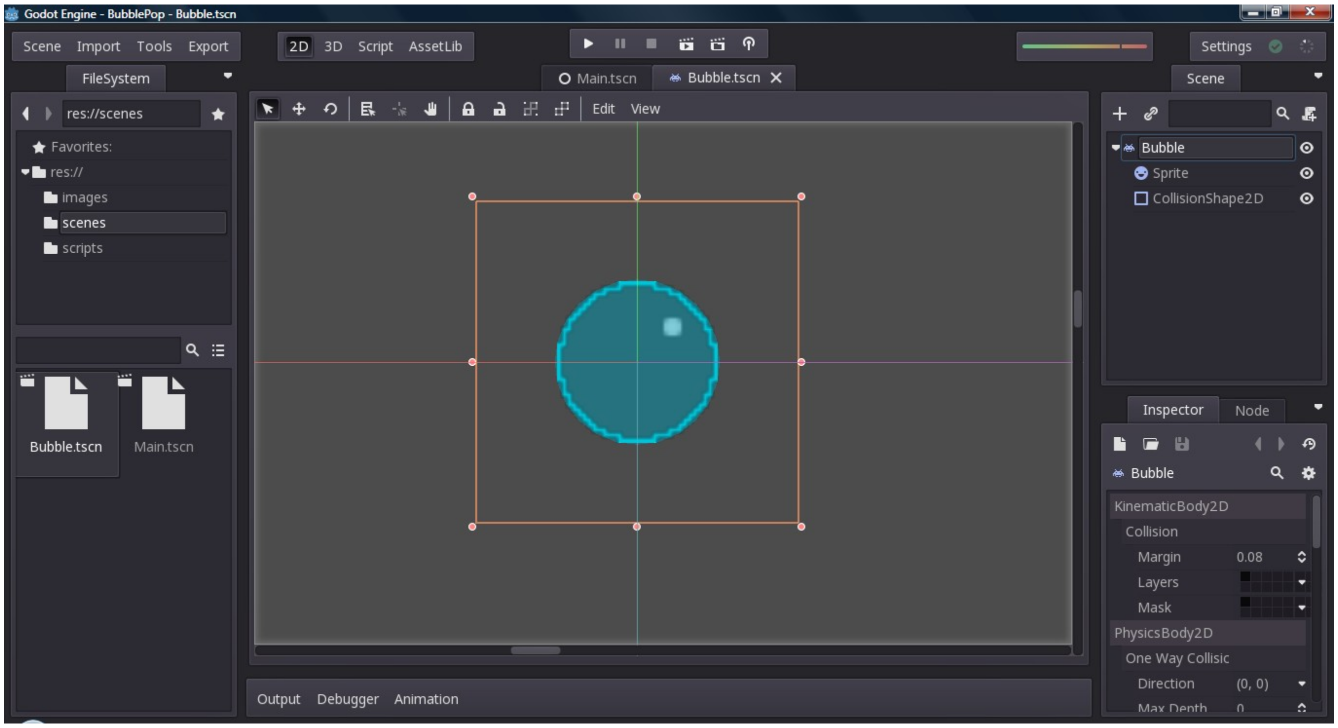
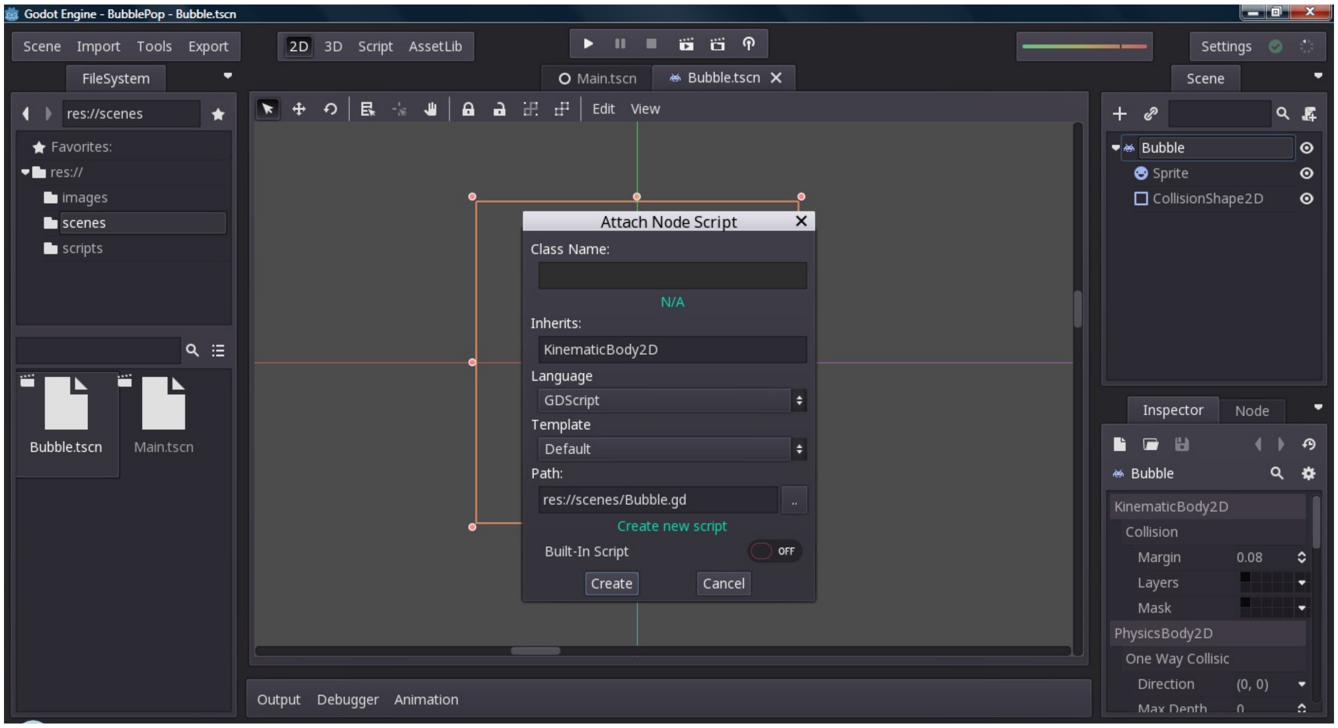
Godot 2D Game
Lesson 7: Object Physics

In this lesson, we will modify our bubble object so that it will bounce around on its own once it has spawned. Let's start by opening up our "Bubble" scene:



We will now attach a script to our bubble object. Right-click the "Bubble" node and choose "Attach Script" to open the attach script dialog:


Godot Engine - BubblePop - Bubble.tscn
Scene Import Tools Export
2D 3D Script AssetLib
Settings
FileSystem
Main.tscn
Bubble.tscn
Scene
Inspector
Node
Output Debugger Animation

Attach Node Script

Class Name:

N/A

Inherits: KinematicBody2D

Language: GDScript

Template: Default

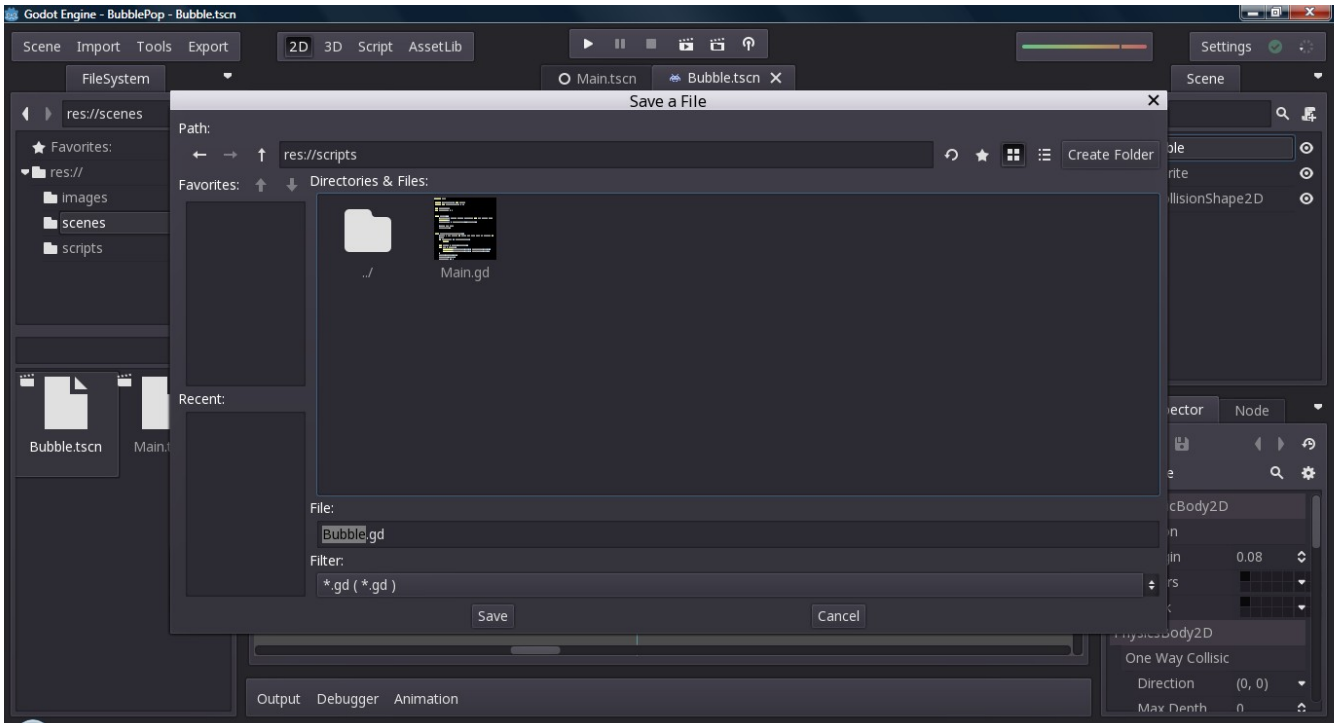
Path: res://scenes/Bubble.gd

Create new script

Built-In Script: OFF

Create   Cancel

Like before, we need to click the button beside the script path and navigate to our “scripts” folder:


Godot Engine - BubblePop - Bubble.tscn
Scene Import Tools Export
2D 3D Script AssetLib
Settings
FileSystem
Main.tscn
Bubble.tscn
Scene
Node
Output Debugger Animation

Save a File

Path: res://scripts

Create Folder

Favorites:

Directories & Files:

../

Main.gd

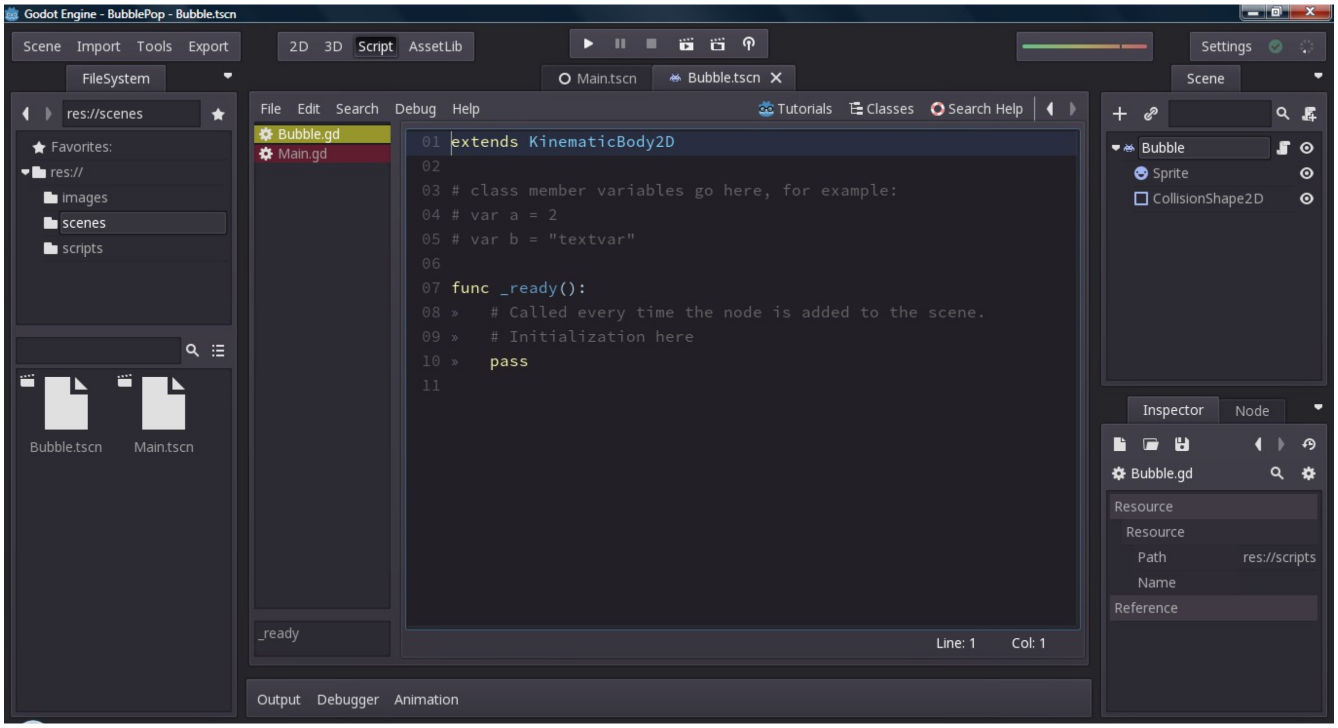
Recent:

File: Bubble.gd

Filter: *.gd (*.gd)

Save   Cancel

Now we can simply click “Save” and in the attach script dialog we will click “Create”. Then the script editor will open:



```
extends KinematicBody2D

# class member variables go here, for example:
# var a = 2
# var b = "textvar"

func _ready():
	# Called every time the node is added to the scene.
	# Initialization here
	pass
```

Resource
Path res://scripts
Name

Remember, we can use the list on the left side of the script editor to switch which script we are editing. Let’s start by defining some properties for our bubble:

```
export var speed = 64
export var hp = 100
```

The “speed” property will determine how quickly our bubbles move and the “hp” property will determine the initial and current HP of each bubble. In order to animate our bubble properly, we will need to keep track of its velocity as well. We will create a variable called “velocity” for this purpose:

```
export var speed = 64
export var hp = 100

var velocity
```

Next, we will randomly generate the initial velocity of the bubble:

```
func _ready():
	#Randomize the initial velocity of the bubble
	velocity = Vector2(speed, 0).rotated(deg2rad(rand_range(0, 360)))
```

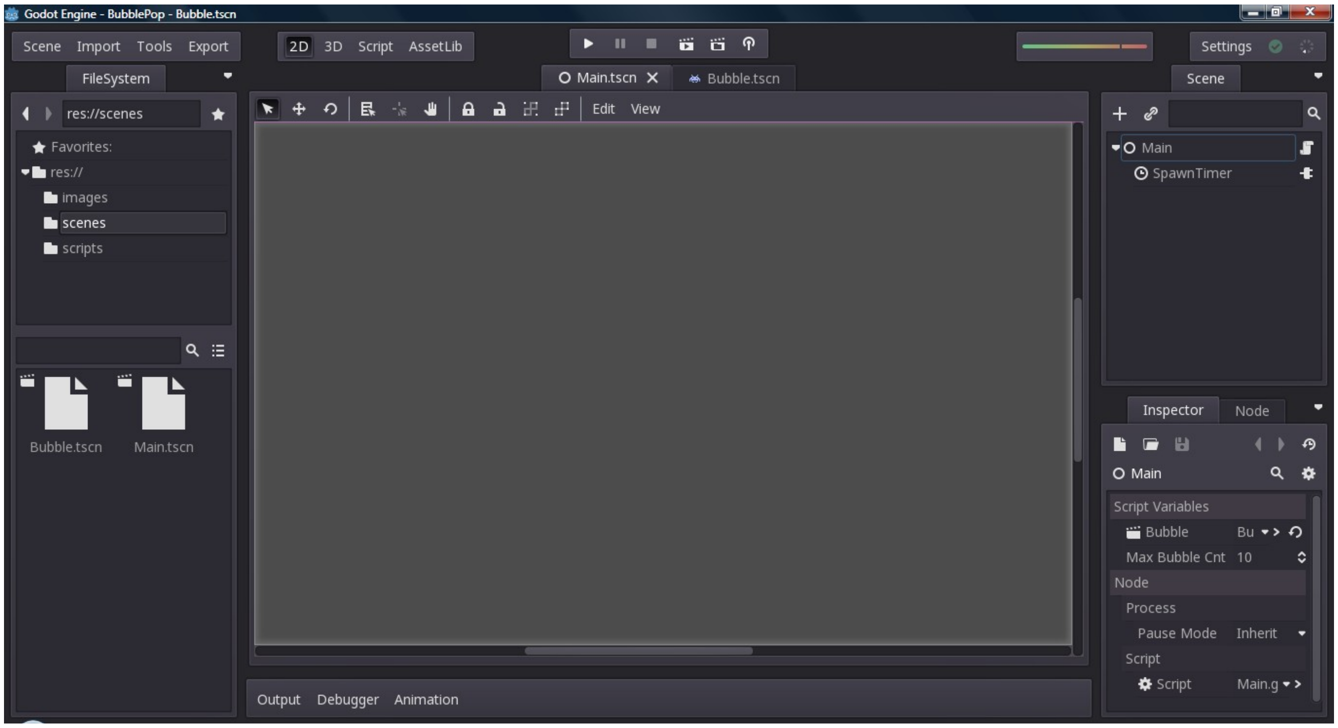
The method we used here is simple to understand. We start with a direction vector that would yield movement to the right along the X-axis. Then we rotate the direction vector by a random number of degrees between 0 and 360. However, we must also remember that the “rotated” method requires our angle to be in radians, not degrees. So we use the "deg2rad" function to convert our angle from degrees to radians. The result is a direction vector that is aimed in a random direction. However, we still need to write physics code to make our bubbles actually move. First, we need to enable physics processing in our "_ready" function:

```
func _ready():
	#Randomize the initial velocity of the bubble
	velocity = Vector2(speed, 0).rotated(deg2rad(rand_range(0, 360)))

	#Enable event processing
	set_fixed_process(true)
```

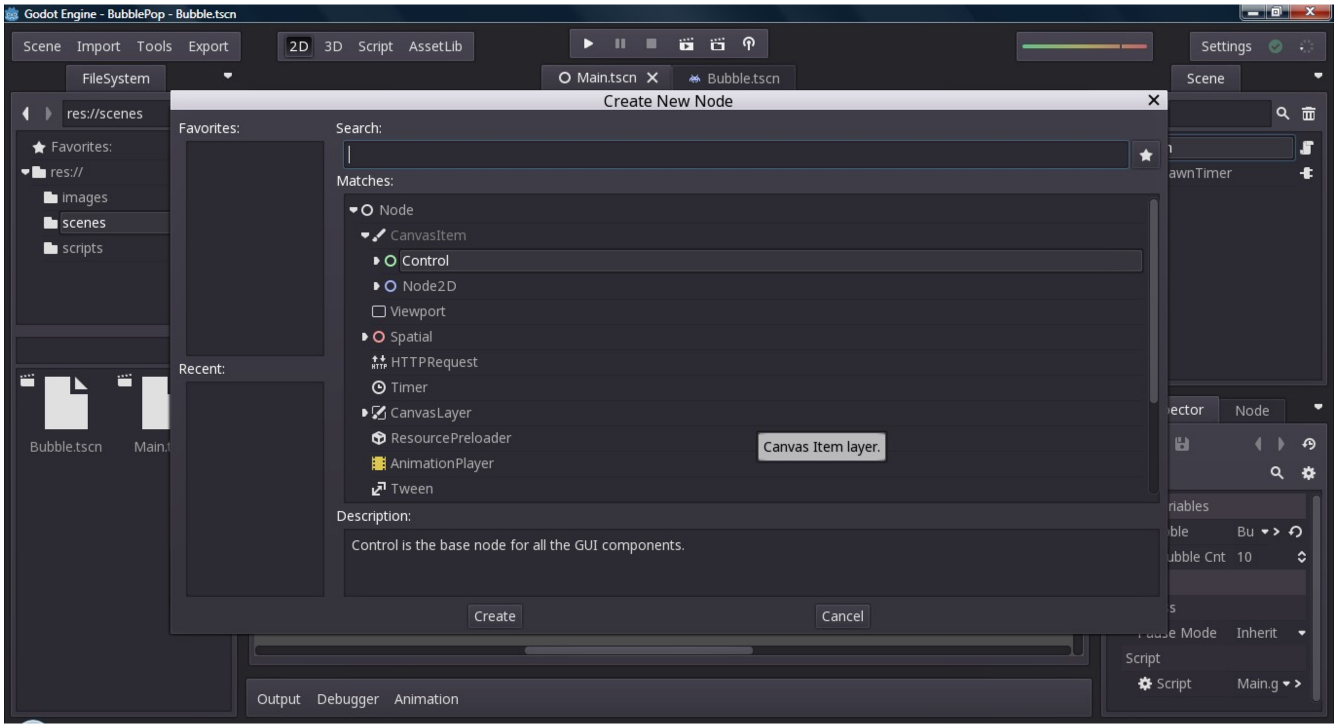
The "set_fixed_process" function is used to enable/disable fixed event processing. Fixed event processing occurs once per frame at a fixed interval. Now we need to write our "_fixed_process" function:

```
func _fixed_process(delta):
	#Update bubble position
	move(velocity * delta)
```

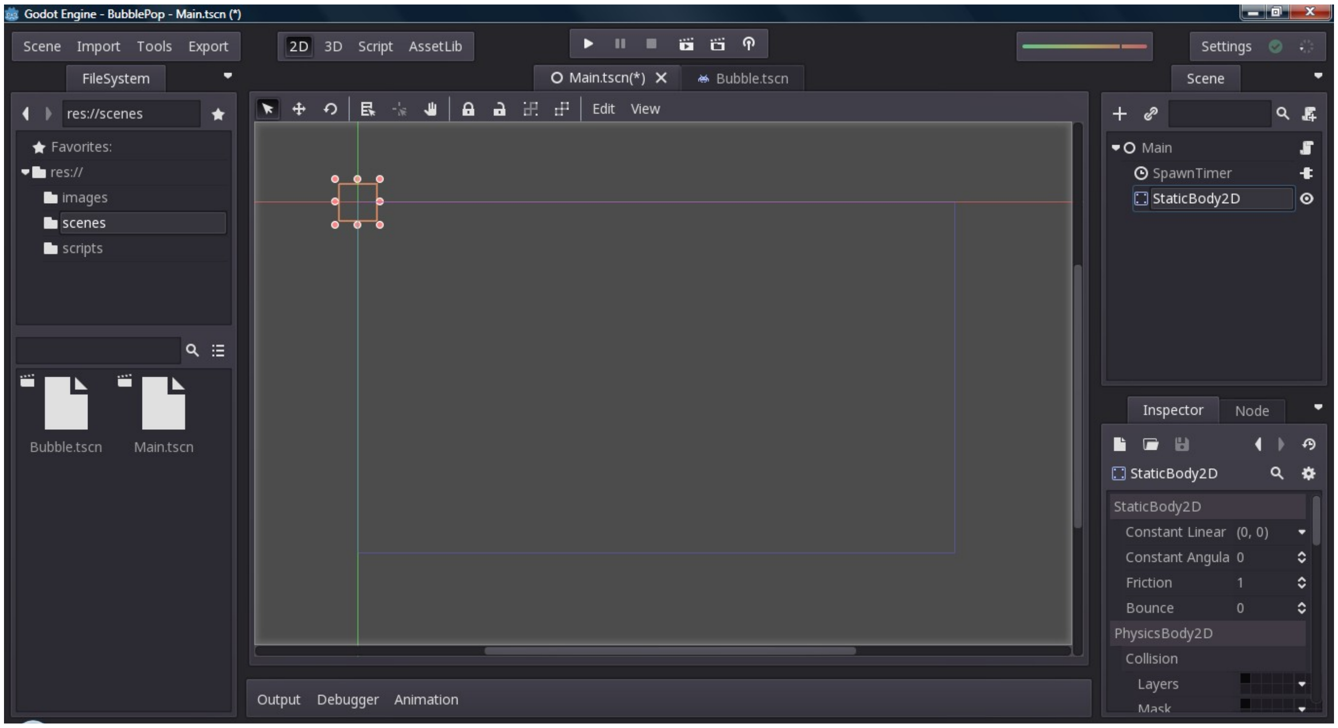
This is a good start for our physics code. Our new "_fixed_process" function will be automatically called once per frame at a fixed interval. The "move" function will move our bubble based on its current velocity. However, what is this "delta" value that we multiply our velocity by? "delta" is the amount of time that passed since the last frame. Multiplying our velocity by it will cause our bubble's movement to be smooth. Please note that there is a similar function called "move_and_slide" that automatically applies the delta time though. However, "move" and "move_and_slide" handle collisions differently. "move" causes the object to stop when it hits another object, whereas "move_and_slide" causes the object to slide along the edge of the other object. Typically, "move_and_slide" is better for objects that will be hitting the ground or floating platforms. If you run the game a few times, you will notice that a few things are off though. For starters, our bubbles go off the screen without stopping. And also, when 2 bubbles collide they don't bounce off each other either. We will start by fixing the first problem. Open your "Main" scene:



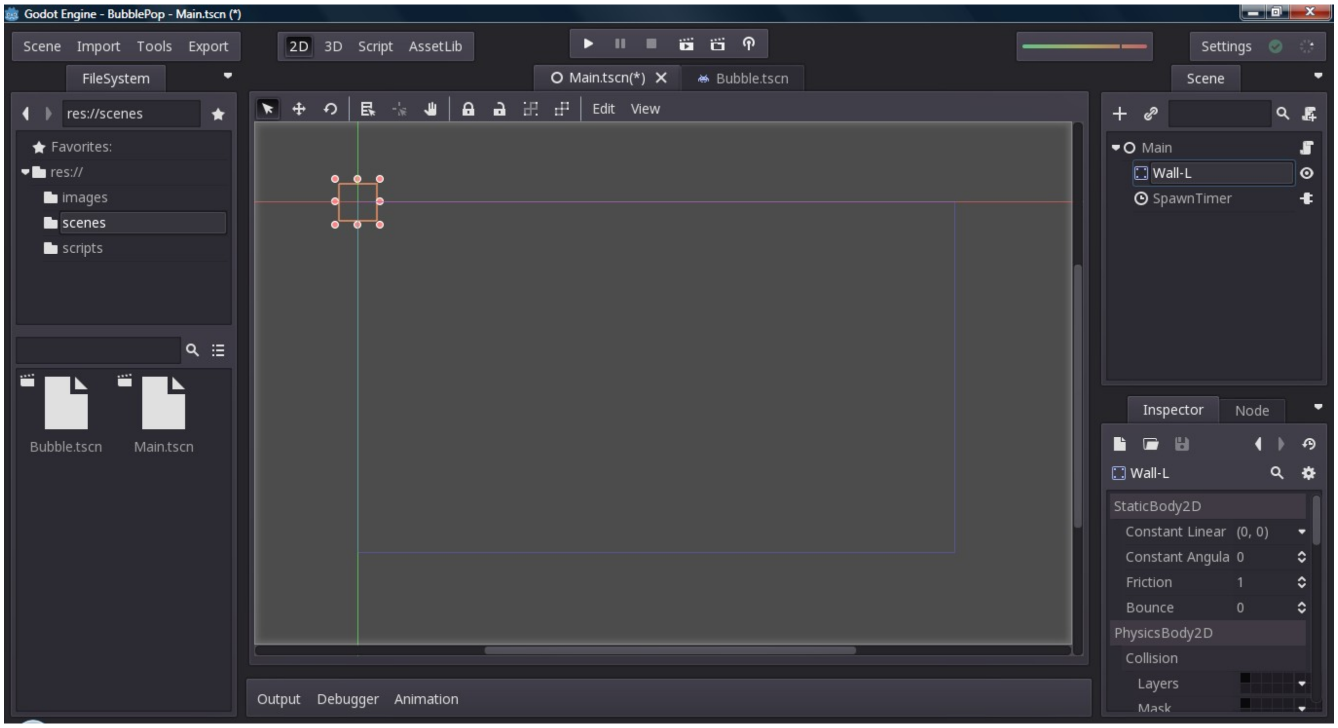
Now right-click the “Main” node and choose “Add Child Node”:



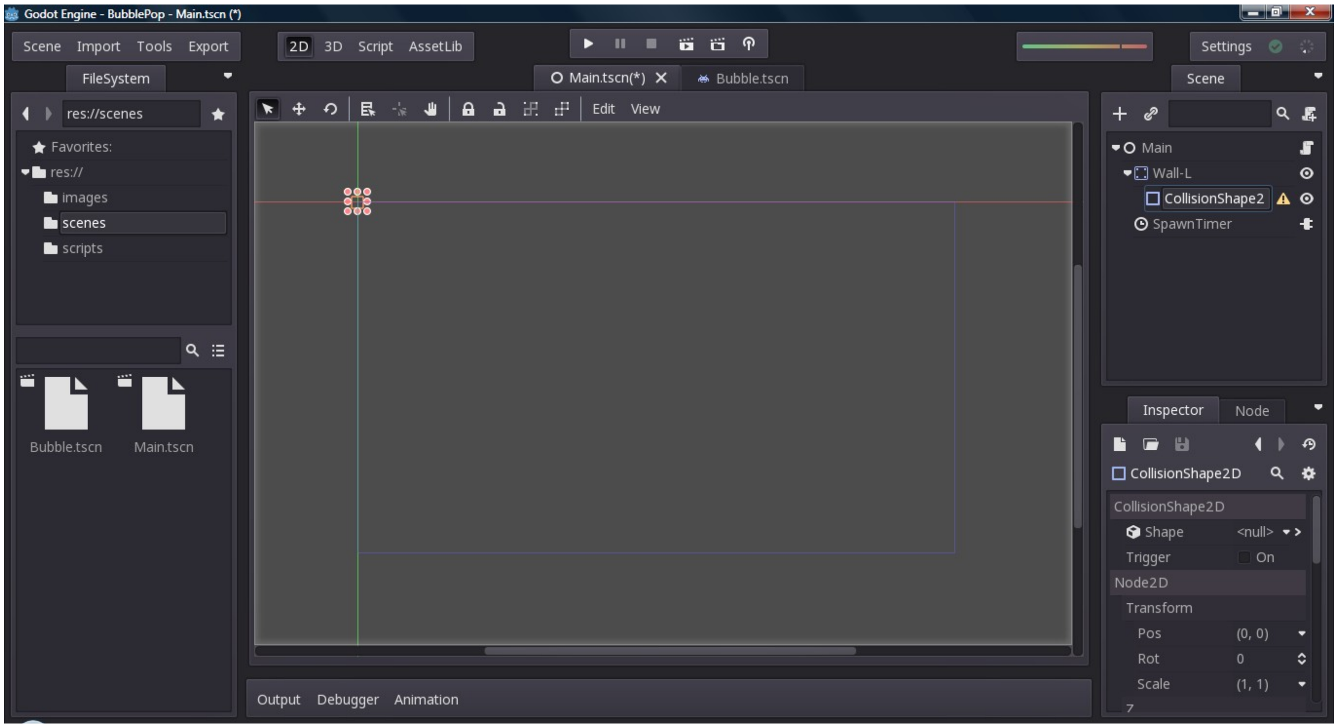
This time, we are going to create a type of node called a StaticBody2D:



A StaticBody is a type of node that can be used to represent unmovable objects such as the ground or walls. Let's start by changing the name of our new StaticBody to "Wall-L" and dragging it above our spawn timer node:

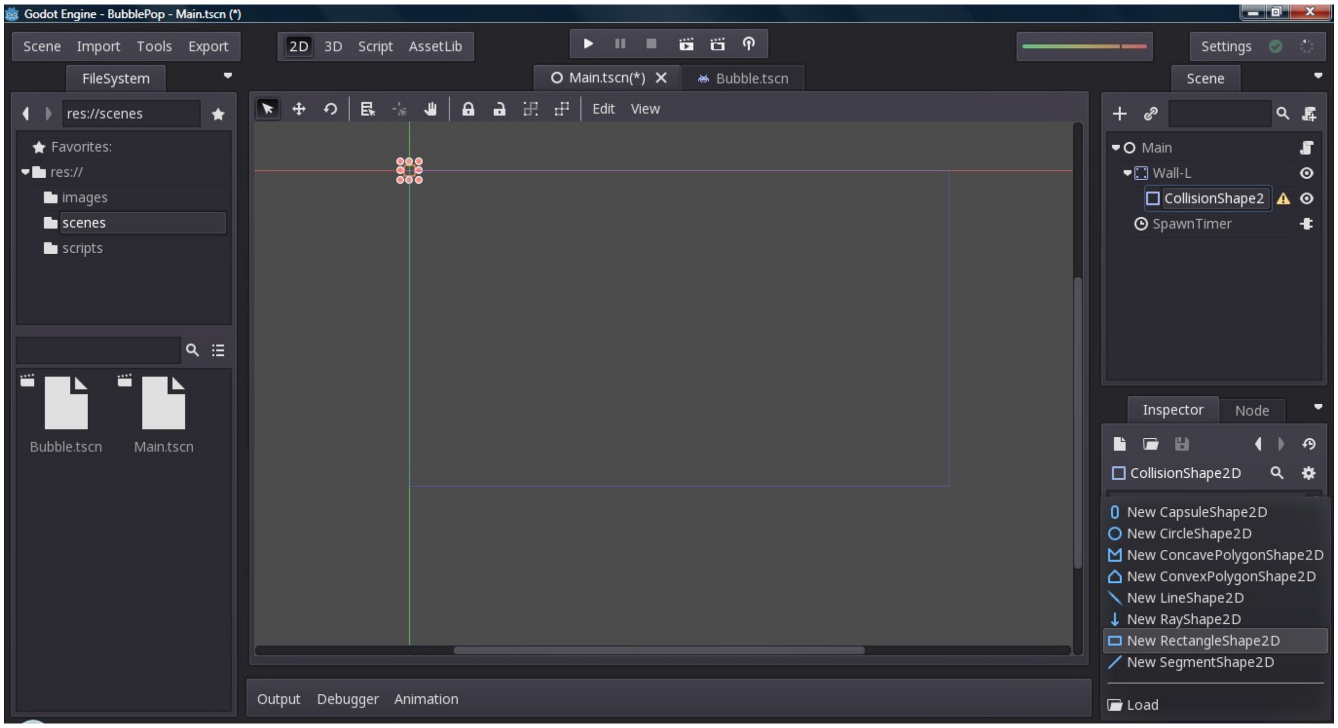


This time we will not need a visual representation for our new node, however we still need to add a collision shape to it as a child node. Right-click "Wall-L" and choose "Add Child Node". Then choose CollisionShape2D and click "Create":

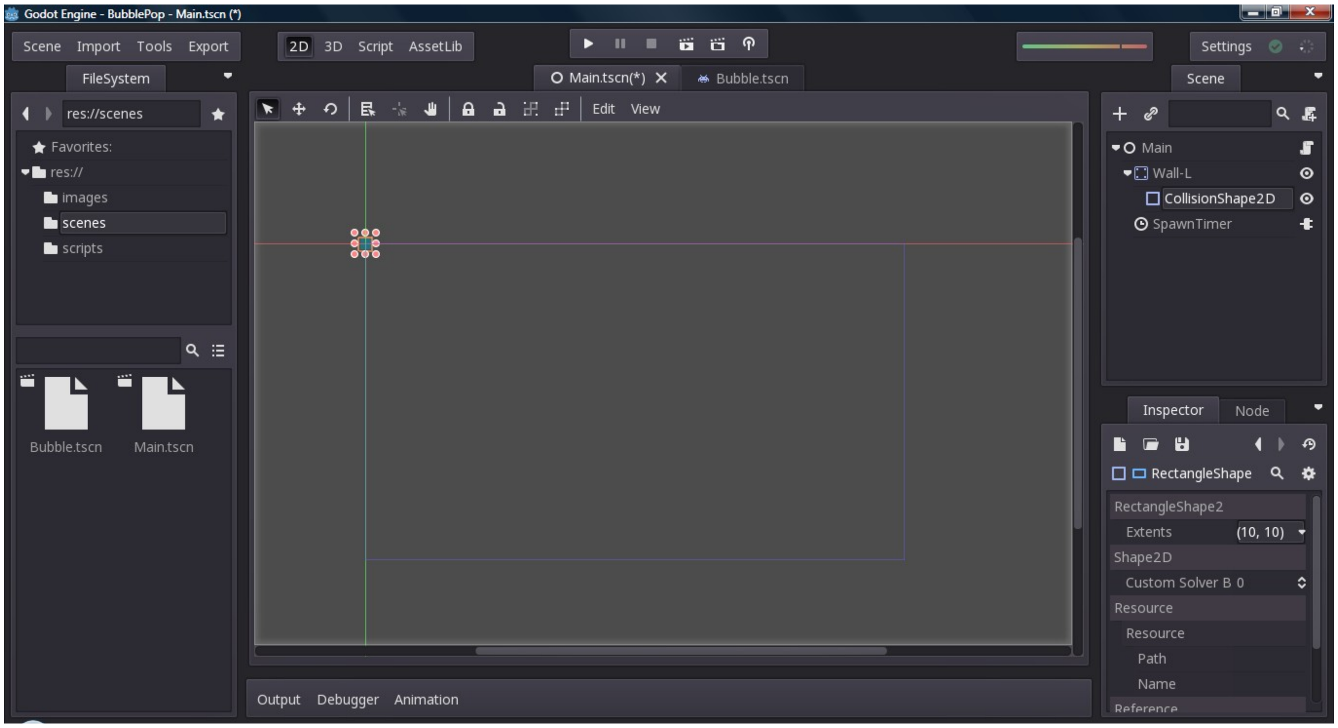


CollisionShape2D	
Shape	<null>
Trigger	On
Node2D	
Transform	
Pos	(0, 0)
Rot	0
Scale	(1, 1)

