

Lab 0 – Catch the Creeps

Due Friday at 5PM of the week this lab comes out

The goals here are to help you get setup with Godot and get exposed to a number of basics of Godot.

Introduction

If using your own computer, make sure to install Godot 4.2.1. You don't need the .NET version for now. If you're on a lab computer Godot 4.2.1 should already be installed - open it.

Like other game engines, Godot has a wealth of useful tutorials. For the first lab, we will use one of these. Open a browser and visit:

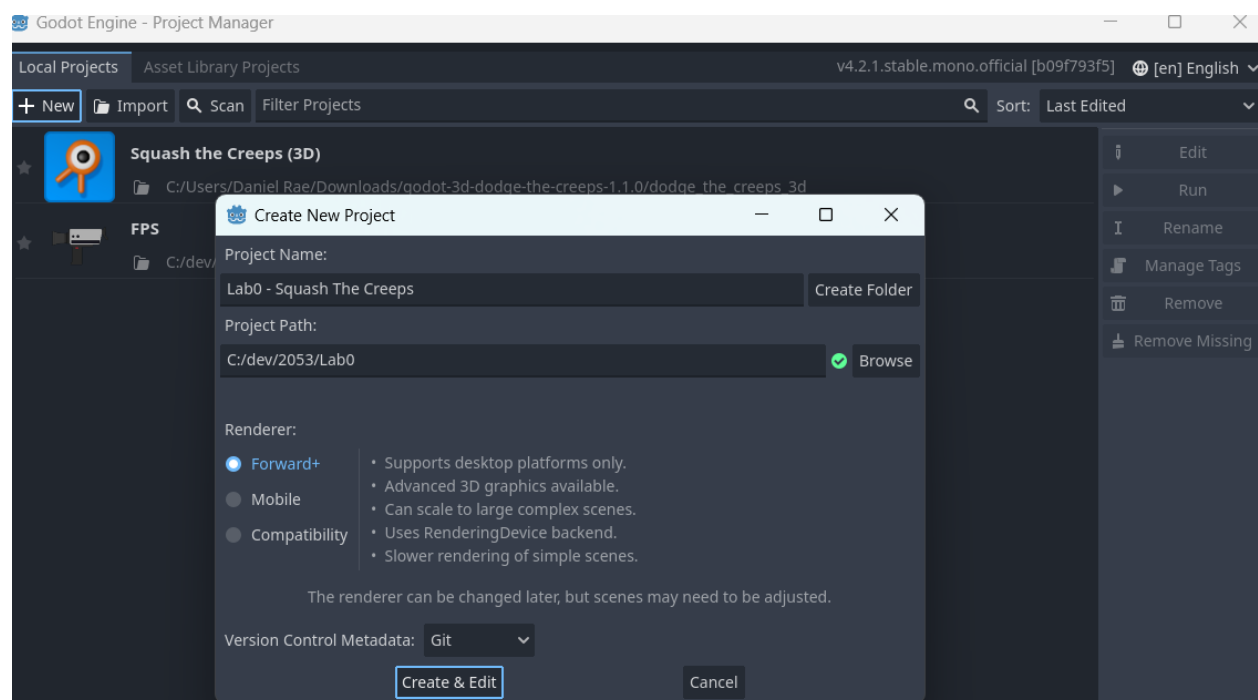
https://docs.Godotengine.org/en/stable/getting_started/first_3d_game/index.html

Follow along and don't forget to download the game assets linked below (assets are resources like 3d models, sounds, and more). While the demo has working source code, I require you to develop the demo yourself to gain experience. That said, start with "squash_thecreeps_stat_1.1.0.zip" here: <https://github.com/Godotengine/Godot-3d-dodge-the-creeps/releases/tag/1.1.0>

