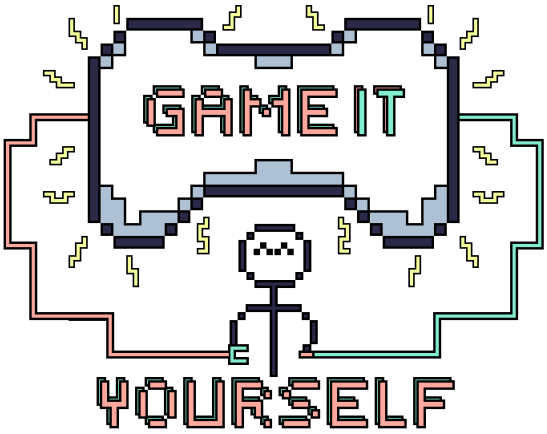

Game-It-Yourself Toolbox

Level 3: Immunity Bandit

Have you ever wondered how your body defends itself against the viruses and bacteria of the world? Have you ever wondered how 3D games were made? Or if you could be the one making them? In Level 3, you will be designing your own game to conquer the human bloodstream, while you learn about the microscopic world of viruses and the secrets of 3D game development! As your virus speeds through the veins, you will add obstacles to slow down the virus's progression and add a scoring system to check on your progress and hit the high score. Tell your own story through the eyes of your virus helper. Grab your mouse, unleash your creativity, and embark on this journey into the world of viruses and 3D game development!

universidade de aveiro
DigiMedia digital media and interaction research centre
FCT Fundação para a Ciência e a Tecnologia
COMPETE 2020
PORTUGAL 2020
UNIÃO EUROPEIA Fundo Social Europeu

This activity seeks to promote awareness about healthcare and the microbiology of viruses. Create, adapt, develop and design a 3D digital game. This program was created in the context of the research project **PLAYMUTATION2** (UIDB/05460/2020) and is directed towards adolescents (12-18 years old) and young adults (19-35 years old) to promote the development of **STEAM** competencies: Science, Technology, Engineering, Arts and Mathematics. **HIV Runner** is a 3D game that takes inspiration from Subway Suffers, where the player is put with the vision of the HIV virus infecting a person. Learn how to create your 3D world, give it light and a character that you can control, to help the spreading of the infection.

Level of skill: [Intermediate]

Skills covered in this course

Concepts:

→ Familiarization with viruses and the circulatory system;

→ Present facts to the player about HIV;

Game Development:

→ Familiarization with Godot 4.1+ 3D game engine components;

→ Import models to your game;

→ Learn the input system;

→ Control character movement;

→ Spawn objects in run time;

Programming:

→ Learn more logic structures to control the flow of execution of the code (sequences, lists);

→ Use input handling to control character movement;

→ Protect your code with error handling;

→ Diagnose and fix bugs, compilation errors, exceptions and code that doesn't perform as expected;

→ Comprehend and apply object-oriented programming principles.

Plan of Activities

- **Mission 1:** The world has 3Dimensions!

Learn what components your 3D game now needs. Give your world light, environment and a sky! Edit your sky and customize your lights.

- **Mission 2:** Where did the game elements go?

With your world set, you need the obstacles and the points! Add a pill as an obstacle, and T Helper Cells as points! Learn how to spawn endlessly any game elements and how to import any model into your game.

- **Mission 3:** I would like to Move Now!

With the game set and ready to be played, all that is left is to add the player movement and the game is ready to launch! In this mission, you are going to learn input mapping, and how to react when an input is made to make your character jump.

Contents

1 The world has 3Dimensions!

1.1 Mission description

The first step to creating your game is to understand the game concept. In our new game, where you play as an HIV virus, you travel through the circulatory system of the human being having the goal to contaminate the host and conquer as many T-Cells as possible. This comes from the fact that HIV uses the T Cell to replicate itself and launch again into the system, destroying important cells of our immune system. Making this connection in the game is a challenge, but we can simplify it by just transforming the hitting of a T Cell with the controllable character into points.

In this stage, we want to set the world, light and environment where the game will occur, the game map and its infiniteness which we will look up ahead now. As mentioned in Level 2, if you find yourself lost or in need to explore further, here are some suggestions of where you can explore:

- Godot's updated documentation
- Official Godot active forum
- Stack Overflow
- Youtube Tutorials - GDQuest

1.2 Goals

1.2.1 Open the Project

To embark on Level 3, let's start by opening the Godot project as we learned in Level 2. If you need a refresher, it is recommended to go back to the guide in Level 2. Here's a quick step-by-step reminder on how to open the project:

- Have Godot 4.1+ on your computer;
- **Download** the project file to a location that you remember;
- **Extract** the project file if it is in a compressed file;
- **Open** Godot Game Engine;
- Click on **Import** on the right side;
- Click on **Browse**;
- **Navigate** to where your project is;
- Click on the "**project.godot**";
- Click on"**Import & Edit**";

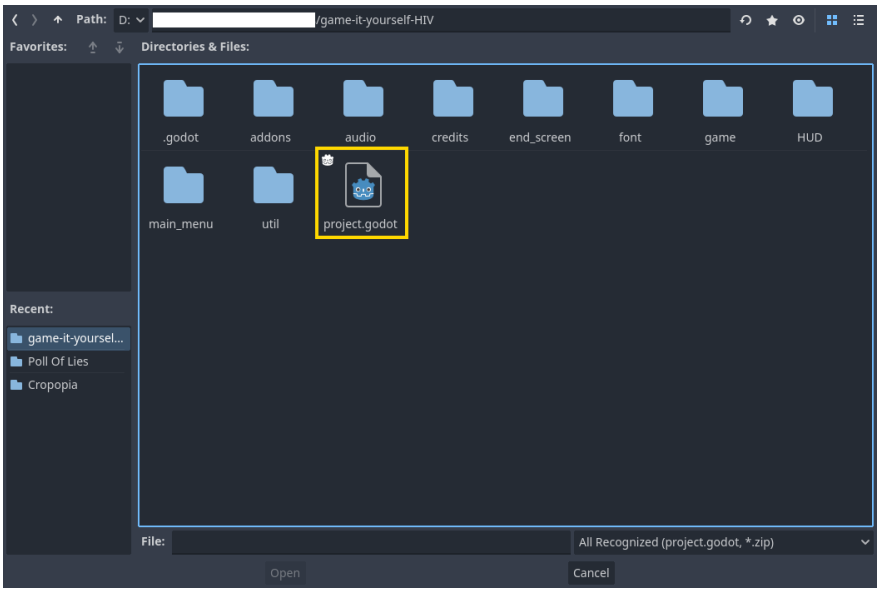

Figure 1: Navigate and open **Immunity Bandit** project file

Your project is now open!