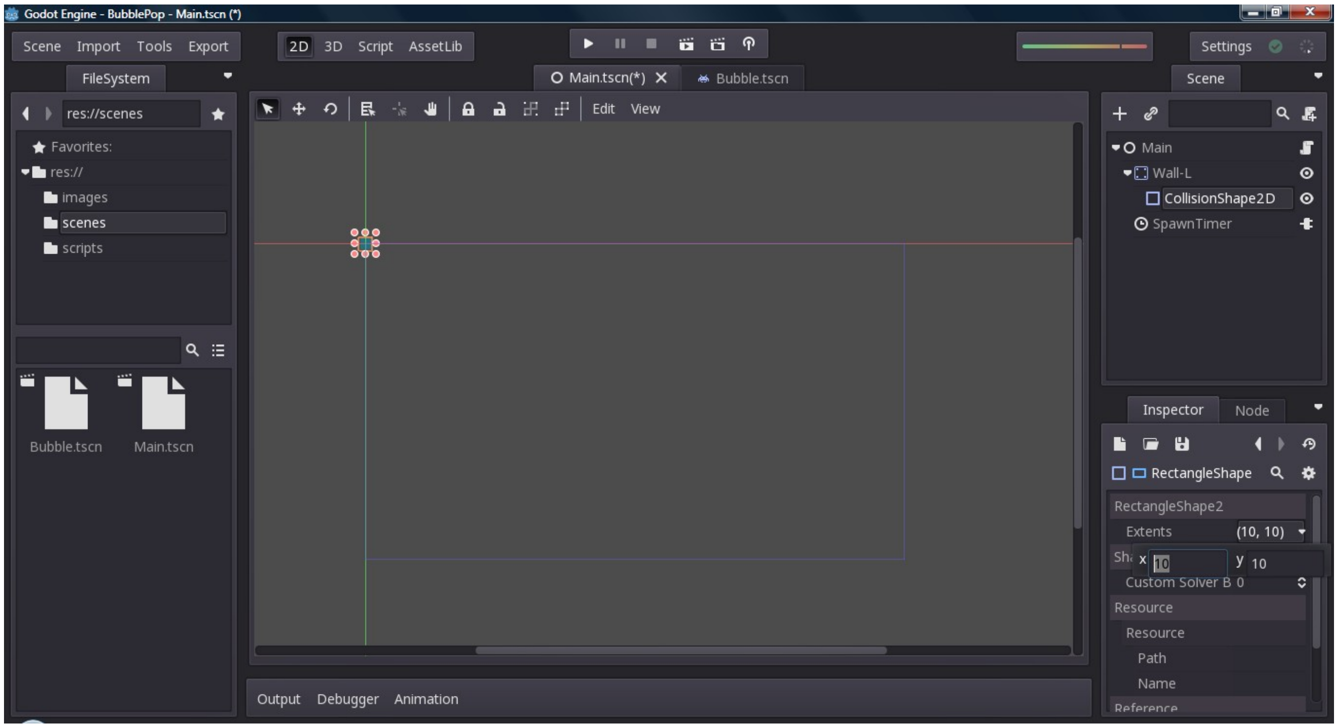
Now we need to set the shape of our new node. This time, we will choose "New RectangleShape2D":



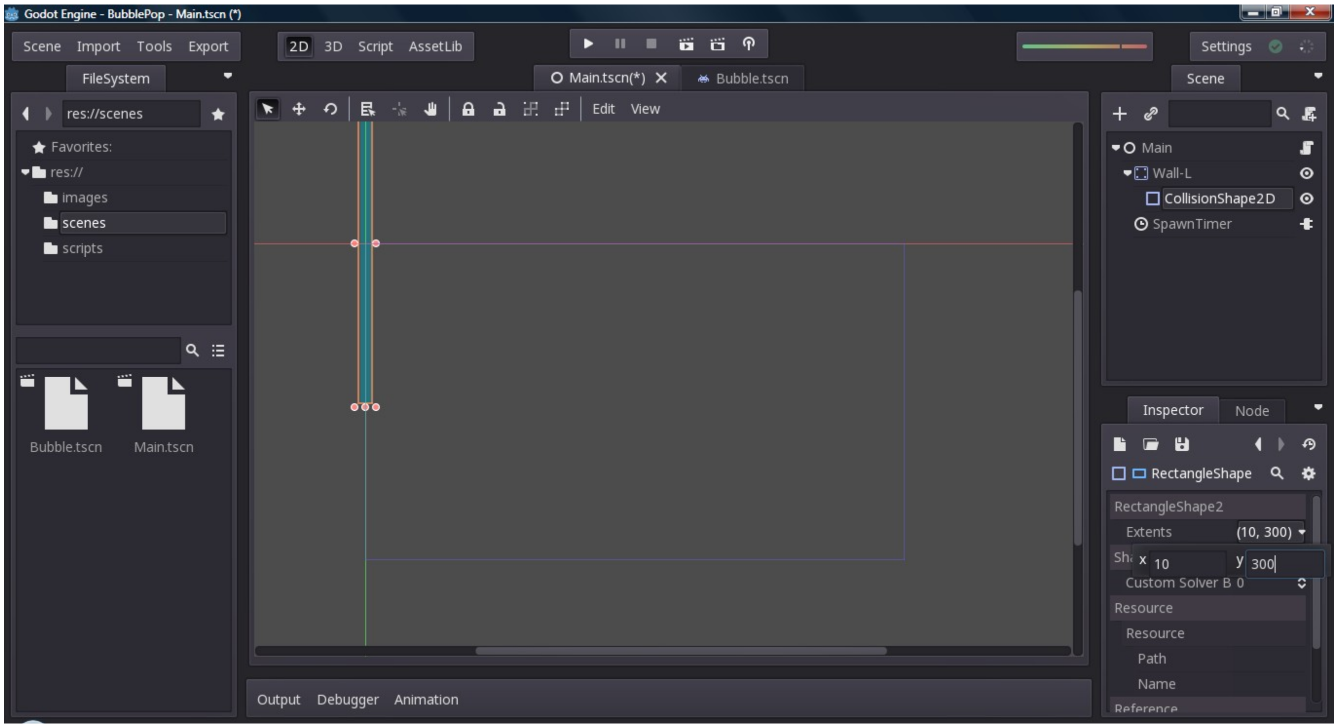
What we want to do now is resize our collision shape so that it is the same height as the screen. Since we know how tall our window will be (check the project settings), we can precisely set the height of our collision shape. If you click the arrow to the right of the shape box, you will see this:



See the "Extents" property? Those 2 values indicate how far the rectangle extends in both directions along the X and Y axes. If we click the box beside the "Extents" property, we will see this:

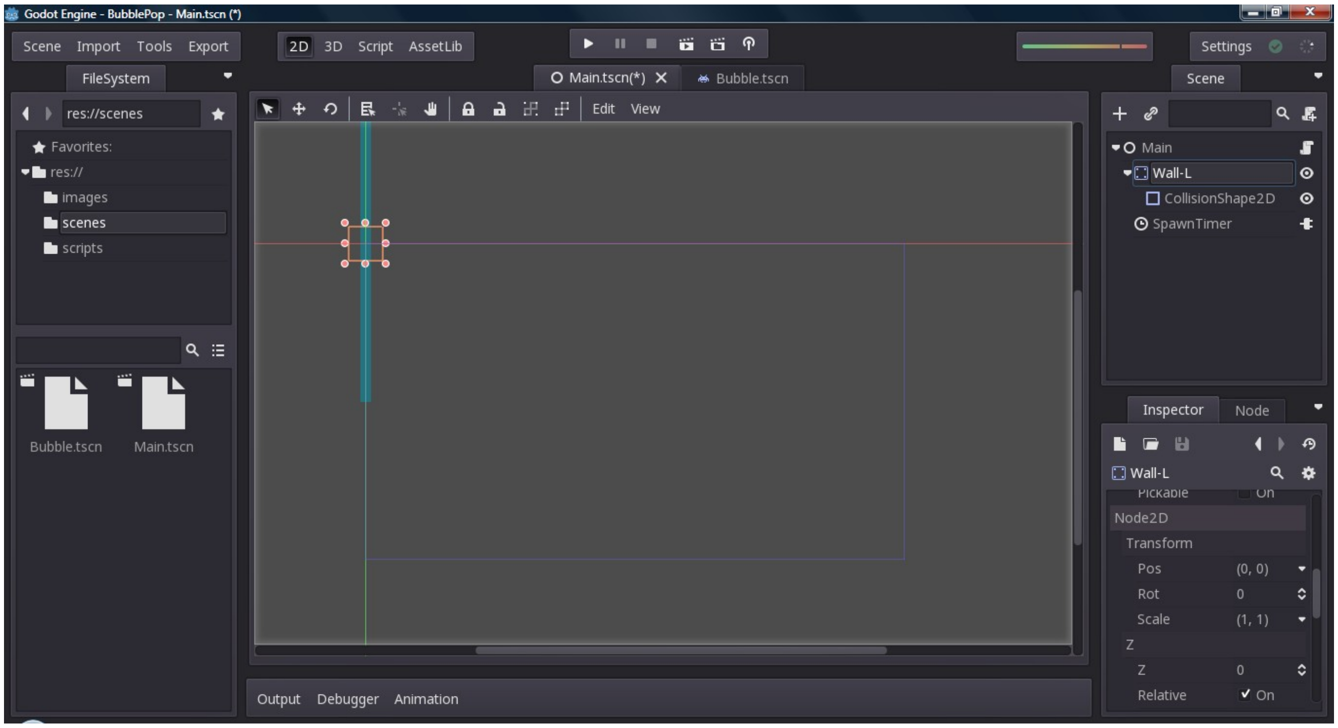


Let's set the "y" value to 300, which is half the height of the screen:



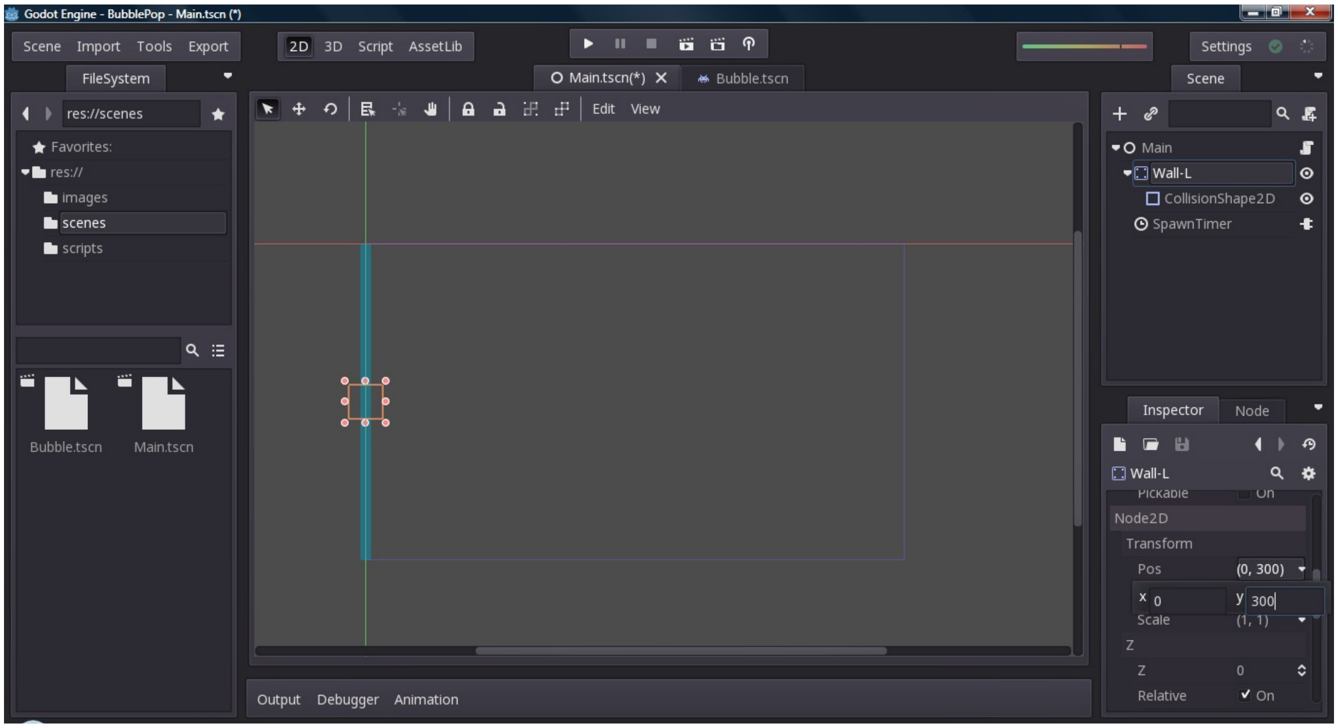
RectangleShape2
Extents (10, 300)
x 10 y 300
Custom Solver B 0
Resource
Resource
Path
Name
Reference