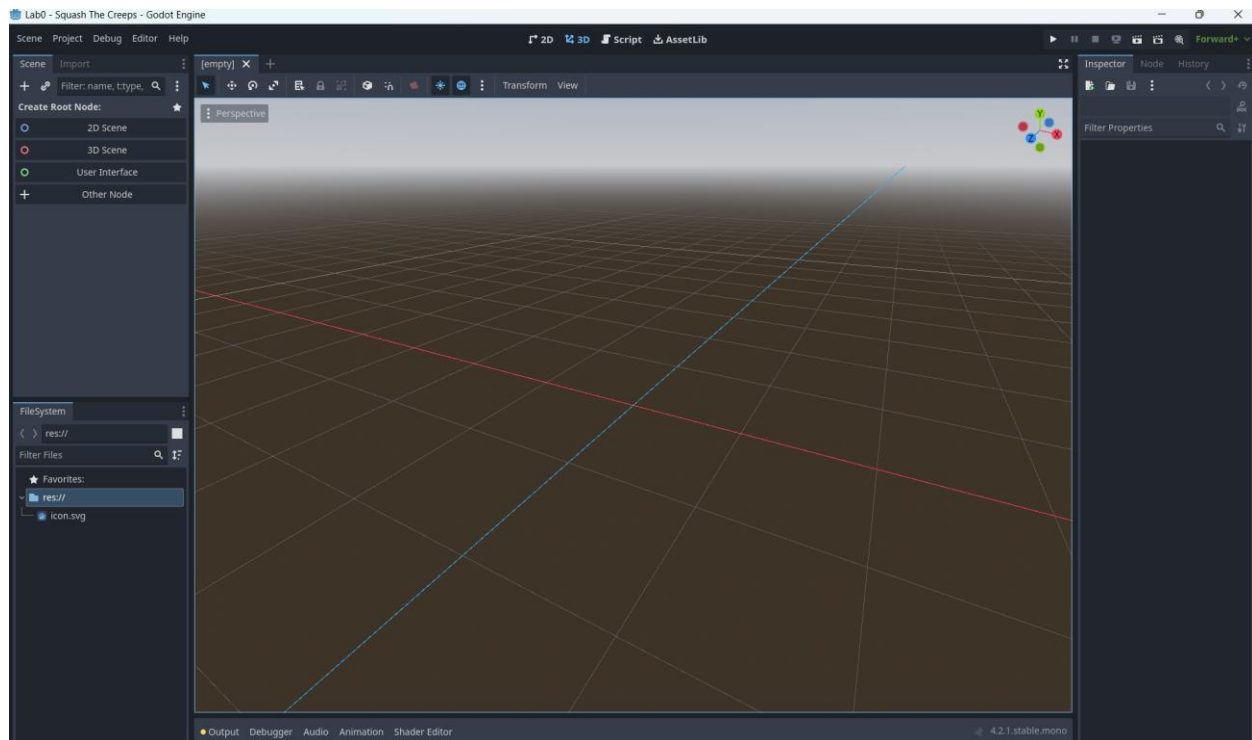
Later, if you want to see final source code for Godot3.x:

"Godot-3d-dodge-the-creeps-1.1.0.zip"

We will not be completely following the tutorial as it's a bit out of date and covers a bit much for today. Once you have the assets, open Godot. You will see the Projects screen. We will create a New project, let's name it Lab0 - Collect The Creeps. Navigate to a good place to save it, and create an new empty folder for this project (as it will create many files within), and create the project. The settings should look something like this, though your name or path may differ:



After opening the new project you should see something like this:



On the left we see the Scene Hierarchy and the FileSystem. First, the file system helps you organize all the many files involved in your game. The base folder is "res" for "resources" - files and data used in games are often referred to as resources or assets. You'll notice it looks like the same layout of your project folder, and in fact will automatically reflect changes in the folder in the editor. Go to the unzipped assets folder and copy the "art" and "fonts" folder into the folder you created for this project. When you return to the editor you should see the folders automatically appear in the FileSystem. You can always quickly go to the folder the project is stored in by right clicking in the file system and selecting "open in file manager".

