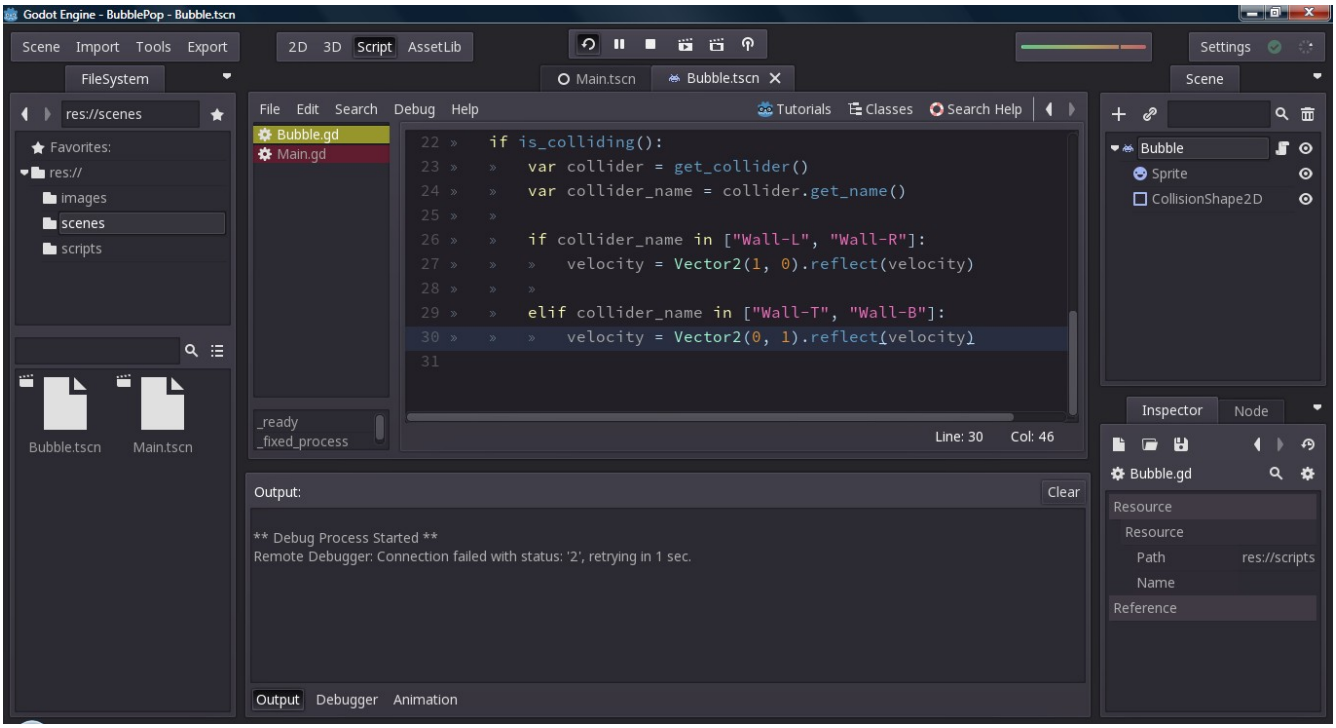
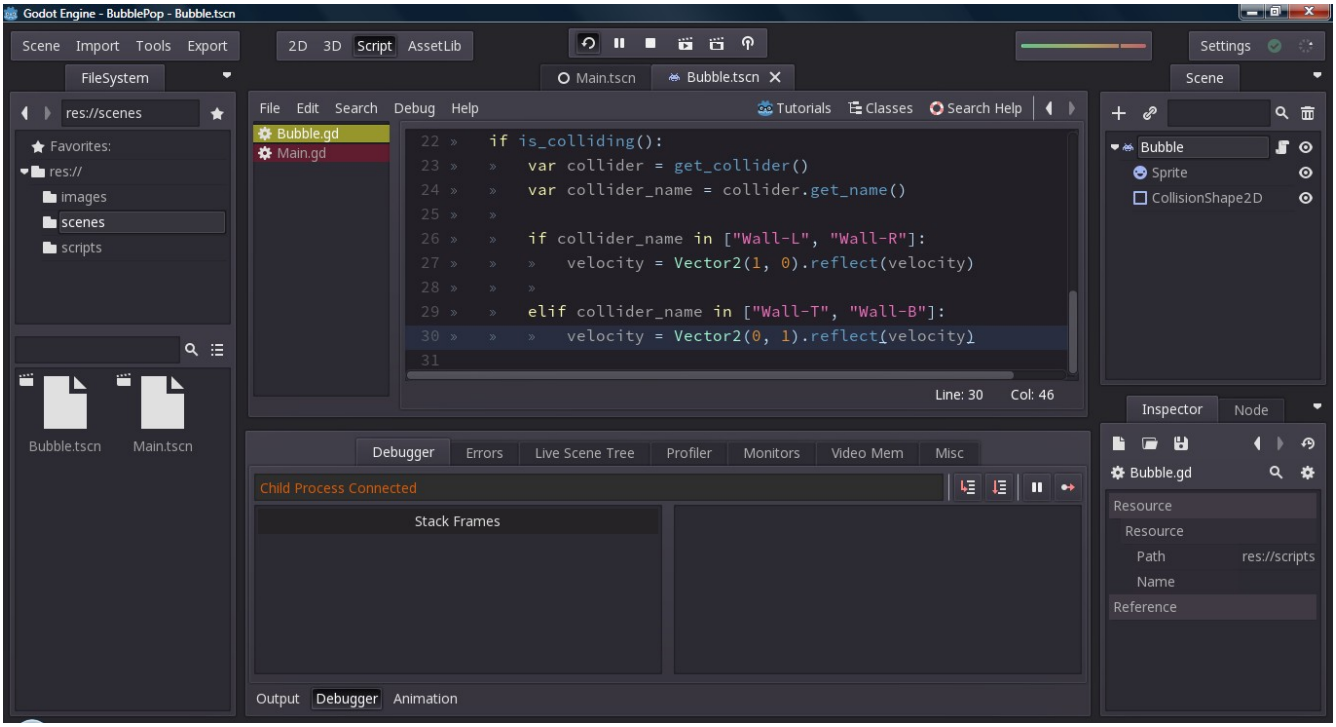
Godot 2D Game
Lesson 8: Custom Signals