Hmm… the height of the rectangle is now correct, however the position is off. Let's fix that right now. Select the "Wall-L" node and scroll the lower right pane till you see the "Node2D" section:

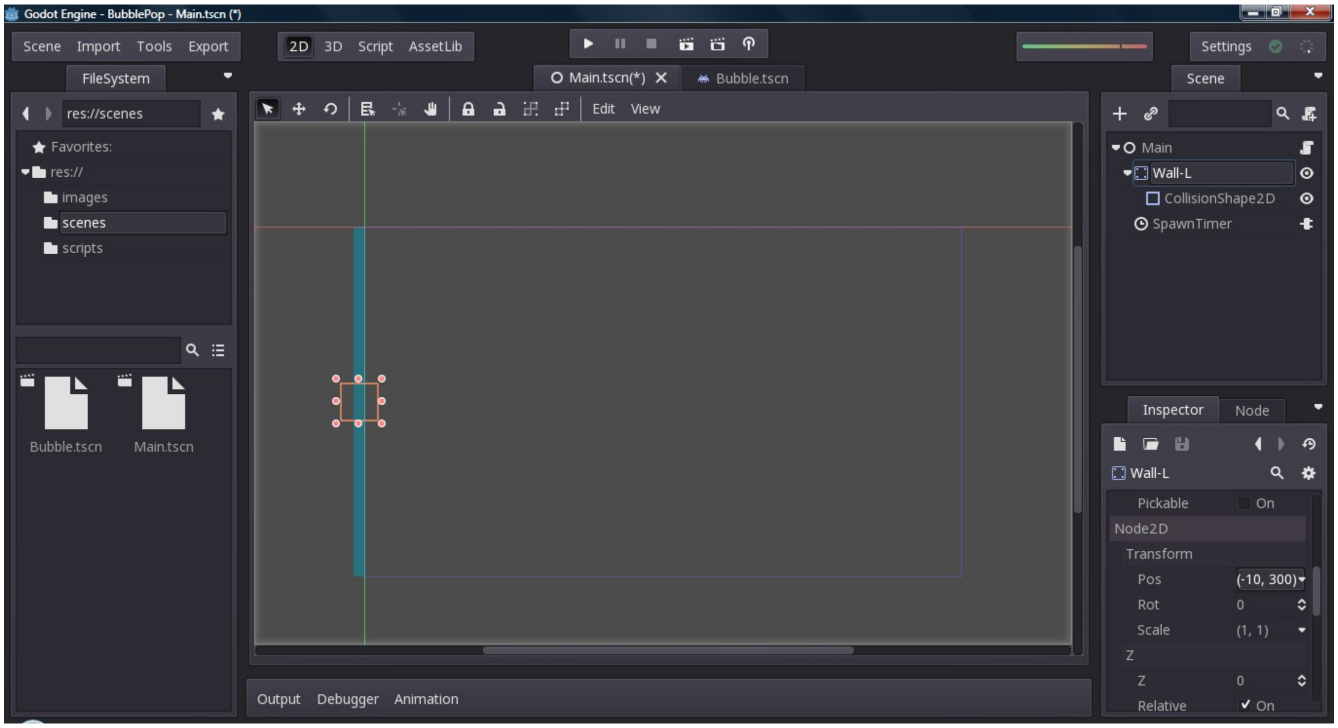


Pickable On
Node2D
Transform
Pos (0, 0)
Rot 0
Scale (1, 1)
Z
Z 0
Relative On

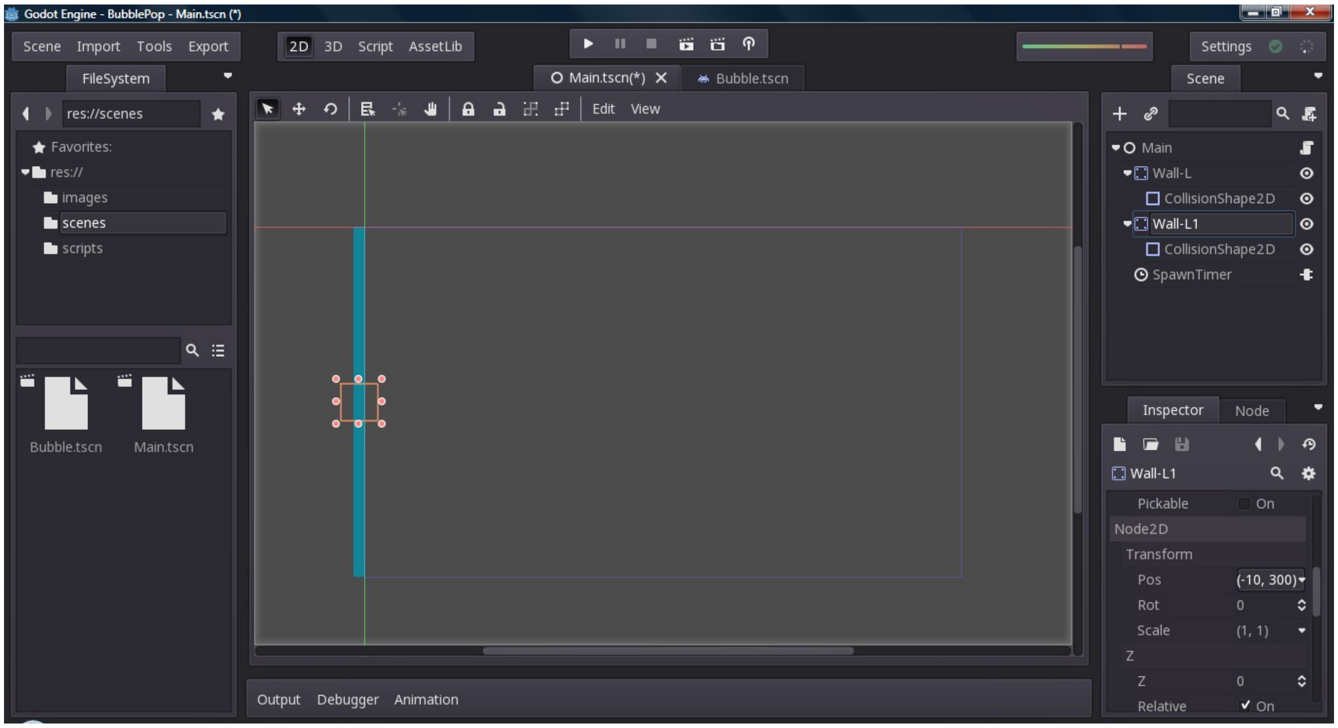
Now click the box beside "Pos" and set its "y" value to 300. This will move the wall down 300 pixels:



Now the wall is correctly positioned vertically. We also need to adjust the "x" value. Set the "x" value to -10:

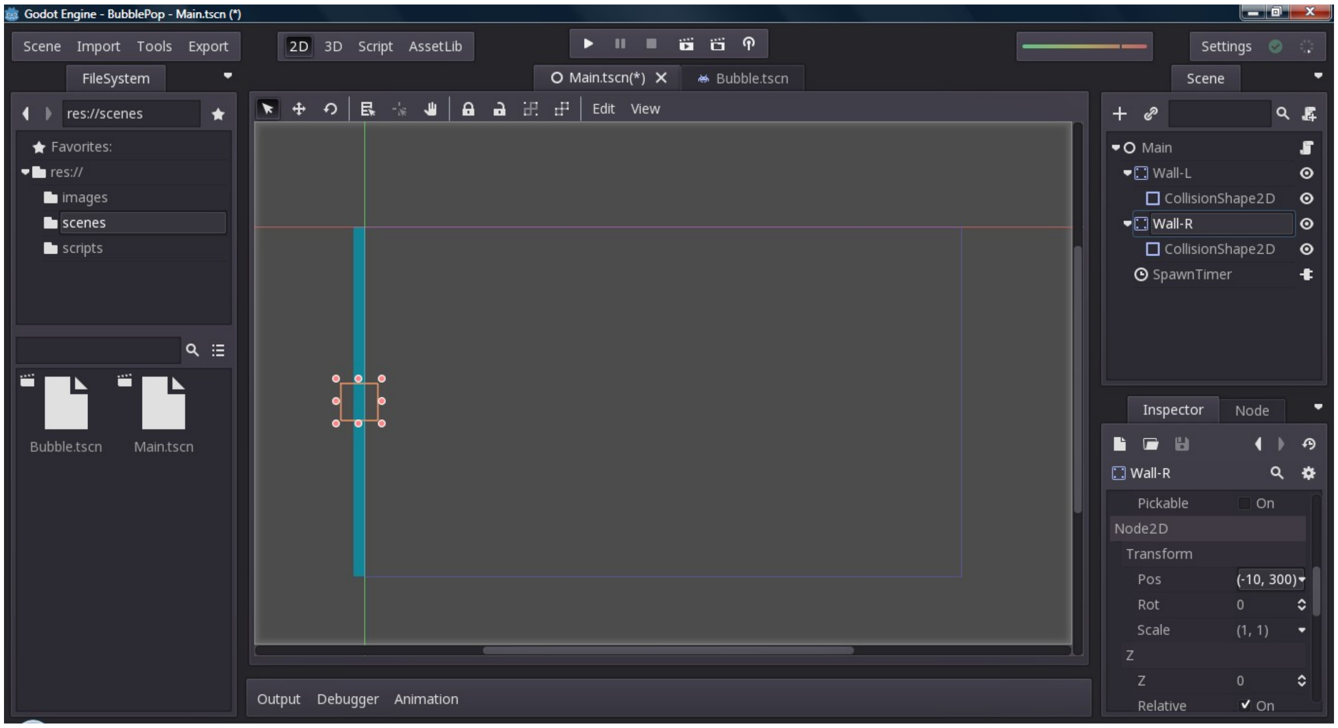


Yay! The wall is in the correct position now. However, we are not done yet. We still need a wall for the other 3 edges of the screen. There is a shortcut we can use though. Right-click the "Wall-L" node and choose "Duplicate":



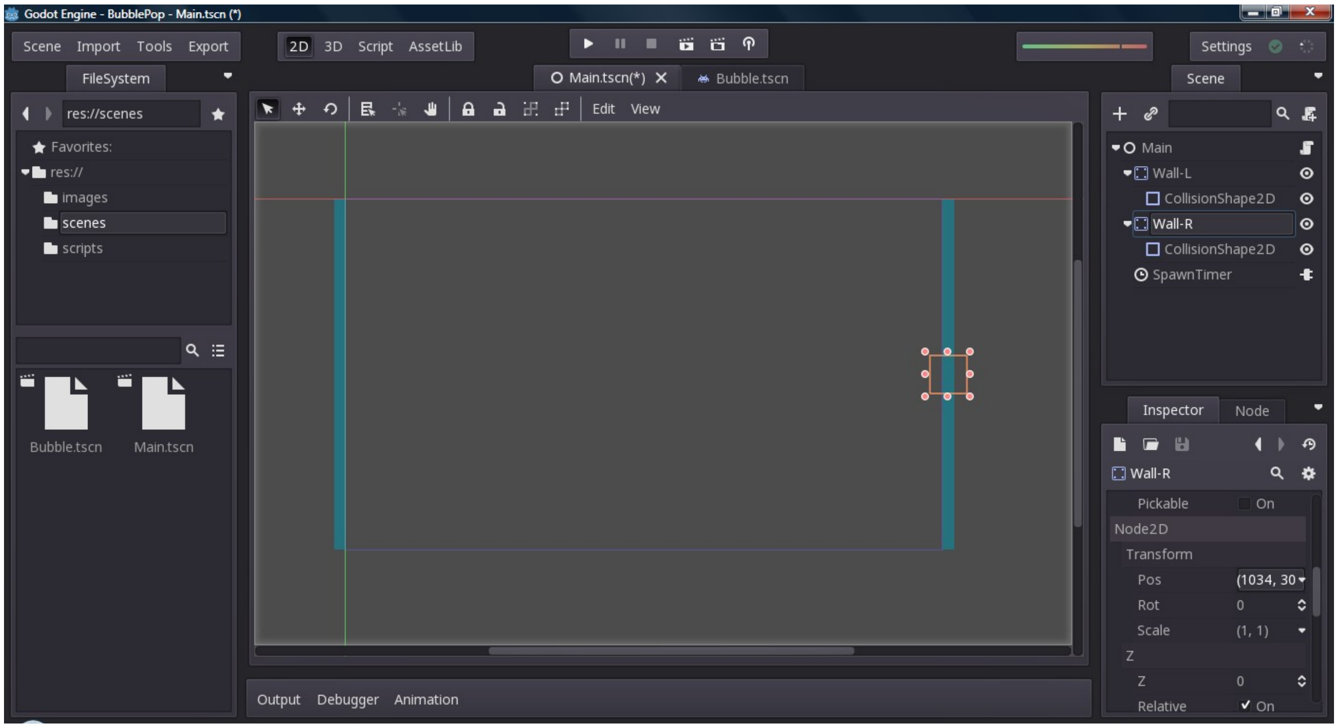
Pickable On
Node2D
Transform
Pos (-10, 300)
Rot 0
Scale (1, 1)
Z
Z 0
Relative ✓ On