Scene and Setup

Now, note the Scene Hierarchy on the left. It should be prompting you to create a root node. Select "Other Node" and search for "Node" (a plain Node type) and select it. It should now appear in the Scene as "Node", but I like to rename it through the right click menu and name it something to indicate its nature like "main" or "root".

Everything in Godot is made up of Nodes, and each Node can itself be a Tree of Nodes. Let's start the "Setting up the playable area" step in the tutorial. During these steps, note how you can create subnodes, and they give you properties to edit in the inspector window (by default on the left). In most game engines, objects are made up of these general objects that themselves contain other objects and variables that define *properties* of that object. If you want to read more, you can read about the Entity-Component system, which is a way of organizing code and code objects similar to Object Oriented Programming. Modern Game Engines generally combine both Object Oriented and Entity-component models of programming. The ground you make here (an Entity of Object type StaticBody3D) is composed of a

components `CollisionShape3D` and `MeshInstance3D`, each of which comes with its own set of properties and methods, like `Size`. After finishing the ground, the whole Game Scene is rooted to the "Main" node of just a plain type Node. Underneath it is the "Ground" Node, which by mousing over it you can see is of Node Type "`StaticBody3D`" - a generic type for any nonmoving 3D object. The ground itself has two components (the Mesh and Collision Shapes). Most game objects will be made up of multiple nodes and components like this.

Note the directional light should NOT be a subnode of Ground, but instead be a direct subnode of Main, a sibling to Ground.

Player Actions

Proceed to the next step - player character and actions. Follow the player scene creation as written. As an extra step, for your player character in the main scene, edit its transform to be slightly above the ground, say 1m. When making actions, let's skip the joystick part, but check back here and try it out if you want to use your own gamepads. For us, let's bind left to A (as in the tutorial), right to D, forward to W, and back to S. You can also use arrow keys, or do both arrows and WASD.

Move to the next step.

Until now, we've been using the editor almost entirely. As discussed in class, some things are easier to do with the editor, and some within code. One is not better than the other, each has strengths and weaknesses. Get used to switching back and forth between coding and editing. That said, let's get into Godot coding.

For Godot, you can code in Godot Script (GDScript), or C#. Godot Script is quite nice to learn if you want, and C# is nice as it can be used in more places, including Unity. However, the built in code editor does not support autocomplete for C# and .NET version compatibility can be an issue. In general, Godot supports its own scripting language better. If you wish to try C#, you are welcome to do so but you will support yourself - I learned Godot with GDScript and can try to help with C# but it will be more difficult. If you wish to enable autocomplete for Godot in another editor, see https://docs.godotengine.org/en/stable/tutorials/scripting/c_sharp/c_sharp_basics.html#configuring-an-external-editor otherwise, you can code without it or use GDScript. For this class, we will use GDScript, which is similar to python. I also recommend using the built in code editor until you become more comfortable with Godot (I tend to use it most of the time)