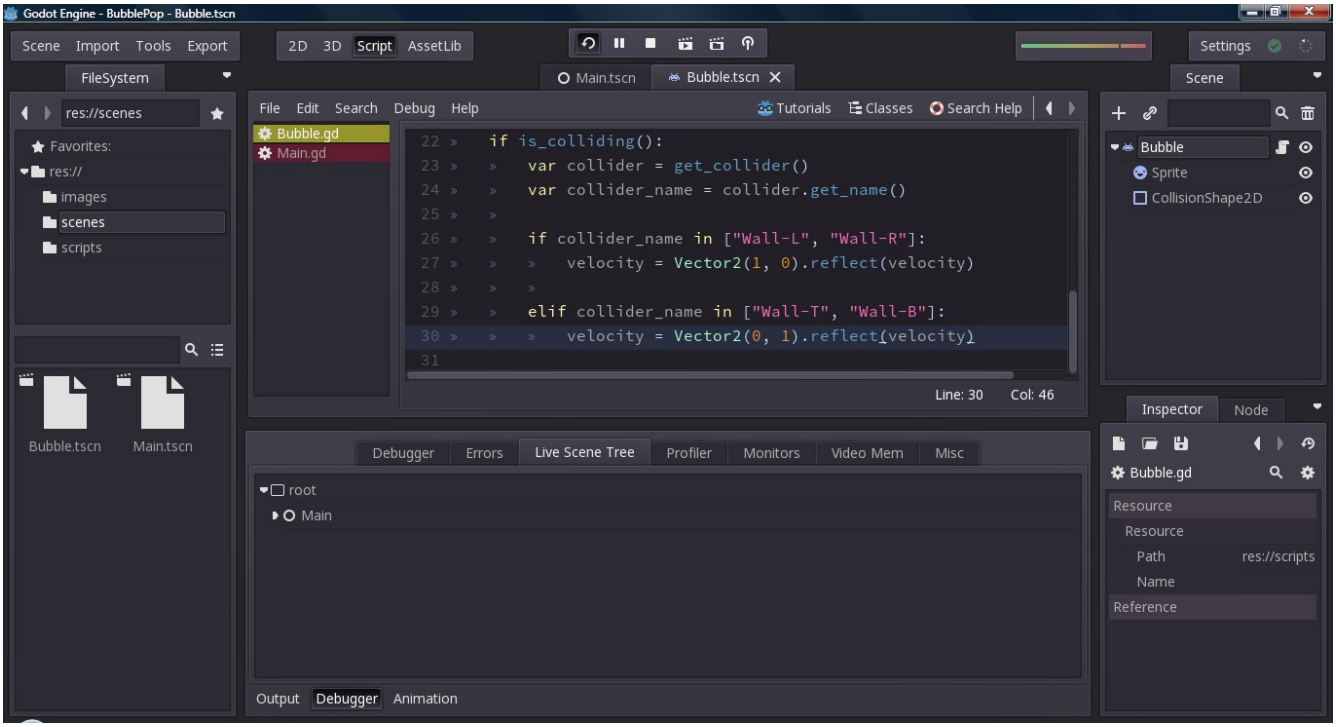
In this lesson, we will modify our bubbles so that they bounce off each other. We will also add code to update our bubbles' HP and pop them. However, checking to see if the other object is a bubble will be a bit trickier than it was with the walls. Try running the game again and switch back to the editor window with the game still running:



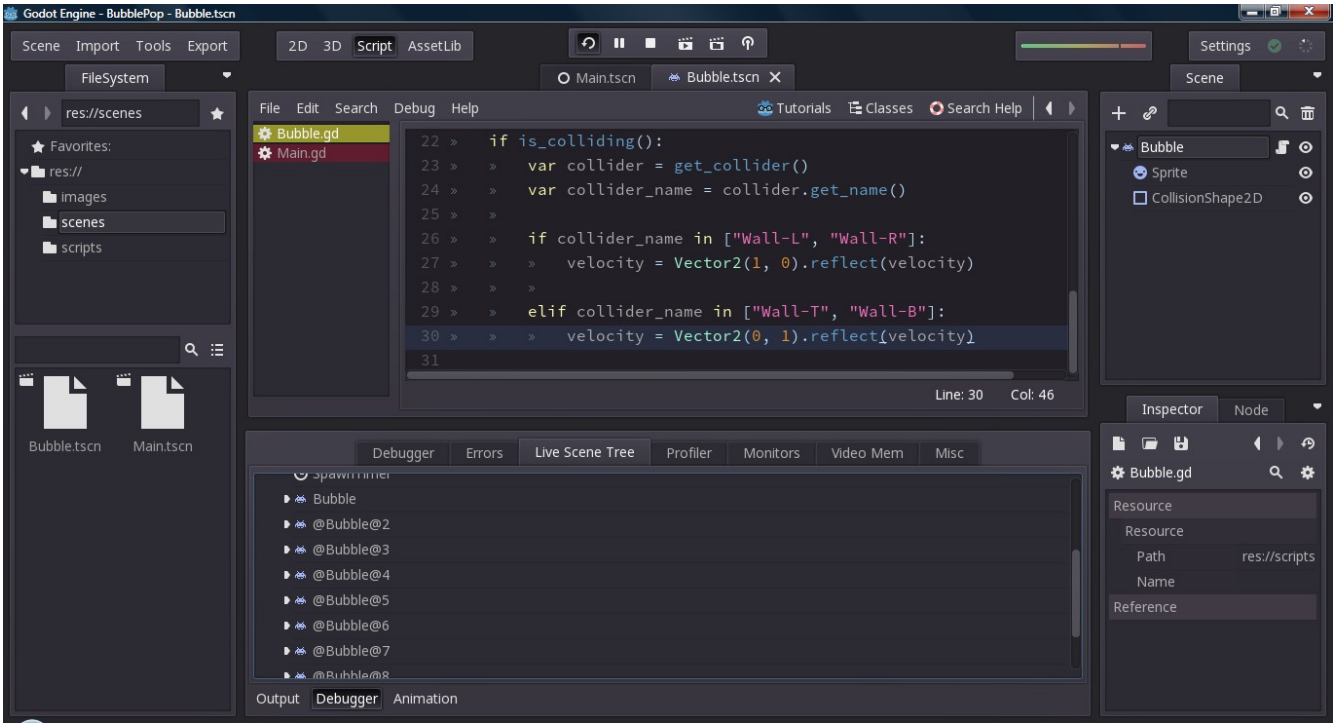
Now click the "Debugger" tab at the bottom:



And click the “Live Scene Tree” tab at the top of the bottom pane:



This tree view shows everything in the scene for the currently running game in realtime. Try expanding nodes until you see the bubbles:



Hmm… isn't this interesting. The first bubble has is called "Bubble", but the others have an "@" symbol at the beginning and an "@" followed by a number on the end. What is going on here? It's like this. For each child of a node, there can only be one node with any given name at a time. When we instance our bubble the first time, everything works fine because no other node is called "Bubble". However, when we instance our bubble again, Godot cannot use "Bubble" for the new node's name since that name is already taken. Therefore, Godot randomly generates a unique name for each additional bubble. This means that we have to check for 2 conditions:

a) the name is "Bubble"
b) the name contains "@Bubble@"

Let's close the game and write some code to check for other bubbles:

```
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)

	elif collider_name == "Bubble" or "@Bubble@" in collider_name:
		velocity = collider.velocity.normalized().reflect(velocity)
```

This time, we need to use a different method to update the velocity of our bubble as well. The walls were either vertical or horizontal, but what direction should we use to reflect bubbles? We use the velocity of the other bubble ofc. However, notice that we call the "normalized" method of the other bubble's velocity before "reflect". The "normalized" method ensures that the length of a vector is 1. Without that method call, our game would freeze when 2 bubbles collided.