Now we instantly have a new wall with a collision shape already attached. Let's start by renaming the new wall to "Wall-R":



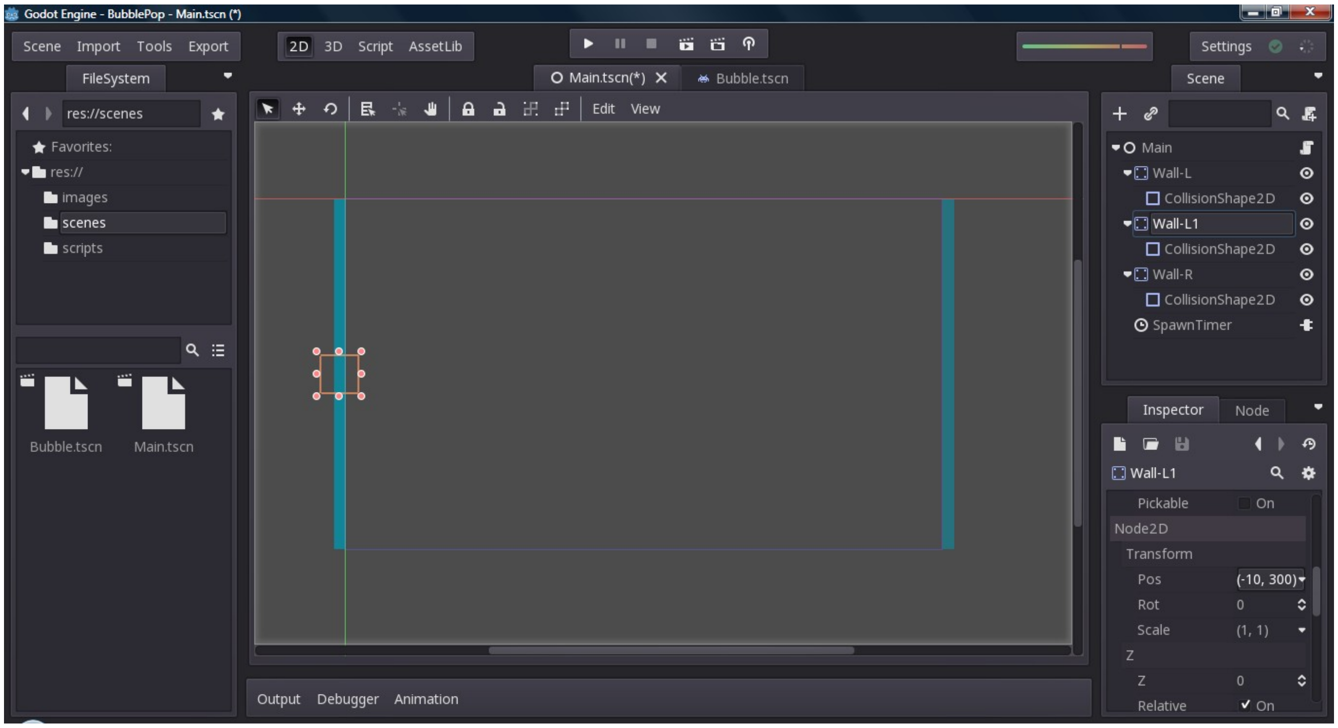
Pickable On
Node2D
Transform
Pos (-10, 300)
Rot 0
Scale (1, 1)
Z
Z 0
Relative ✓ On

We also need to set the "x" value of its position to 1034:



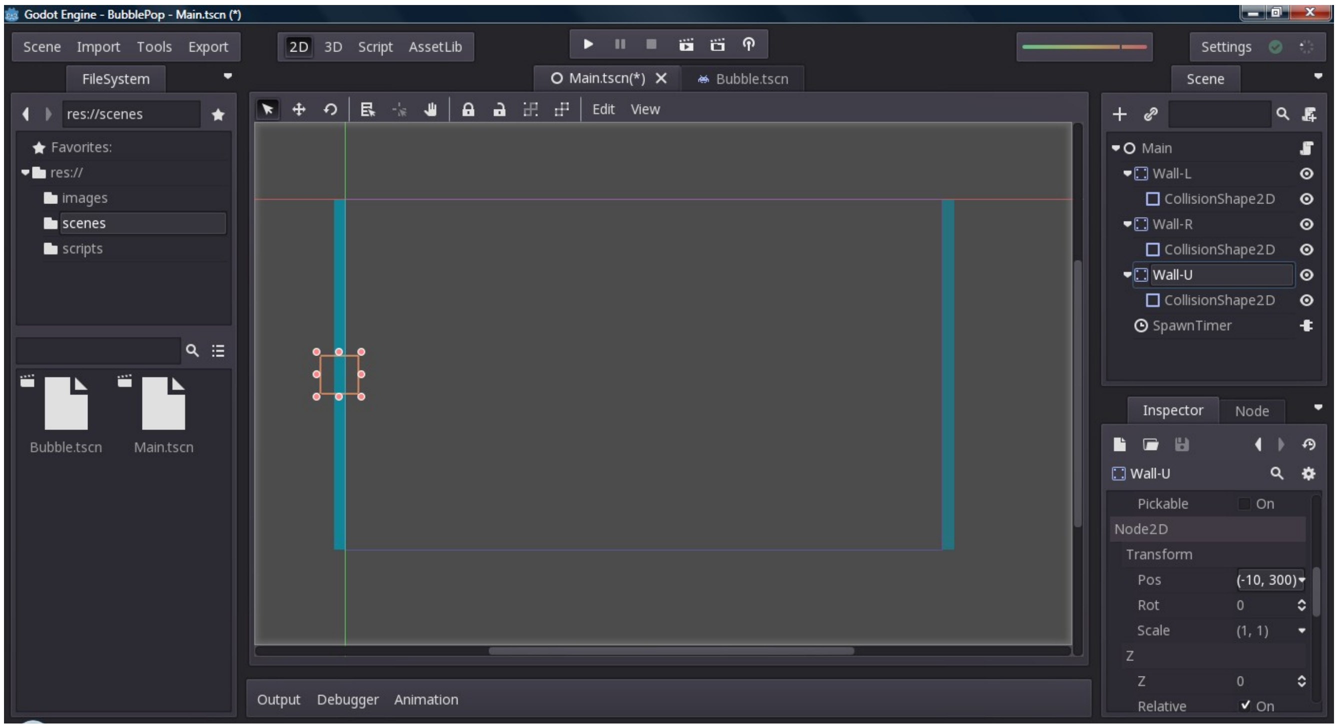
Wall-R
Pickable On
Node2D
Transform
Pos (1034, 30
Rot 0
Scale (1, 1)
Z
Z 0
Relative ✓ On

We now have 2 walls. Let’s duplicate the left wall again:



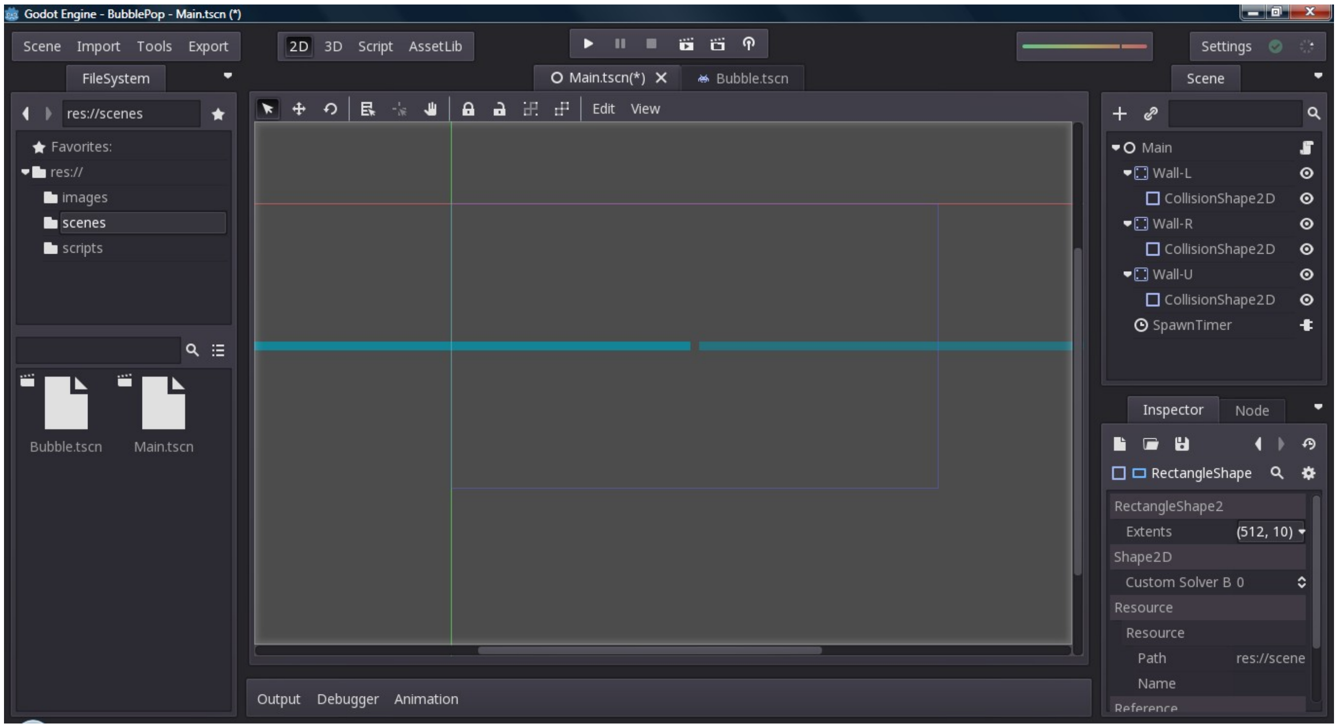
Wall-L1
Pickable On
Node2D
Transform
Pos (-10, 300)
Rot 0
Scale (1, 1)
Z
Z 0
Relative ✓ On

Change the name of the new wall to “Wall-T” and move it below the “Wall-R” node:



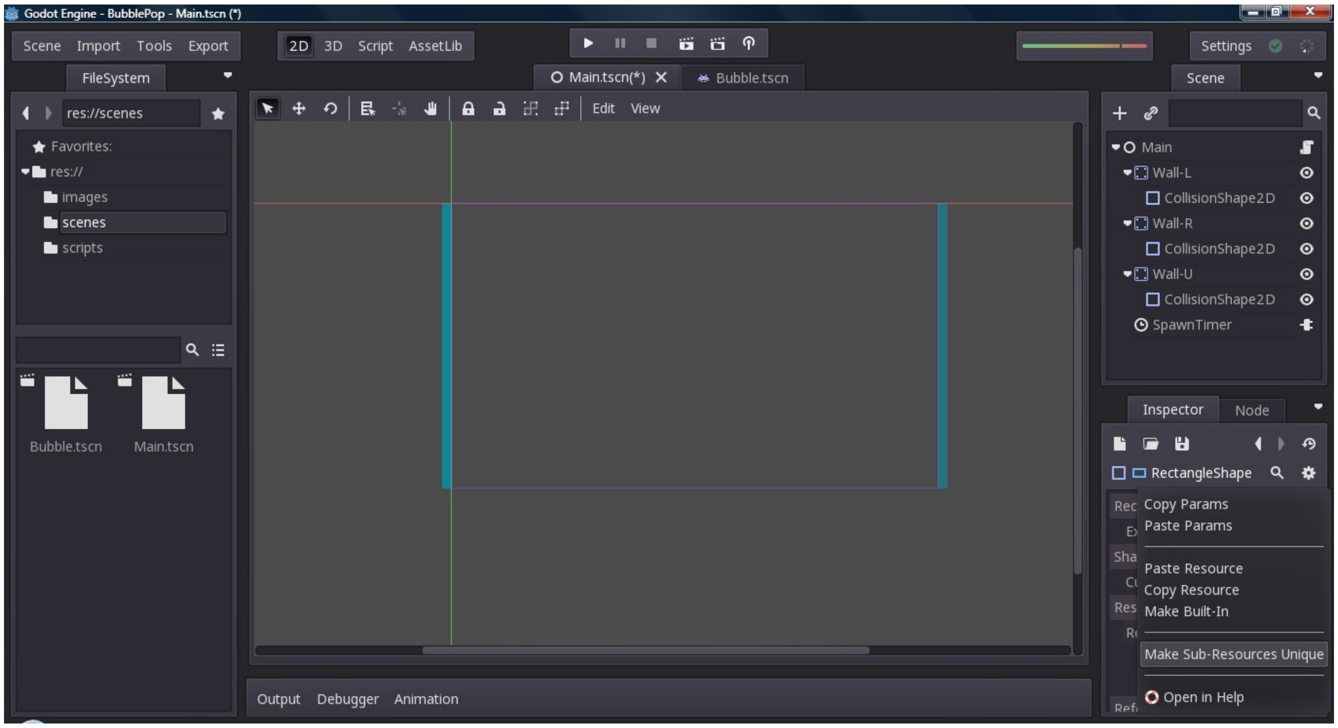
Wall-U
Pickable On
Node2D
Transform
Pos (-10, 300)
Rot 0
Scale (1, 1)
Z
Z 0
Relative ✓ On