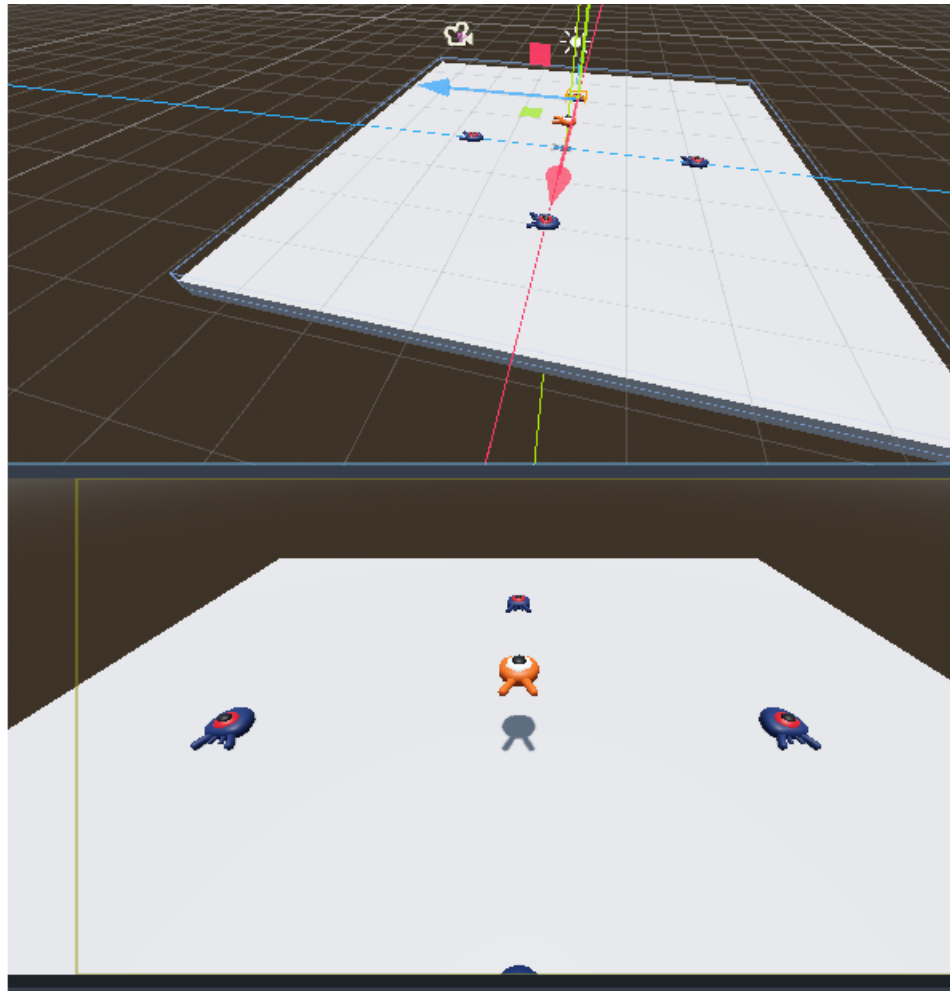
Test your code, you should be able to move your player in 8 directions and jump.

Having trouble? Ask the instructor or TA. Also, note you can access the built in documentation by highlighting an object type, method, etc, right clicking, and selecting "Lookup symbol" - it's very handy! Also, the editor has a built in debugger which is very useful. It's good to get familiar with these things, and to review your debugger skills if you have already taken CS2263.

Enemies

First, we will follow the tutorial for the final time: let's change our window size. As described there, select Project -> Project Settings -> General Tab -> Display -> Window. One of the first options is "size". Change the Viewport Height and width to 540 and 720 respectively.

This is where we will start to differ from the tutorial, to simplify it for this lab. Create an enemy like you did the player, making its own scene, importing the model (use mob.glb), making a collision sphere, and saving the scene. Now, back in your main scene, we will create 4 enemies around the player, something like this:



Note I've also set these enemies to be the same height above the ground for now: 1 m. Try out your game. The enemies look a bit boring because they don't move. Let's change that!

Note you already made 4 enemies by tediously instantiating child scenes multiple times. It would be a pain to update them all right? Luckily, if we edit the base scene, it will automatically update all other instances of that scene! So, let's return to the enemy scene you made, right click the base node and add an empty GDScript.

This will be quite basic. First, let's add an editable variable: rotation_speed:

```
@export var rotation_speed = 1
```

Then, we'll make our process update with the physics engine:

```
func _physics_process(delta):  
    transform = transform.rotated_local(up_direction, rotation_speed*delta)
```

Why multiply by delta? Try it without! Inspect delta and see what API says - it's the time since the last physics engine update, and this can be very quick, and also inconsistent. We'll talk about this later in class about the game loop.