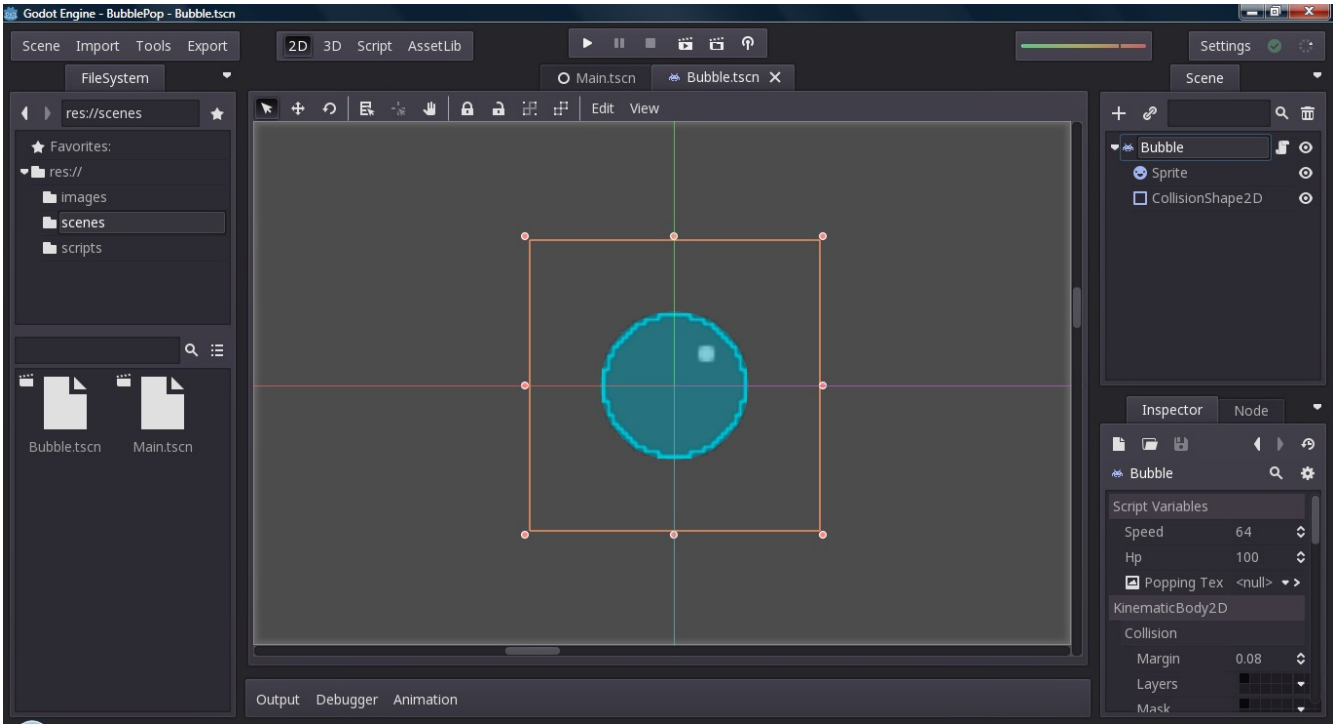
Now that our bubbles bounce properly, we should also add code to update the HP of our bubble. Let's start by creating a new function called "add_hp":

```
func add_hp(inc):
	#Update the HP of this bubble
	hp += inc

	#Did the bubble pop?
	if hp <= 0:
	    pass
```

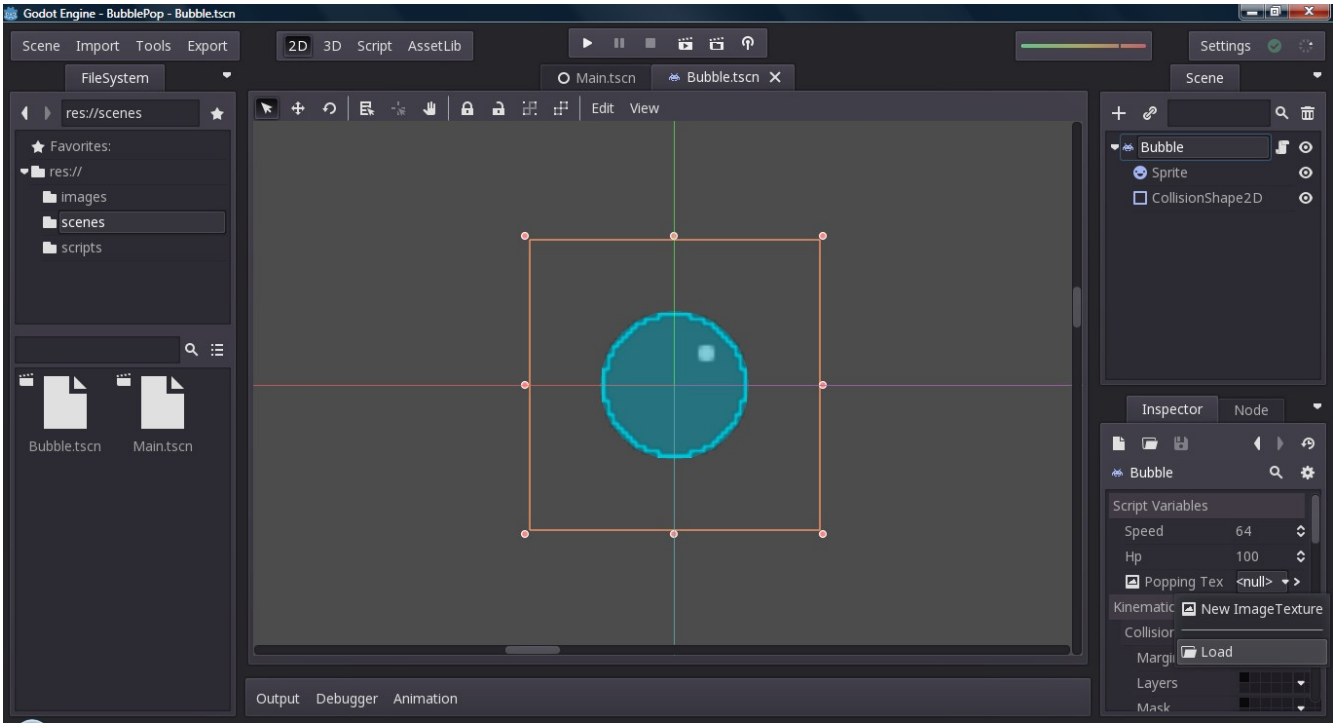
This function is pretty simple. For starters, our new function takes a single parameter called "inc". "inc" is the amount of HP to add. If we want to subtract from the bubble's current HP, we simply use a negative increment. We also check to see if the bubble's HP is less than 0. However, we need to setup a few more things before we can finish writing our "add_hp" function. When the bubble pops, we need to change it's sprite to the popping sprite. But how can we do that? Every Sprite node has a "set_texure" function that we can call to change the texture for the sprite. But first, we need to load the texture we intend to use. Let's start by adding a new property called "popping_tex":

```
export var speed = 64
export var hp = 100
export (ImageTexture) var popping_tex

var velocity
```