This time, we will need to adjust the extents of our collision shape and reposition the wall. Select the collision shape for "Wall-T" and set its extents to (512, 10):

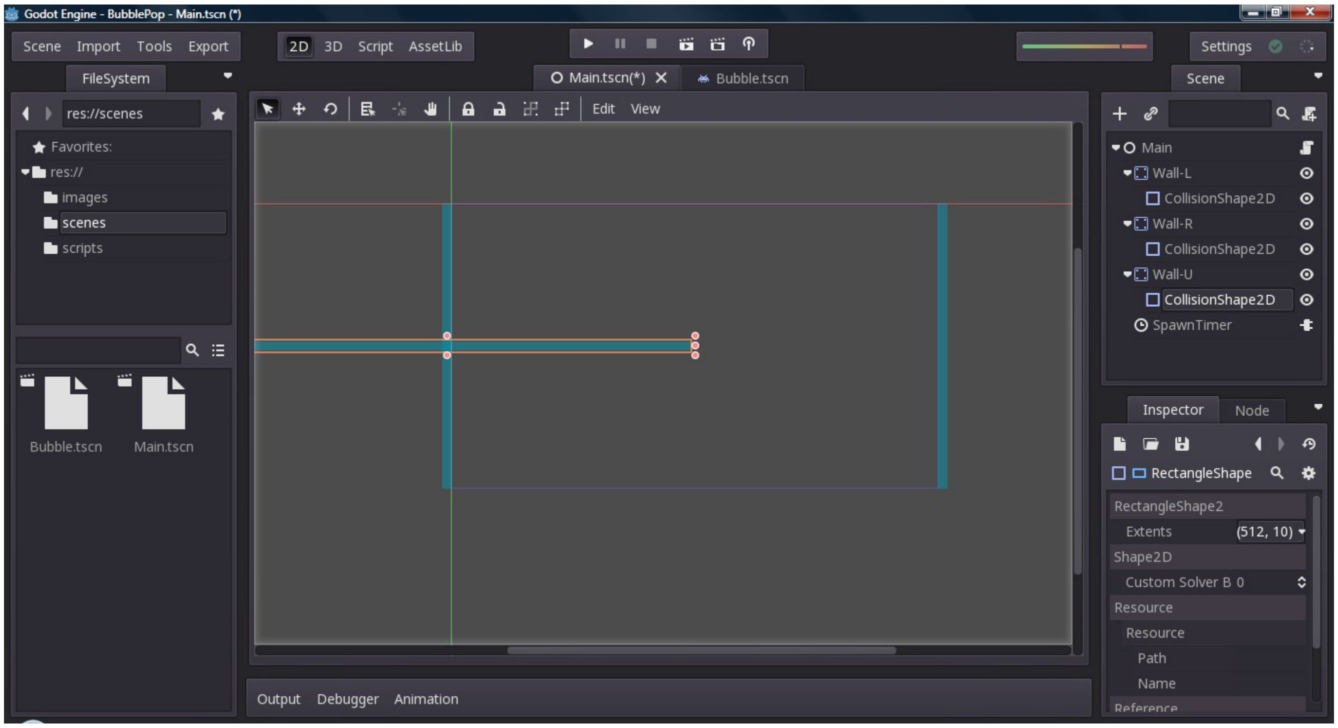


RectangleShape
RectangleShape2
Extents (512, 10)
Shape2D
Custom Solver B 0
Resource
Resource
Path res://scene
Name
Reference

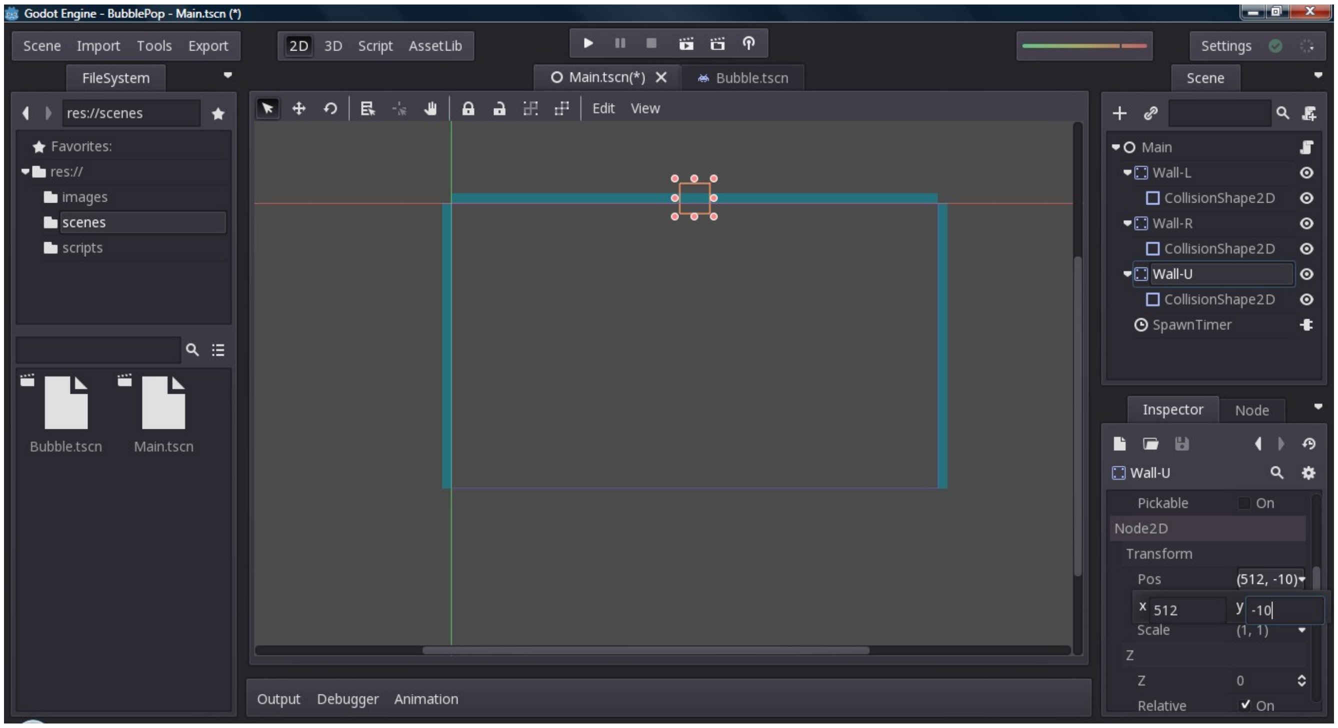
Oops. Looks like we made a mistake. What went wrong? The problem is that each duplicate collision shape shares the same shape data. Press Ctrl + Z to undo your last action, then click the little back button in the lower right pane. Afterward, click the little gear button in the lower right pane:



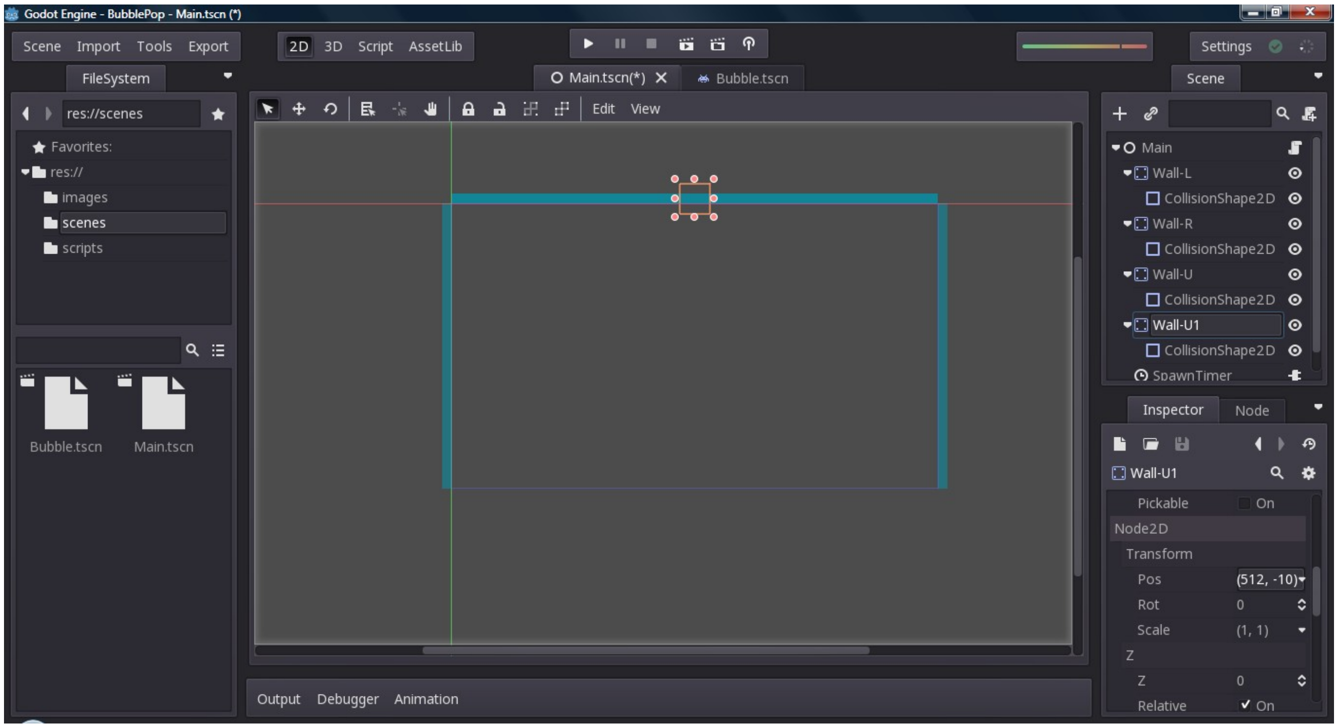
Click the “Make Sub-Resources Unique” option. This will make create a unique copy of the shape data for this CollisionShape and assign it. Now try changing the extents to (512, 10) again:



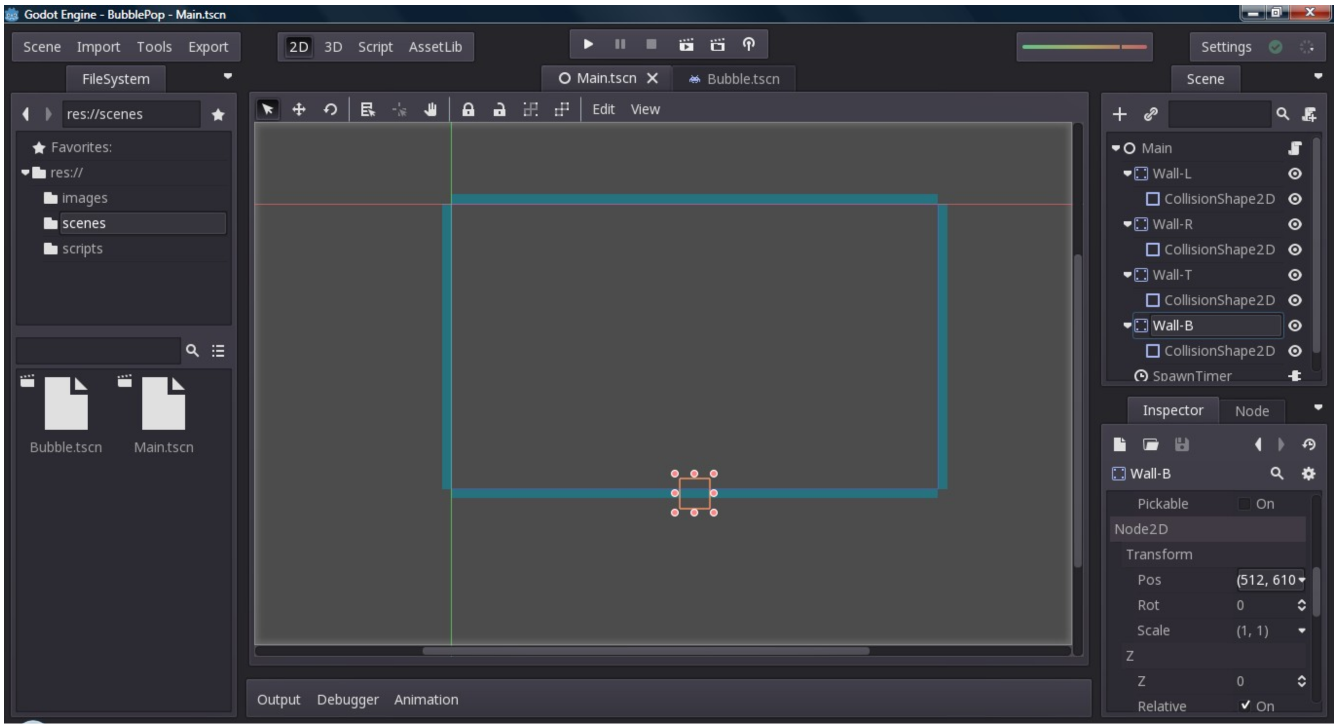
This time it worked as expected. Now we need to set the position of "Wall-T" to (512, -10):