Note while in an object, we have access to all its properties. The "transform" variable is the same transform that is in its Node3D (see the inspector tab), and we can access it with its variable name directly. This lets us leverage a lot of built in functions the Godot engine has developed for common game dev needs. Once again, get familiar and comfortable accessing and searching the API, it will save you a lot of time!

Save your enemy scene with the new script. Go back to your main scene and note that all your enemies have been updated to have the script you attached. Handy!

Try your game. The enemies should spin on the spot. You can also try to transform other ways...try rotated() instead of rotated_local()...what is different? Can you think of why?

Collisions

Godot physics works on a relatively unique system of masks and layers. Simply, every object that can interact with other objects physically must belong to a layer, and that layer can only interact with objects on layers specified in its Mask. For example, you may have Coins to pickup for an Italian Plumber character, and enemies to jump on. You could make a layer for the player, one for the pickups, and one for the enemies. You could set the player to interact with both the pickup layer and the enemy layer, but perhaps you would not allow the enemies to interact with pickups, so you would not include the pickup layer in the enemy mask, or vice versa.


For this simple lab, we will have three layers: one for the enemies, one for the world environment, and one for the player. Open project settings, and scroll down on the left menu in the General tab to find Layer Names. Select 3D physics (because we are making a 3D game with 3D assets), and name Layer 1 to be player and layer 2 to be enemies, and layer 3 to be world

Return to the player scene and select the player top level node. In the inspector (by default, the right hand menu), note you can see all the subnode properties as well. In the CharacterBody3D menu, expand "Collision" and you will see by default this node is on layer 1 and its mask contains only 1. Layer 1 is supposed to be for the player, so let's leave it on that layer. For the mask, we should ask "what objects can the player interact with?" In this lab, it will be everything! So let's make sure layers 1, 2, and 3 are selected in the mask. If you forget which is which, holding the mouse over the mask number will show your layer name we just added.

Let's do the Enemy Scene next. It should be on layer 2, and the Ground in the main scene should be on layer 3, but what should they interact with? For now, let's make their masks also 1, 2, and 3, but this is something to consider for later.

Now, all our objects should be able to interact with each other. So how does this work? Well, we can look up what an object collided with every frame. But how do we tell if we collided with the floor (from falling) or by catching the enemy?

Groups

Groups are a powerful way to tag different types of nodes so that they are easy to look up in a variety of situations. Go to the enemy scene, and in the inspector tab, note there is another tab beside it called "Node". Go to it, and select "Groups" right under it. Create a group "enemy".  Make sure you are on the enemy scene when you do this as it adds the current scene to the group.

Now, we can see if the player has collided with this group. Return to the main scene and re-open your player script.

We're going to add more to our `_physics_process` function, but it's getting a little large, so let's put this new code in its own function, which I called `_handleCollisions()`. Don't forget to call this from your `_physics_process` function!

My collision handler looked like this:

```
func _handleCollisions():
    # Iterate through all collisions that occurred this frame
    for index in range(get_slide_collision_count()):
        # We get one of the collisions with the player
        var collision = get_slide_collision(index)

        #for reasons we won't go into here, collision detection is
        #complicated and we may get a null collision, skip it!
        if collision.get_collider() == null:
            continue

        # If the collider is with an enemy
        if collision.get_collider().is_in_group("enemy"):
            var enemy = collision.get_collider()
            enemy.catch()
            break # Prevent further duplicate calls.
```

The comments should be explanatory. Note how we can check if an object we collided with is in a group. Any Node can be in a group, so this is quite useful to sort through your nodes, and you can make many groups if it helps you. Note we also asked to call a `catch()` function we haven't defined yet. Let's add it to our enemy script.

Add this at the top of your script to define a *Signal*

```
# Emitted when the player jumped on the mob.

    signal caught

# you're old code here

func catch():
    caught.emit() #sends the signal that an enemy has been caught
    queue_free() #delete this node later this frame
```

This emits the signal so we can take actions elsewhere because of this, then cleans up the enemy node.