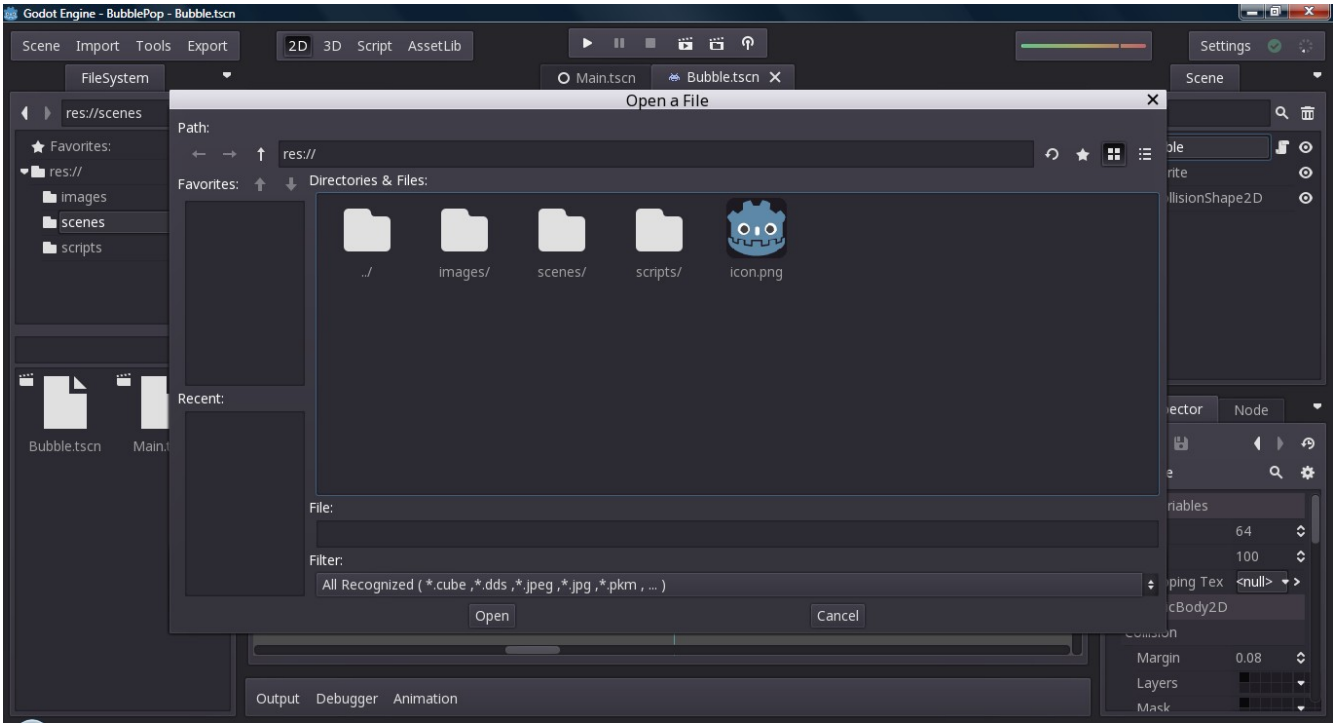
Now if we select our “Bubble” node, we will see this:



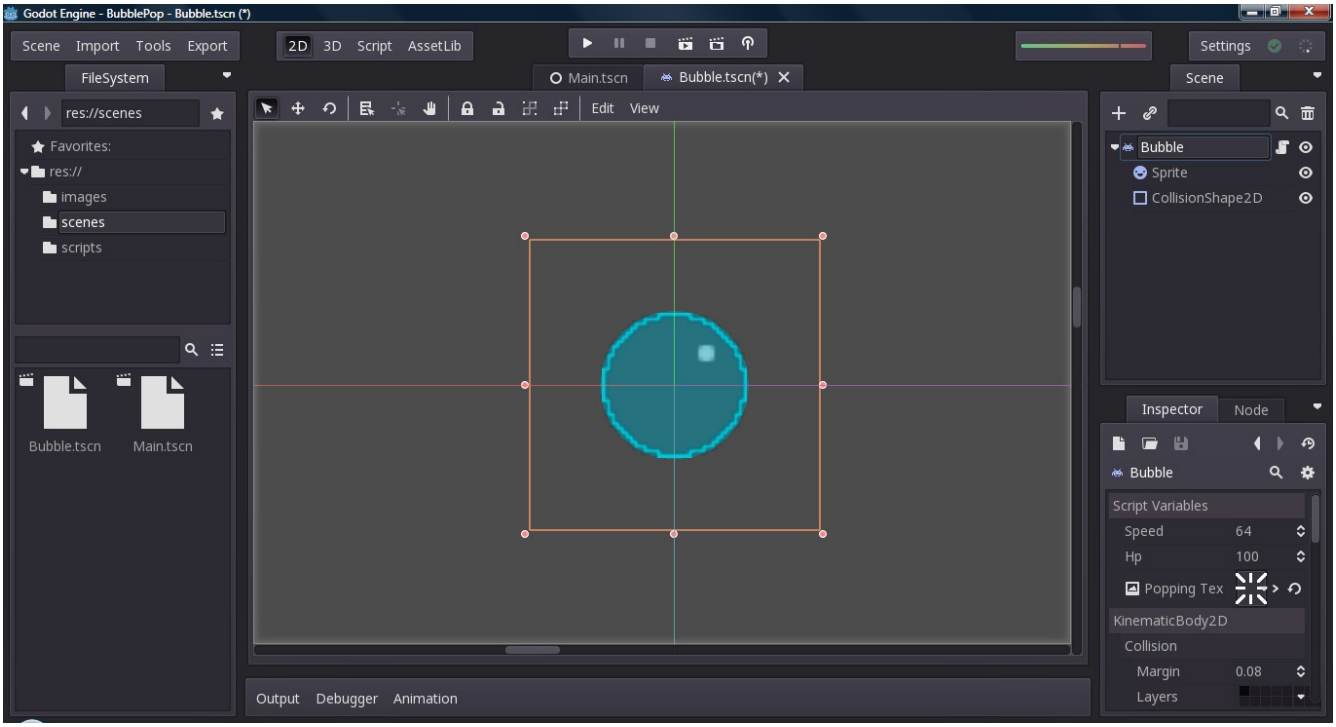
Click the box beside the “Popping Tex” property:



Choose “Load”:



Now select the “popping-bubble.png” file and you will see this:



Notice that now our popping bubble image is displayed beside our "Popping Tex" property. This means that when we run our game, the value of our "popping_tex" variable will be the popping bubble texture. Now we can add more code to our "add_hp" function:

```
func add_hp(inc):
	#Update the HP of this bubble
	hp += inc

	#Did the bubble pop?
	if hp <= 0:
		#Change the bubble's texture to the popping texture
		get_node("Sprite").set_texture(popping_tex)
```

We can now change our bubble's texture to the popping texture by passing "popping_tex" to our bubble sprite's "set_texture" method. We also need to modify our "_fixed_process" function so it will call "add_hp" when 2 bubbles collide:

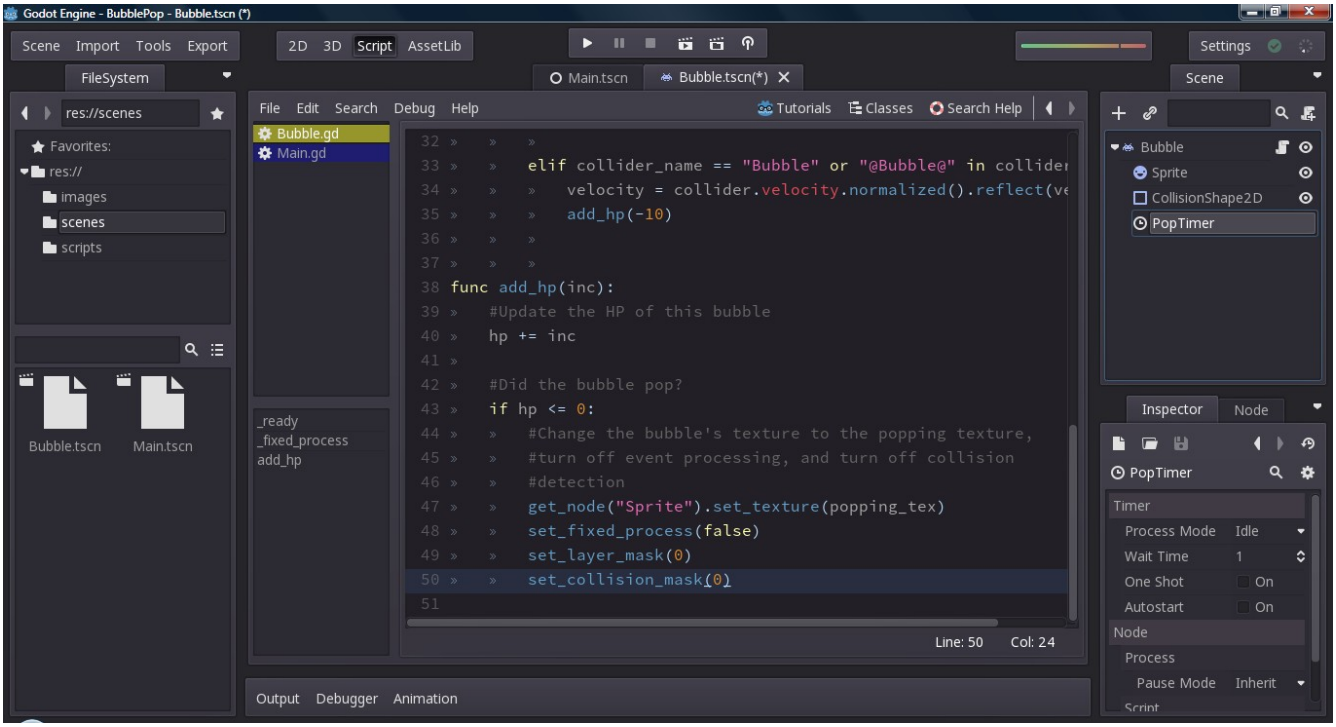
```
func _fixed_process(delta):
	#Update bubble position
	move(velocity * delta)

	#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)

	elif collider_name == "Bubble" or "@Bubble@" in collider_name:
		velocity = collider.velocity.normalized().reflect(velocity)
		add_hp(-10)
```

We will pass -10 to "add_hp" so that when 2 bubbles collide they will each lose 10 HP. However, if we run our game for a bit, we will notice something is obviously incorrect. After the bubbles pop, they continue to bounce around and collide with each other. We will need to modify our "add_hp" function to fix this bug:

```
func add_hp(inc):
	#Update the HP of this bubble
	hp += inc

	#Did the bubble pop?
	if hp <= 0:
	   #Change the bubble's texture to the popping texture,
	   #turn off event processing, and turn off collision
	   #detection
	   get_node("Sprite").set_texture(popping_tex)
	   set_fixed_process(false)
	   set_layer_mask(0)
	   set_collision_mask(0)
```

To make a bubble stop bouncing, we will disable fixed event processing. Doing so will cause our "_fixed_process" function to no longer be called for the bubble that popped. Setting both the layer and collision mask to 0 disables collision detection.

If we run our game again, we will notice that when a bubble pops it will stop moving and colliding with other bubbles. However, the popped bubbles remain on the screen. What we want to happen, is for a popped bubble to disappear after a brief amount of time. Let's start by adding a new timer called "PopTimer" to our bubble object:



We will use this timer to call a function that deletes a popped bubble after a set interval. Since we only need the timer to fire once, go ahead and turn on the "One Shot" property in the lower right pane. Then attach a new function to the "timeout" signal of our new timer:

```
func _on_PopTimer_timeout():
	#Free this bubble
	queue_free()
```

The "queue_free" function marks a node for deletion. It will then be freed as soon as the physics system and all other sub-systems are no longer using it. We also need to modify our "add_hp" function so that it will start our pop timer:

```
func add_hp(inc):
	#Update the HP of this bubble
	hp += inc

	#Did the bubble pop?
	if hp <= 0:
	    #Change the bubble's texture to the popping texture,
	    #turn off event processing, and turn off collision
	    #detection
	    get_node("Sprite").set_texture(popping_tex)
	    set_fixed_process(false)
	    set_layer_mask(0)
	    set_collision_mask(0)

	    #Start the pop timer
	    get_node("PopTimer").start()
```

