in (sounds will be provided just like images, but you can also use wav or MP3 files that you have on hand. 'Freesounds' is a great resource for this). For now, I will be using the same sounds provided in the air hockey game.

For our bumper, we will need to do a few things:

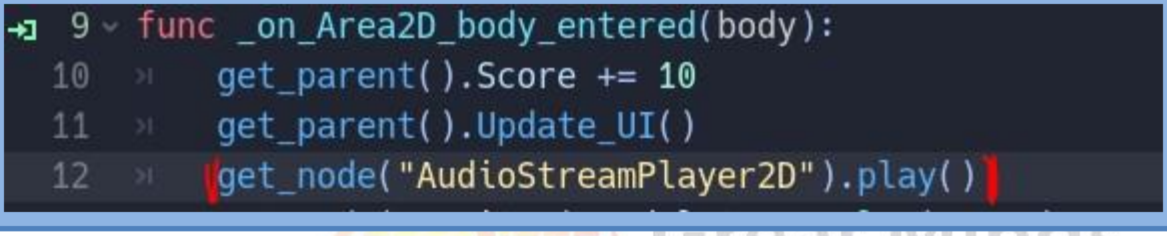
1. Add a **AudioStreamPlayer2D** as a child Node to the bumper.



2. Import in the sound effects that you wish to use (or provided).
3. In the inspector tab of the AudiostreamPlayer2D, set the empty stream to be the sound effect that you wish to be played.



4. Now in our Bumper script we will use the getnode function to access the audiostream player and use the Play feature to play our audio.



```
func _on_Area2D_body_entered(body):
	get_parent().Score += 10
	get_parent().Update_UI()
	get_node("AudioStreamPlayer2D").play()
```

5.

Adding sounds.

As an added challenge, see if students can add more sounds to the game such as making a noise when the ball respawns.