Try your code, you should be able to intersect with the enemies and catch them. If you simply go over top of them, make sure your player or enemy collision spheres are large enough to touch.

Scoring (UI)

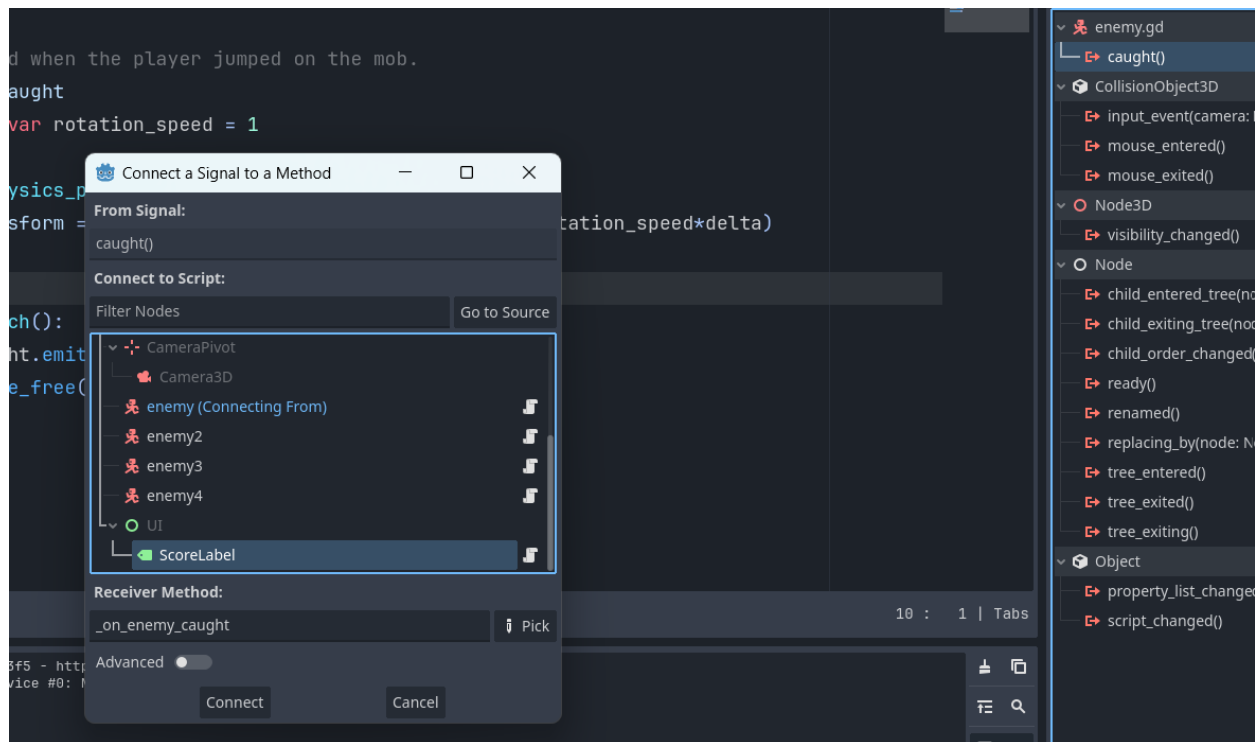
You can follow along at the later tutorial step:

https://docs.Godotengine.org/en/stable/getting_started/first_3d_game/08.score_and_replay.html#creating-a-ui-theme

Just create the UI score element (ignore the other steps). Once you get to the "keeping track of score", come back here. We skipped a part of the tutorial to simplify this lab, so a lot of the rest of the tutorial won't make sense here. We'll go our own way here.