1.2.2 Remind me again of the Godot's Interface

This small chapter is just to remind you of some of the more important components of Godot's Interface, but also to teach you that Godot's Interface has most of the game engine interfaces can be changed around. This serves the purpose of customization to the user's liking! In Level 3, the default positions of Godot are going to be used as can be seen in Figure 2. The main changes from the interface shown in Level 2 are the following:

0. **Workspaces:** Choose the view of your scene: 2D, 3D, script editor or dialogues.
1. **Scene hierarchy:** It shows all the objects that compose your scene, which you can rearrange in order, add or delete, make (in)visible or add scripts to.
2. **Filesystem:** This is where all the files of your project are: scenes, images, sounds, scripts, *etc.* You can also add new files here.
3. **Inspector:** Every object in your scene has properties, like size and color, all of which can be inspected and changed here.

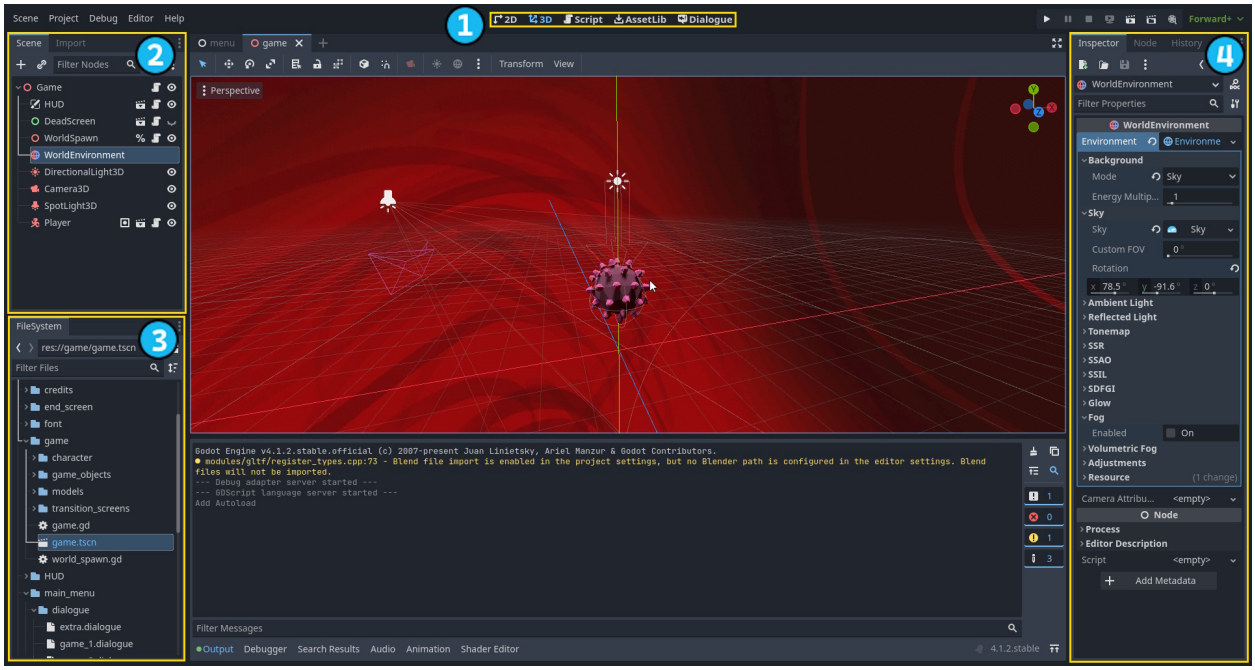

Figure 2: Gogot's Interface: Main components

1.2.3 Investigate the 3D project setup

Well, well, well, looks like you already have things set and ready to go. But wait, when I get to the game, it's all black! Why is that!?!? (Figure 3) It looks like some game elements got corrupted (Oppsie!). Looks like I need you to help me fix it!

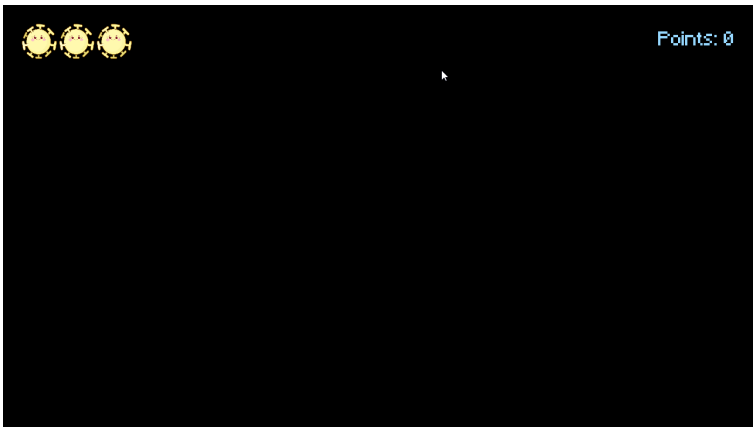

Figure 3: First game run: I can't see anything!

Let's investigate why everything is so dark! Perhaps there's something **missing** from the **game scene**. To take a look, open the "**game.tscn**" file using the **Filesystem**. Double-click it to open the game scene (see Figure **??**)

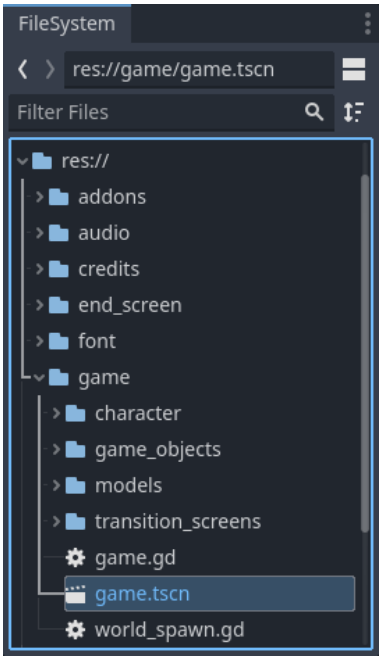

Figure 4: Go to Fylesystem to open the game scene

In Figure 5 it is represented the **3D workplace** with the following **scene components**:

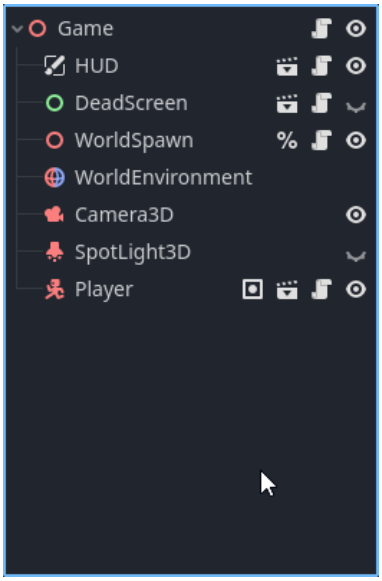

Figure 5: Main 3D Node Scene: Child nodes that make the game world

Before starting to fix the bugs our developer left us, let us understand each **component** and what they do inside our game:

- **Game** - It's the parent node of the scene;
- **HUD** - Head-up Display is another scene that contains user interface elements that display game-related information to the player in a non-intrusive way;
- **DeadScreen** - Is a small scene that plays an animation everytime you die;
- **WorldSpawn** - Contains the script that generates the world elements as you explore, so the game never ends.
- **WorldEnviromnent** - It allows you to control various environmental effects and settings that affect the overall appearance of your game or scene. Example of this: Sky colour;
- **Spotlight3D** - Emits light in a cone-shaped pattern. This means that they can be used to illuminate a specific area of the game world while leaving the rest of the world in darkness;
- **Player** - This is a scene that contains our playable character. Further down in this level, we will learn how to control their movement.

In the Figure 6 bellow and your 3D Workplace you can see the current state of the game, and it looks like there are some missing elements.

The **Spotlight** to the left is not active, the **Camera** is in the centre of the screen, the **Background** of the game is just black, aside from that I left you a clue that something else is missing and needs to be added, but we will get there!

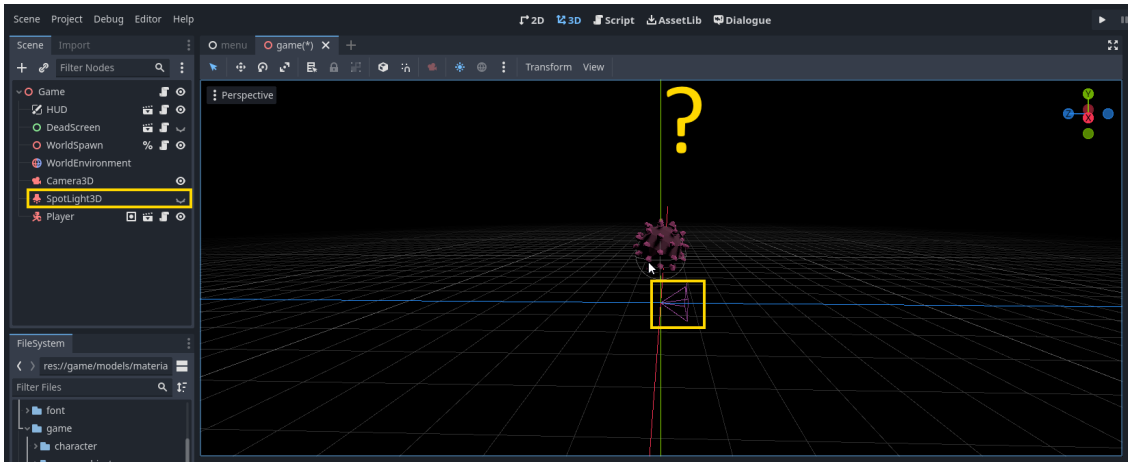

Figure 6: Current game state with missing elements

1.2.4 Let there be light!

First, let's give the game some **light**! Start by enabling the **spotlight** in the scene, try to run the game and check if you can see something now.

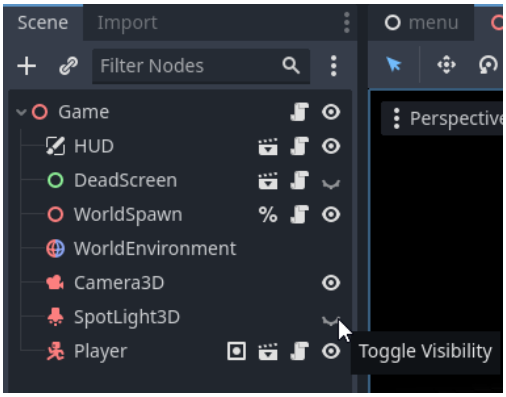

Figure 7: Activate Spotlight3D in game scene

Now that the spotlight is enabled, let us check the light inside our workspace!

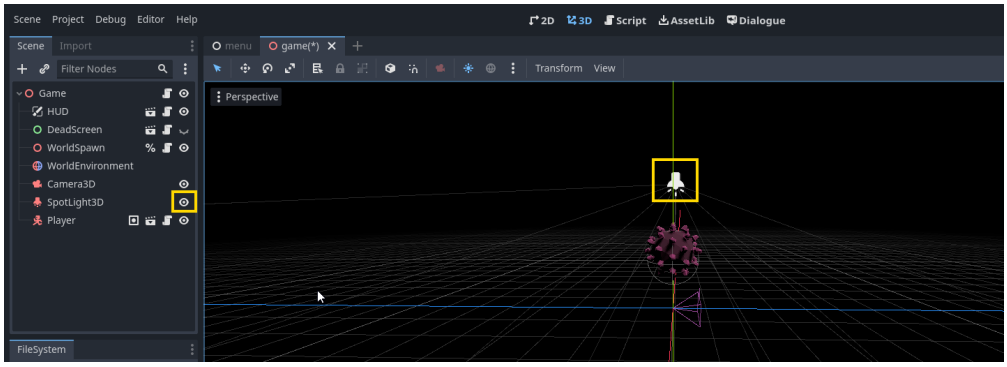

Figure 8: Spotlight active and visible on the 3D workspace

Let's give it a go and check what we can see inside the game now!

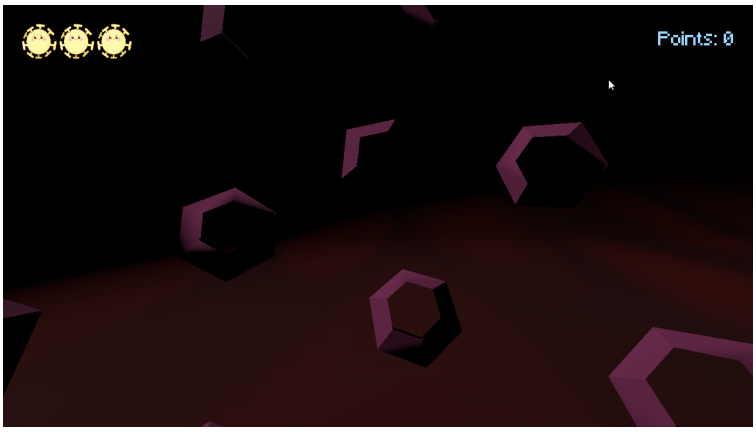

Figure 9: Game running with Spotlight active

Well... It still looks dark, doesn't it?
This is where it comes to the missing element pointed before. In 3D digital games, we have to think carefully about the **lightning** in our environment for that you are going to learn **Spotlights** and **Directional Lights**. These will be the building blocks for all the games you'll create in the future.