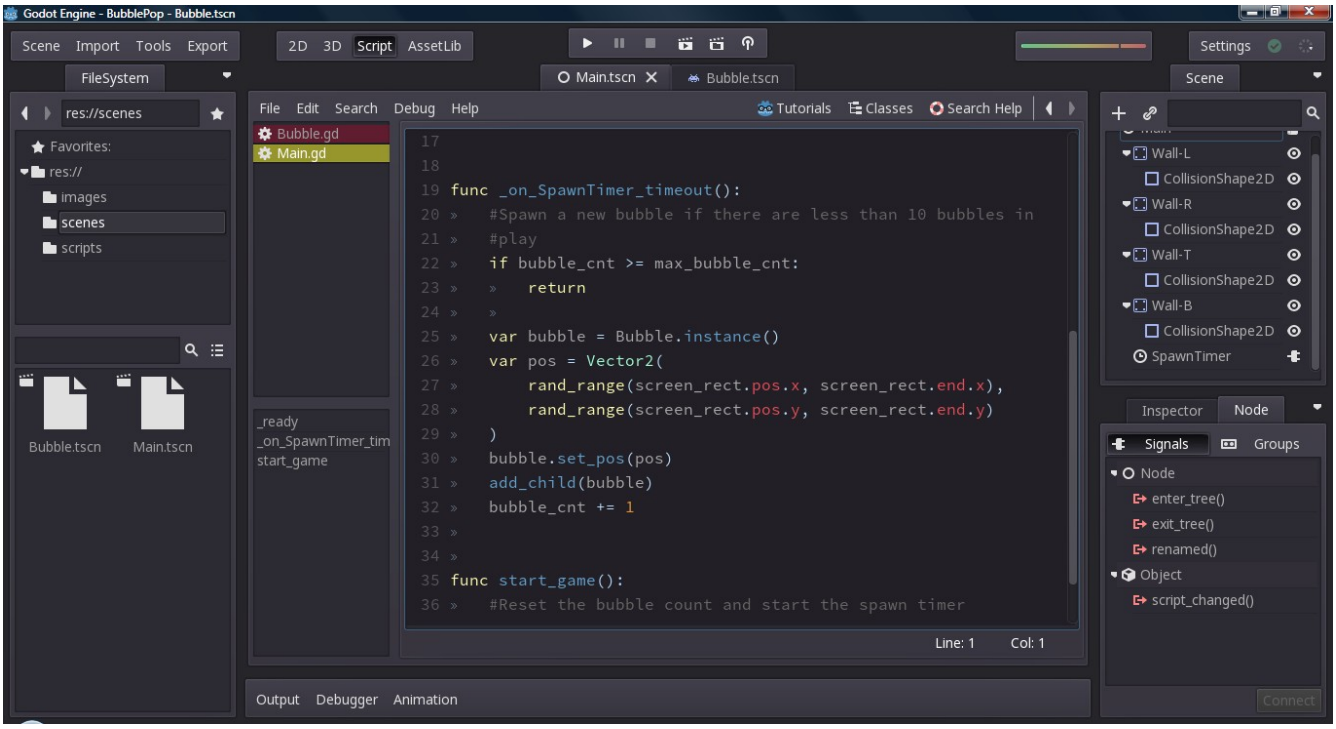
If we run our game once more, we will notice one more problem. Even though the bubbles pop correctly now, no new bubbles can spawn. The reason why this happens, is because we also need to decrease our bubble count when a bubble pops. But our bubble count is controlled by the script attached to the "Main" node, whereas our bubble popping is controlled by the script attached to our "Bubble" node. How can we update the bubble count then? The answer is simple, we can create a custom signal and attach a function in our main script to it. Let's start by declaring a new signal in our bubble script:

```
extends KinematicBody2D

signal popped

export var speed = 64
export var hp = 100
export (ImageTexture) var popping_tex

var velocity
```

To declare a custom signal, we simply use the "signal" keyword followed by the name of our custom signal. A custom signal can also take parameters just like a function if we want it to. However, this time we won't need any parameters for our new signal. Now let's add another line to our "_on_PopTimer_timeout" function:

```
func _on_PopTimer_timeout():
	#Emit the "popped" signal and free this bubble
	emit_signal("popped")
	queue_free()
```

The “emit_signal” function allows us to emit any signal that has been declared on the current node. If a signal requires parameters, we list them after the signal name. Now let’s open our “Main” scene again:



Now we can attach to our new “popped” signal. But wait a minute… how can we attach to an object that is not yet in the scene tree? Let’s start by creating a new function called “_on_Bubble_popped”:

```
func _on_Bubble_popped():
	#Decrease the bubble count by 1
	bubble_cnt -= 1
```

This new function will simply decrease our bubble count by one after a bubble has popped. Now we need to update our “_on_SpawnTimer_timeout” function:

```
func _on_SpawnTimer_timeout():
	#Spawn a new bubble if there are less than 10 bubbles in
	#play
	if bubble_cnt >= max_bubble_cnt:
	return

	var bubble = Bubble.instance()
	var pos = Vector2(
	   rand_range(screen_rect.pos.x, screen_rect.end.x),
	   rand_range(screen_rect.pos.y, screen_rect.end.y)
	)
	bubble.set_pos(pos)
	add_child(bubble)
	bubble_cnt += 1

	#Connect "popped" signal
	bubble.connect("popped", self, "_on_Bubble_popped")
```

To connect a function to a signal via code, we call the “connect” method of the object that can emit the signal. The first parameter is the name of the signal we want to connect to, the second parameter is the object containing the function we want to be called when the signal is emitted, and the third parameter is the name of the function to call when the signal is emitted. If we run our game now, a new bubble should spawn shortly after any bubble pops.

In the next lesson, we will make our game interactive by getting input from the player.