Okay, now that we have your score text, we need it to change as we catch the creeps. In Godot, we can create events (Signals) that others can react to. This is part of the Observer software design pattern, and enables Event Based Programming which is very useful for games as they have to react to many things dynamically.

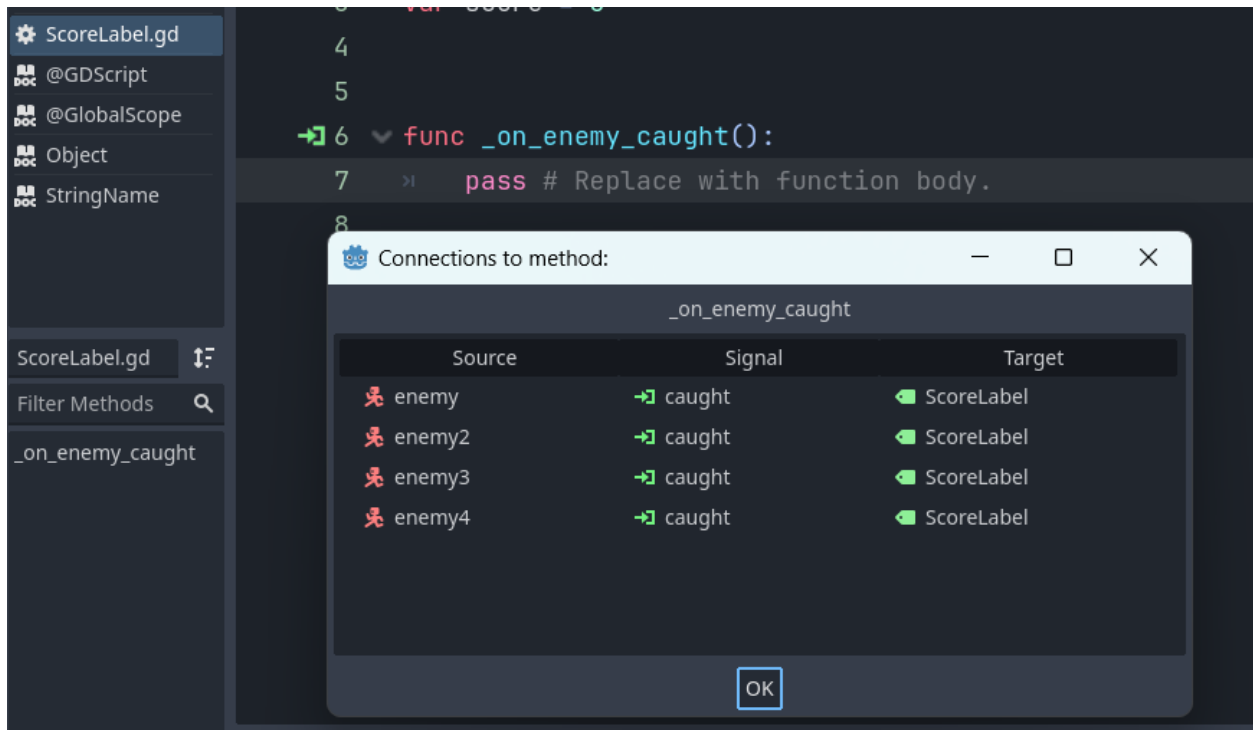
Click on one of your enemies. In the right-hand inspector menu, switch to the Node tab and underneath it, select Signals. These are all the signals the node you selected can emit. Note you should be able to see "Caught", which we made earlier during the collision section. Double click `caught()` and it will prompt you in a new window to create a function. You'll see your scene tree - find `ScoreLabel` and select it with "connect"



This will take you back to your ScoreLabel.gd with a new function. Now, every time the "caught" signal is emitted (remember, this is emitted when the player collides with an enemy), this function will be called. Handy! Do this for all 4 enemies. However, note that when you try to connect the signal to a function for enemy2-4, it tries to create a new function like `_on_enemy2_caught()`. That's no good – we want the same function to be called for all of them. While ScoreLabel is selected in the Connect dialogue (see the above image), pick "Pick" and select the `_on_enemy_caught()` function you already made.