Spotlight, this type of light works exactly like a lamp that you probably have in your bedroom, a **focus of light** that illuminates everything in their path, for example Figure 10.

Figure 10: Spotlight

Directional light, a type of light very important in a game world because of its **global illumination**, it emits light with parallel rays uniformly in a specific direction across an infinite distance. In simpler terms, you can think of the sun as an example, a faraway light that illuminates **everything equally** and at the **same angle**, an example of this can be seen in Figure 11.

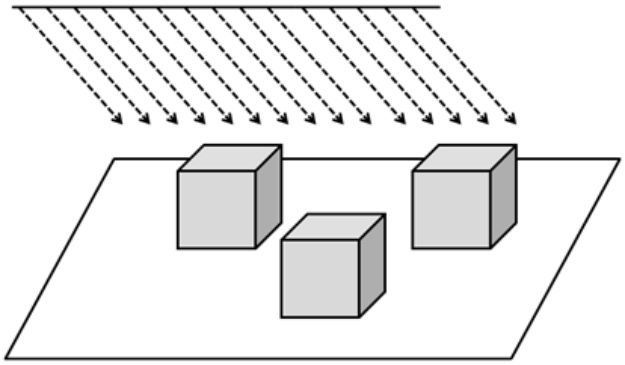

Figure 11: Directional Light

Now you know that our game is missing a global illumination source! That important element allows us to see the world clearly. So let's add a **Directional Light** to our game scene. First, go to the scene right-click the **Game Node** and select "**Add Child Node**" as seen in Figure 12.

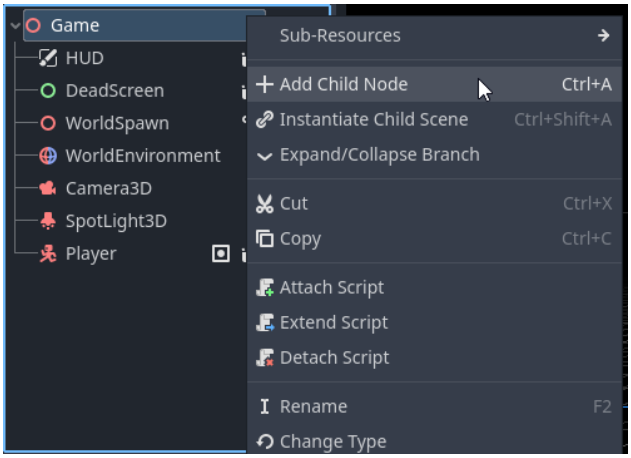

Figure 12: Add Child Node to the Game Scene

A **new window** will appear (Figure 13), where a chosen node can be added. In this window, you can **search** for the type of node you are looking for. This is always useful since there are a lot of types of nodes and can be hard to find the one you need.
For our case, let us search for "**light**", and look in the list for **Directional Light 3D**. Once selected, click on **Create** and a Directional Light will appear on the scene.

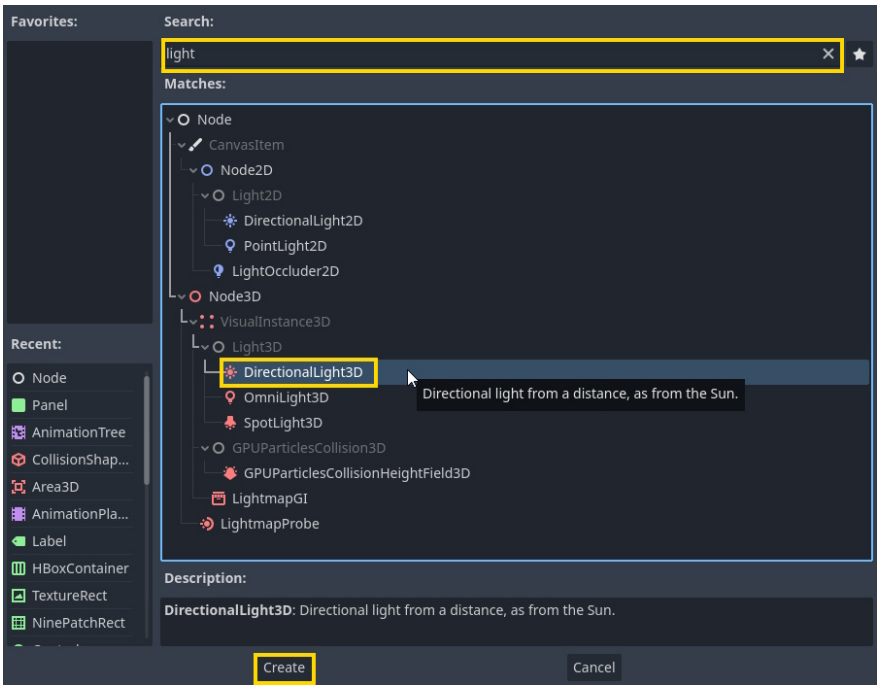

Figure 13: Search for Directional Light 3D in Nodes

Once the light is added, if you try to run, you can see more of the world around the virus, but the floor still isn't quite clear, and if you check the **arrow** with the **direction of the light**, you can see the light is **not pointing to the floor**! So, let's **click** on the **DirectionalLight3D** (Figure 14), on click the inspector appears and here you can change the direction of the light and other values.

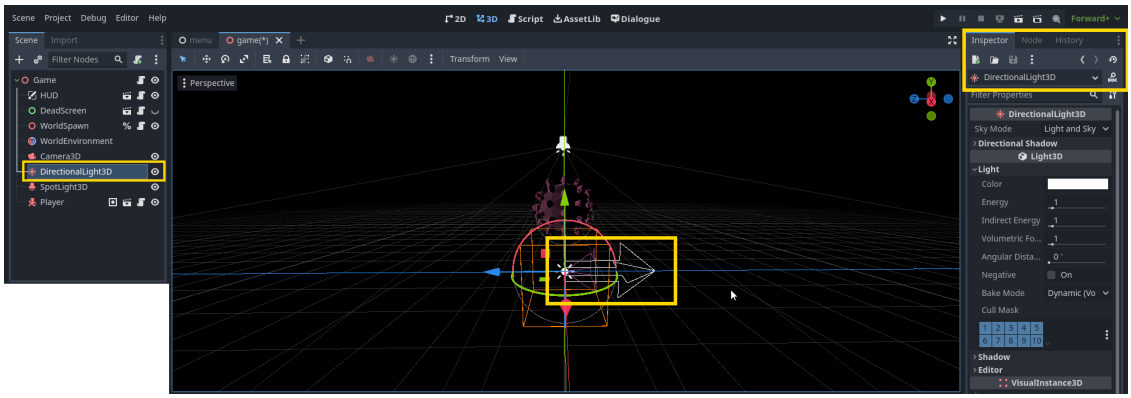

Figure 14: Directional Light 3D in World Space

By changing some settings of the **Directional Light 3D** in the inspector, the quality of light in the game will improve. The items below are the modifications recommended for the light, but feel free to adjust the settings to your liking. Trial and error are a big part of game development and you alone can achieve better lightning settings than the recommended ones.