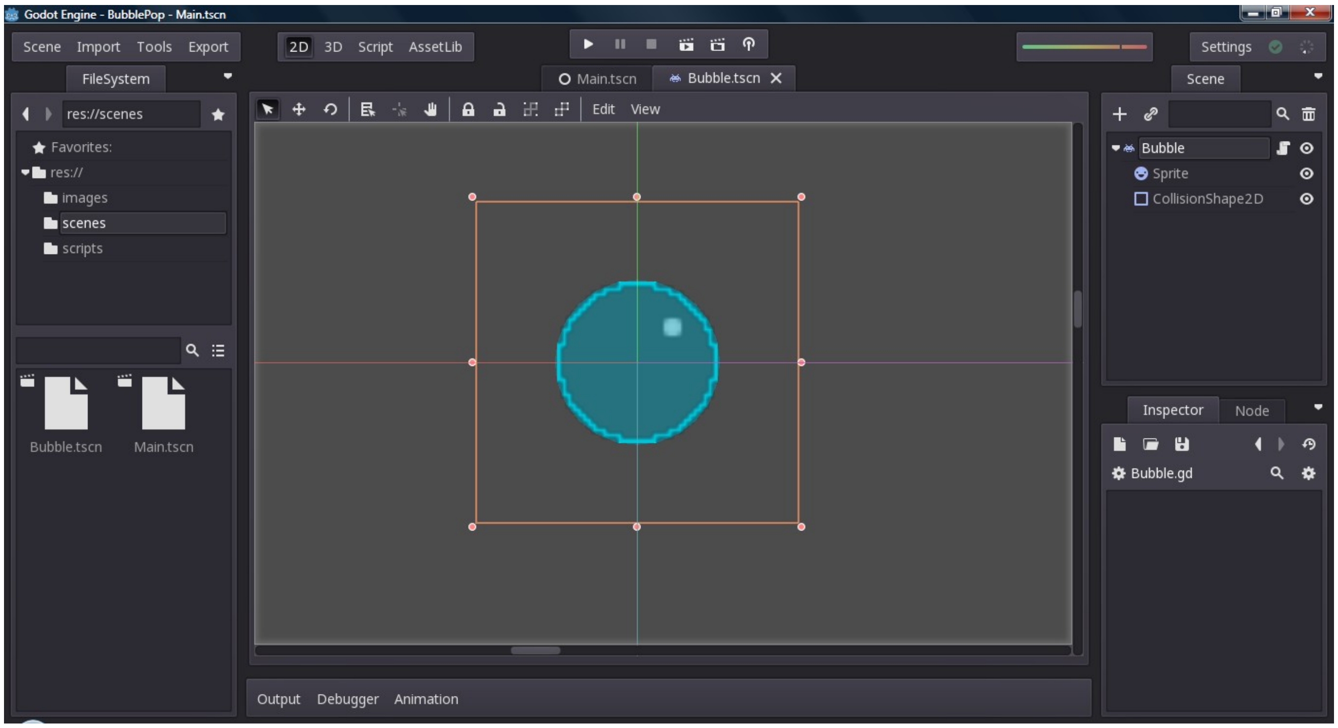
The upper wall is now in place. Let's duplicate it:



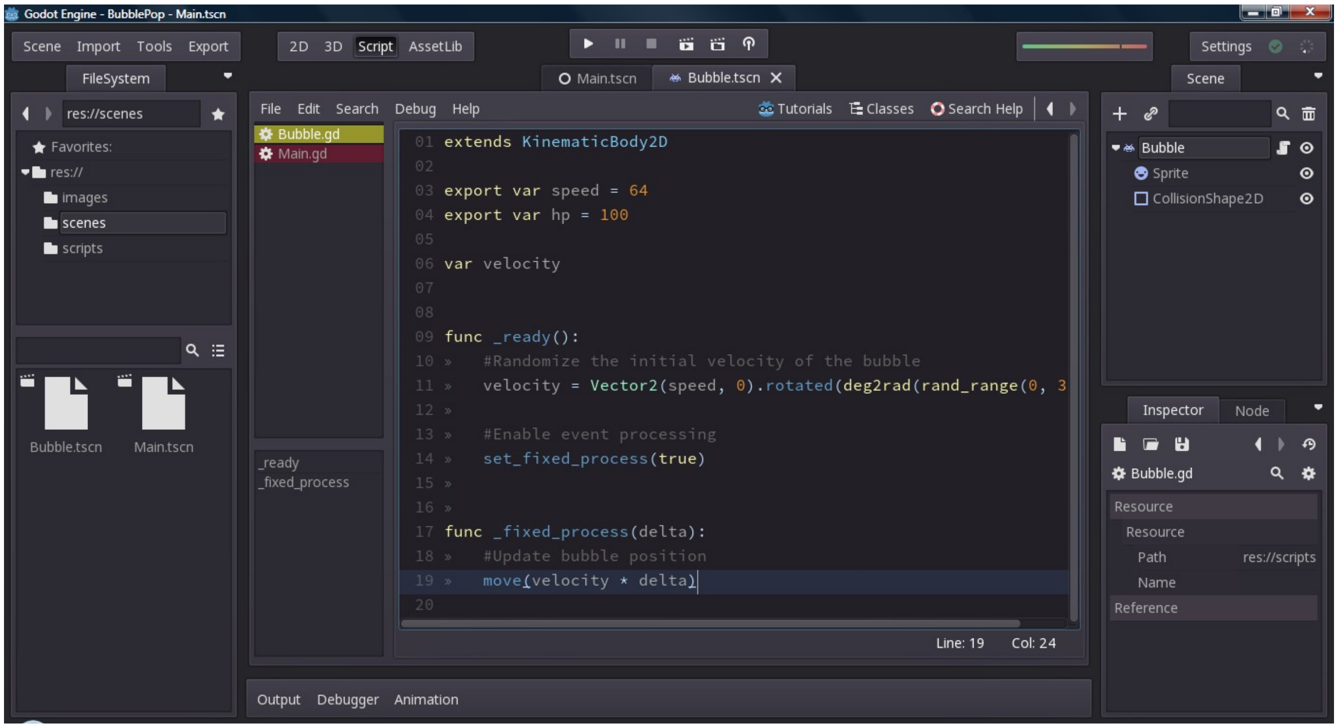
Now rename the new wall to "Wall-B" and move it to (512, 610):



We now have a wall at each edge of the screen. It is time to modify our bubble so it bounces off walls. Switch back to your “Bubble” scene:

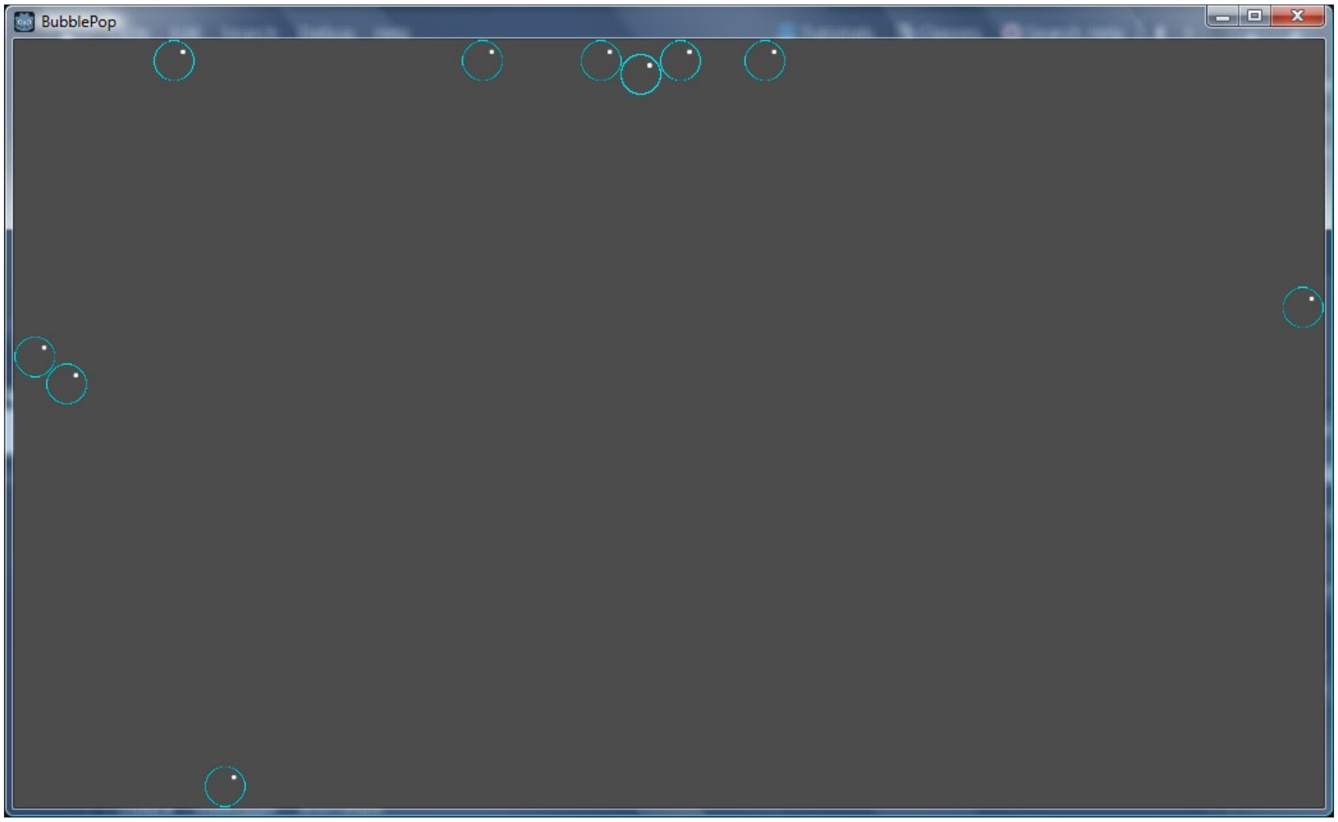


Click the little script icon next to your “Bubble” node:



```
extends KinematicBody2D

export var speed = 64
export var hp = 100

var velocity


func _ready():
	#Randomize the initial velocity of the bubble
	velocity = Vector2(speed, 0).rotated(deg2rad(rand_range(0, 3
	
	#Enable event processing
	set_fixed_process(true)
	
	
func _fixed_process(delta):
	#Update bubble position
	move(velocity * delta)

```

That particular icon is a useful shortcut that opens the script attached to a node. Let’s try running our game before we modify it further:



Hmm… Now our bubbles are getting stuck on the edges of the screen. Why is this happening? Remember how the “move” function causes a KinematicBody to stop when it hits another object? In order to get our bubbles to bounce off the walls, we need to adjust the velocity of our bubble after it hits a wall:

```
func _fixed_process(delta):
	#Update bubble position
	move(velocity * delta)

	#Update velocity
	if is_colliding():
	    var collider = get_collider()
	    var collider_name = collider.get_name()

	    if collider_name in ["Wall-L", "Wall-R"]:
		    velocity = Vector2(1, 0).reflect(velocity)

	    elif collider_name in ["Wall-T", "Wall-B"]:
		    velocity = Vector2(0, 1).reflect(velocity)
```

The "is_colliding" method returns true if our bubble has collided with another object. If a collision is detected, we need to get the object that the bubble collided with and then get its name so we can identify which type of object the bubble collided with. If the name of the other object is the left or right wall, we will reflect our current velocity along the X-axis. And if the name of the other object is the top or bottom wall, we will reflect our current velocity along the Y-axis. Now if we run our game again, the bubbles will bounce off the walls correctly. However, we still need to modify our bubble object so that it can bounce off other bubbles too. We will do that in the next lesson.