We'll find out how to do this programmatically in another lab. For static game assets, this dialog is the easiest way to do it.

Now all 4 enemies should be connected to the `_on_enemy_caught` function. To check this, look at the function code in ScoreLabel.gd. You should now see a small green link symbol by the function. By double clicking it you can see all signals linked to this function.



There! Now let's update the score and text when a creep is caught. Change the function `_on_enemy_caught()`:

```
func _on_enemy_caught():
    score += 1
    text = "Score: %s" % score
```

Test your code. Your score should now update as you catch creeps!

Ending and restarting the game

If you want to restart, currently you have to close and reopen the game. Let's add a popup message window when the player finishes and a restart button.

Right click your root UI node and add a child, "Accept Dialog" and name it "WinScreen". It will default to invisible so you can't see it. In the inspector, change the OK Button Text to "Restart" and add a congratulatory text like "You win!!!" in the "text" field. Finally, in the Window option pane, change "Initial Position" to a nice option like "Center of primary scene" or something that looks good to you.

Now, let's make it appear on win. In the `ScoreLabel` script, add a new variable:

```
@export var winScreen : AcceptDialog
```

This tells Godot this variable should be a Godot Node of type `AcceptDialog`. Save, and you should see `winScreen` in the inspector (you may need click to a new node and back to refresh it). Drag the `WinScreen`

node from the scene tree to this variable. This is assigning the Node to that variable at game setup. In the enemy_caught function, we can now make it visible when we meet the max score.

```
@export var MAX_SCORE = 4
@export var winScreen : AcceptDialog
... #old code here
func _on_enemy_caught():
    score += 1
    text = "Score: %s" % score

    if score == MAX_SCORE:
        winScreen.visible = true
```

You can test your code. When the game ends, your dialog should become visible, but that's it, even if you hit okay, nothing happens. Let's change that.

Now, click the WinScreen node in the scene tree and connect its **Confirmed()** signal to a new function we'll make in the scoreLabel.gd script. That function will simply reset the main scene tree:

```
func _on_win_screen_confirmed():
    get_tree().reload_current_scene()
```

Try it out!

Recap

Phew! That was a lot!!! It's okay if many things seemed confusing to you this time. Try to go over the code and things we did in the inspector. Change some code or values and see what happens. We'll be repeating a lot of these things, learning more about some of them, and just getting used to others. However, you should be able to make some basic things now in Godot! Remember to consider what you can do in the editor, the inspectors, and code.

Reflect on what you've done. We've learned about:

- Godot Nodes and the Scene Tree
- Collision shapes, meshes, handling collisions
- `_physics_process()` function, `move_and_slide()`
- Basic Lights

- Input
- Scenes as subcomponents in a main scene
- Adding meshes
- Camera basics
- Attaching scripts
- Basic Godot scripting
- Communicating between nodes with Signals
- Basic UI
- Event based programming

Now, you may be confused by much of this, but we'll get more familiar over time. However, I really encourage playing around with this game, adding things, and keeping it handy to refer back to as you go on.

Want more to do? Well here are some ideas, in order of challenge:

Extra Challenges (Optional)

- 1) A) Try to make the enemies move as well to make the game more challenging.
 B) Can you make the enemies only interact with the player and not other enemies?
Hint: think of Masks!
- 2) Can you stop player movement when the Win screen pops up?
Hint: add a Signal!
- 3) For *extra* challenge, only allow yourself to catch them by jumping on top of them.
 if you do this, try making non-jump collisions "kill" the player (game over)
 -these are actually in the full tutorial, but try to figure it out yourself!
 -of course, you can always just look at the full tutorial
- 4) Spawn additional enemies (this will require quite a bit of learning)
- 5) Make enemies move in random paths