List of steps made to improve the Directional Light visible in Figure 15:

- Give more **Energy** to the **Light**:
  - Go to Light 3D and **expand** Light;
  - Now Color and Energy can be modified;
  - Give more **Energy** to the light;
  - You can change the Color of the Light too;
  - Feel free to adjust to your liking;
- It is recommended to **enable Shadows**:
  - Shadows always add some **immersion** to our games;
  - Shadows help be aware of the **distance** to the **floor**;
- Change the **rotation** of the Light:
  - **Expand Transform** field;
  - **Transform** is where the **position** and **rotation** of the elements in the game space can be changed;
  - Recommended **rotation** for **X** is: **-90** degrees;
  - Try other angles to check the effects of the lightning in your game;

- Change the **position** of the Light:

  - Directional Light is always **equal**;

  - Changing position **won't** affect the lightning;

  Changing the Y position can help us better **visualize** the elements inside the 3D workspace;

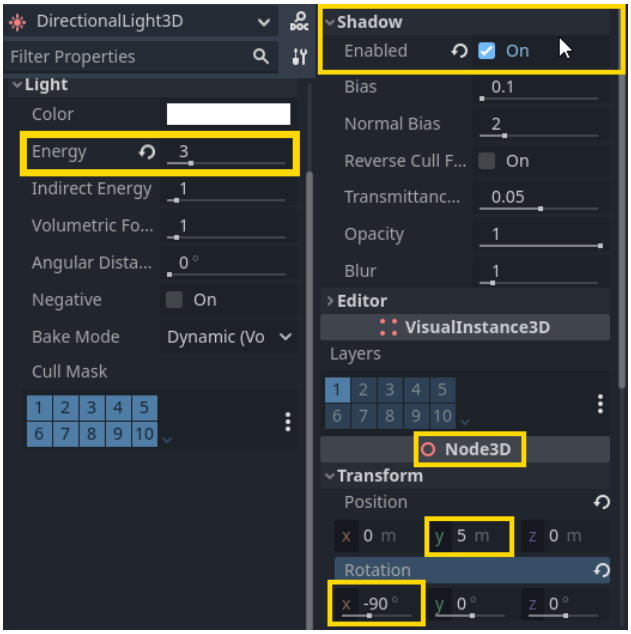

Figure 15: Settings of Directional Light 3D

In the case of you feeling lost in the values, where are the values of the final version:

- **Color**: White
- **Energy**: 3
- **Shadow**: On
- **Position Y**: 5
- **Rotation X**: -90

If you run the game now it should look like Figure 16, where you can see the floor!!! Directional light makes all the difference inside a game.

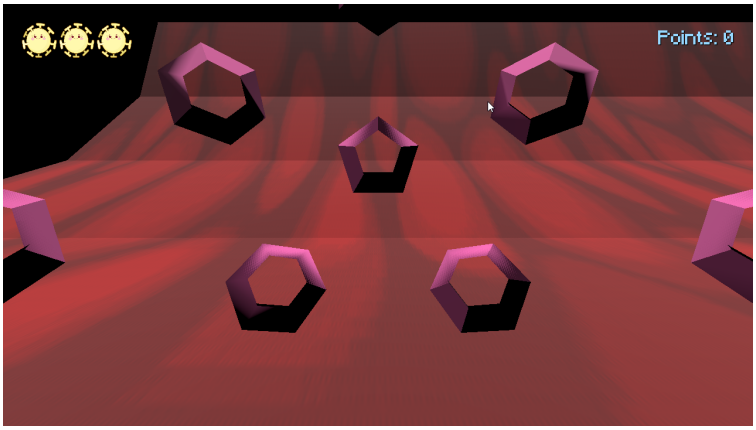

Figure 16: Game running with Directional Light present

1.2.5 Fix this weird camera!

The lighting in the game is perfect now! But, why do I see those empty hexagons on the screen?!? And where is the game itself?
Well, if you check the game world, the camera is at the centre (Figure 17), the same as the virus when it drops, so we are actually seeing the **virus from the inside**! This should not happen.

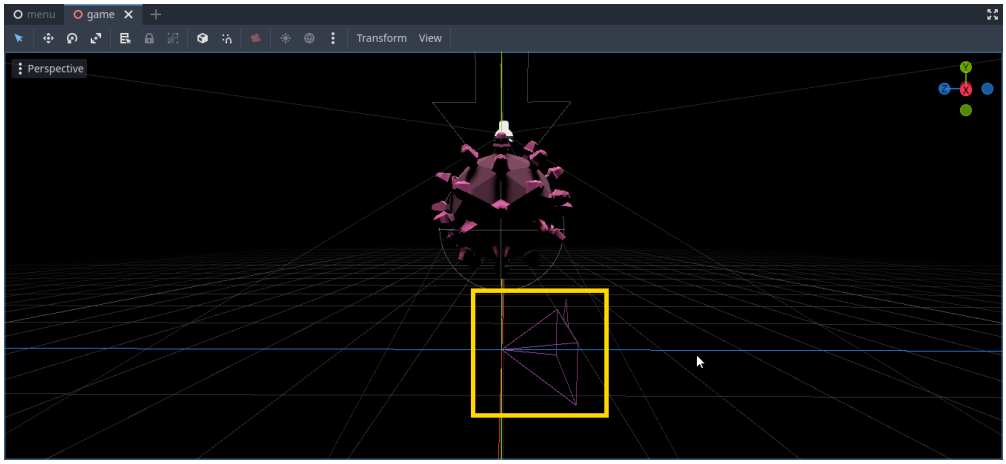

Figure 17: Position of camera in the game world

Now that you know how to modify the **Transform** of the game objects, this is going to be easy! Try to do it yourself, you have Figure 18 and a small guide below to help you out, but feel free to try your own angles and positions!

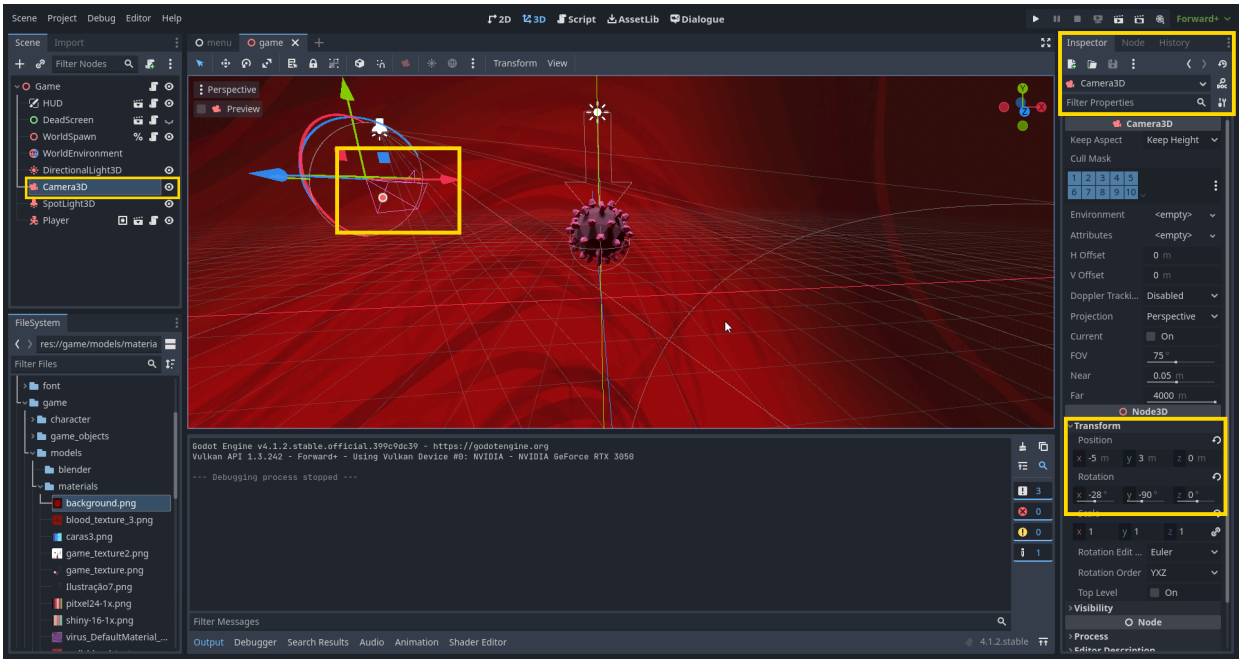

Figure 18: Camera Position

Here are the steps to change the position of the camera:

- Start by **selecting** the **Camera**;
- In the **inspector**, search for the Node 3D that contains the **Transform**;
- Change **position** to: (**X: -5; Y: 3;** Z: 0)
- Change **rotation** to: (**X: -28; Y: -90;** Z: 0)

Don't forget that you can adjust the camera to a position and angle of your liking. Explore and run the game to check if it gives a good view of the game field. With these steps, we already have a normal view of the game field as seen below:

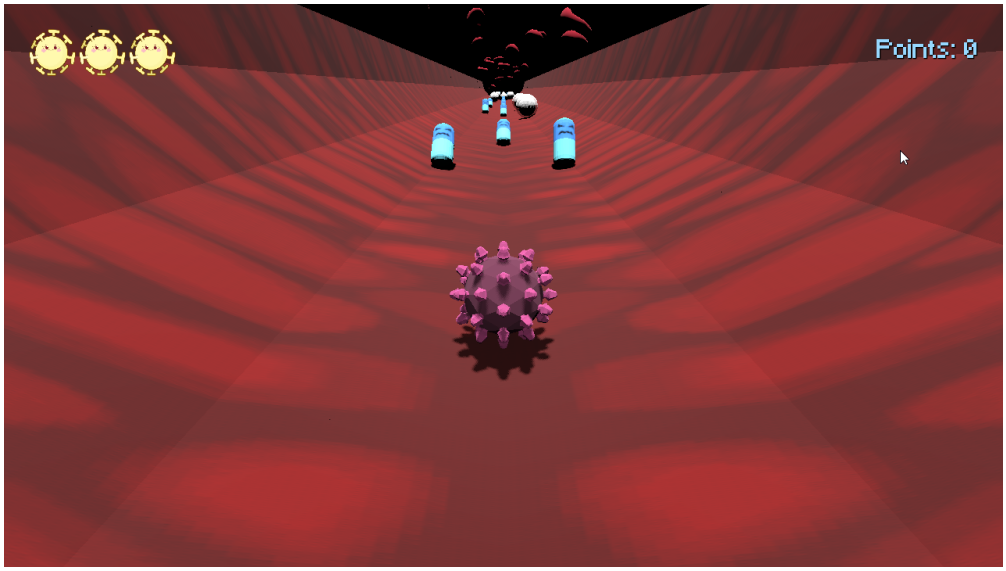

Figure 19: In-game field view

The developer is playing tricks on your mind, you still don't have the Pills and the T Helper Cells in your game! But your game view should look like Figure 19.

1.2.6 Add your sky to the game

In the case of our game, the game field is covered by the veiny walls, but in most cases we have an open world with a sky, sometimes even a night and day sky.

That is also applied to Immunity Bandit: in the top part of the wall, **it is dark**, because the sky outside the walls is black. If we change the **sky colour** to something more appropriate we will see the **difference**.

To do so, we can change the world environment to have a sky with a fitting image. Start by selecting **WorldEnvironment** in the game scene as seen in Figure 20.

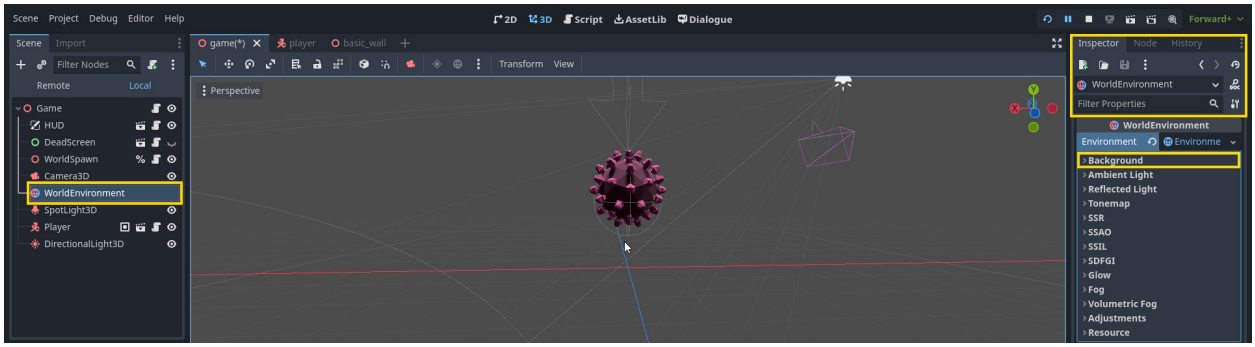

Figure 20: Open WorldEnvironment Inspector

In the inspector, **expand** the **background**. Here there are different modes that can be chosen to apply to the sky colour, including a **custom colour**. But for Immunity Bandit we have an image for the background in mind. To do so **select** the mode "**Sky**" as seen in Figure 21.

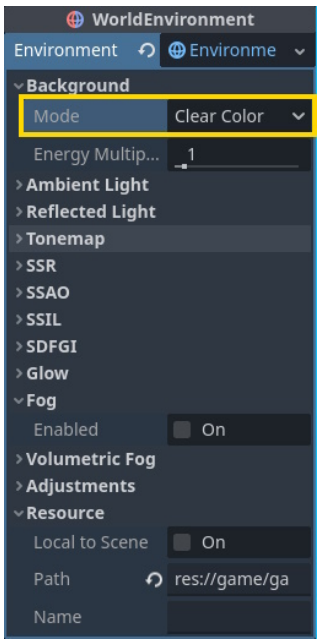

Figure 21: Change background mode to Sky

After selecting the mode "Sky", a new menu **Sky** will appear, **expand it** and more options will appear. Here **select** a "**New Sky**" as seen in Figure 22.

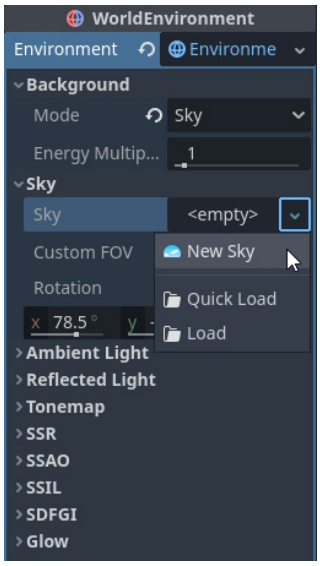

Figure 22: In WorldEnvironment add a New Sky

After a New Sky is added, click up the "**Sky**" with the cloud symbol, check Figure 23 and a new expansion will occur. Here add a **New PanoramaSkyMaterial** (Figure 23), which can contain images and apply them as a game background.

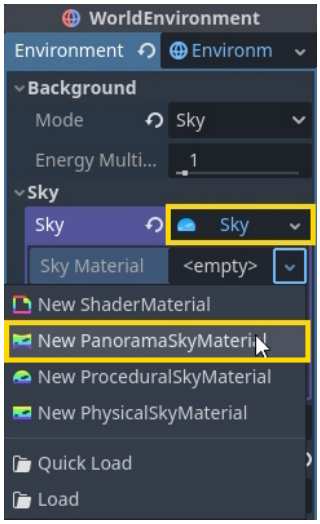

Figure 23: In Sky add a New Panorama Sky Material

The World Environment is ready to receive a new image as a sky background. Go to the **filesystem**, **search** for the background image called "**background.png**", you can drag it over to the panorama field as seen in Figure 24. This will apply the background image to the colour of the sky.

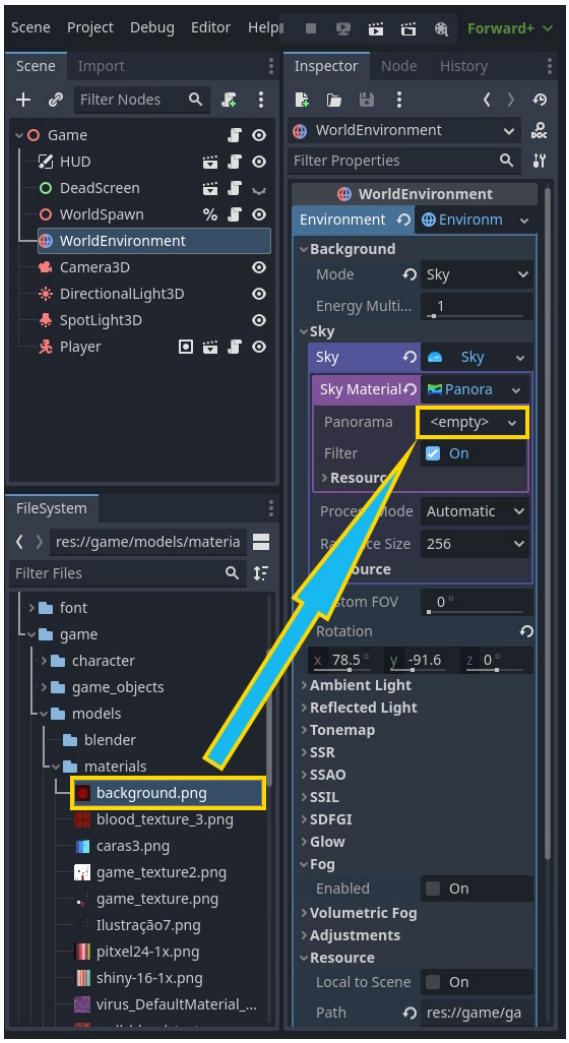

Figure 24: Background colour being applied to the sky

In Figure 25 the final result can be seen, since the game happens inside the human body, flowing through the veins, this background is red, going towards the dark in the centre, giving a sense of distance.

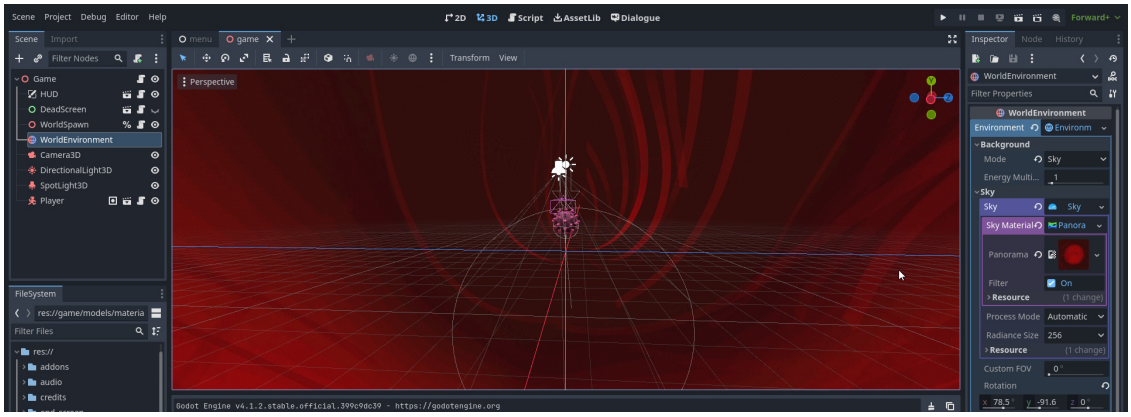

Figure 25: 3D workplace with background image applied

When you run the game, this is what the final result will look like:

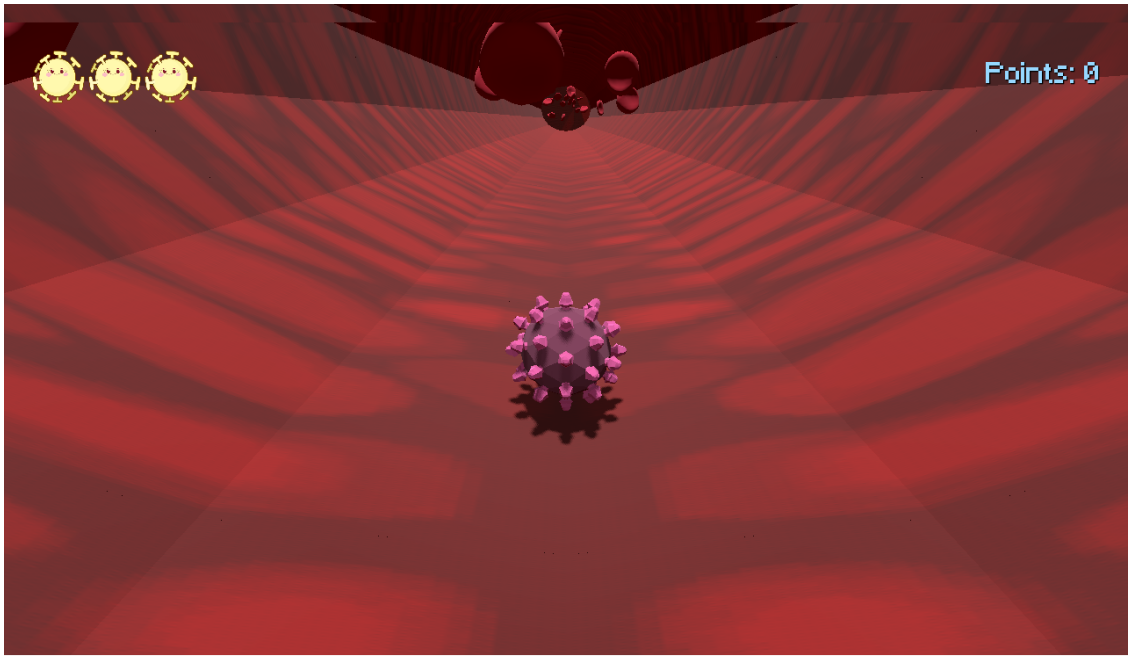

Figure 26: Game running with environment set

Incredible! The world is set up and ready to be played!

But... wait... Where did all the obstacles go!?!? I swear I saw them in the other Figures. It looks like the developer made another mistake and removed the spawn of game elements. :c

Only you can help me now! In the **next challenge**, you are going to learn how to **spawn** these **game elements** and maybe replace this developer who keeps adding bugs to the game!

1.3 Rewards

Current XP: 30

Congratulations on completing the mission! This was great progress you have made and so much knowledge that you've gained throughout this journey. You've learned a lot about the 3D game space, from lighting to the position of game objects in the world. With this mission you know have the skills and creativity to create your own amazing game world. Take a break, you deserve it, The next challenge is on, but remember there is always something new to learn, so keep exploring and experimenting!

1.4 Summary

Table 1: Documentation and Help - Where you can find answers to your questions.

Godot's updated documentation	Available at Godot Docs
Godot Engine Forum	Available at Godot Engine Q&A
Stack Overflow for anything related to programming	Available at Stack Overflow
Godot 3D Lights and Shadows	Available at Godot Docs 3D Lights
Godot 3D Camera	Available at Godot Docs 3D Camera

In this chapter, you embarked on an exciting journey through the components that make the game world come to life. Each element takes a big role in how and what the player visualizes. Here's a summary of what you learned:

Game Element	**Summary**
Lightning	Shapes the atmosphere guide navigation, highlight objects and can even influence emotions. Creates an immersive experience for players.
Spotlight	Spotlight is a focused light that highlights a specific area or object.
Directional Light	Simulates the sun, casting sharp shadows and creating a realistic outdoor effect.
Camera	Controls the player's view, determining how they see the game and influencing their gameplay experience.
World Environment	Defines the overall lighting and atmosphere of the game world. Defines the colour of the sky as a texture image or as a solid colour.

2 Where did the game elements go?

2.1 Mission description

The world's foundation is set, but something happened to our game elements! In this mission, you are going to learn how to import 3D models to the Godot game engine, while creating obstacles while playing the game. This will transform your virtual realm into a fun and dynamic game experience. As an HIV virus, our aim is to infiltrate the T cells, to replicate ourselves and that translates into gaining points. But obstacles are a fun part of this gameplay. If you hit a pill (one of the obstacles) you lose a life. Let us start this mission and figure out what the developer left for us to do, so we can add back the game elements.

2.2 Goals

2.2.1 Why isn't my model in the game?!?

To start we need to find a way to insert the models back into the game!

What is this model I am talking about?

These models are the 3d models we use as game objects to represent our obstacles and targets that we can catch.

In this course, we don't go through the use and creation of 3D models since that requires different software, but we are nice enough to give you some references if you want to add your own, such as:

- Blender - Download Blender Software
- SketchFab - Download 3D models
- Free 3D - Download 3D models
- Youtube video - Export Obj from Blender

But the most important thing is to learn how to add those models into the game and for that, we will use what we have.

This code was developed to have some separation of concepts, which means, the developer of the game wanted to have the code, or more specifically the script, of what spawns the game elements such as T cell and Pills separated from the game logic.

This small practice helps a lot in the development of the future, just like in real life, keeping things organized can cost us some time in the present, but save a lot of time in the future!

So the developer separated the code of spawning things from the game logic. Let us check what happens there and identify the mistake!

So... In the **world spawn**, weird, there it is! The model of the wall, the blood cell, but where is the pill, and the T cell?

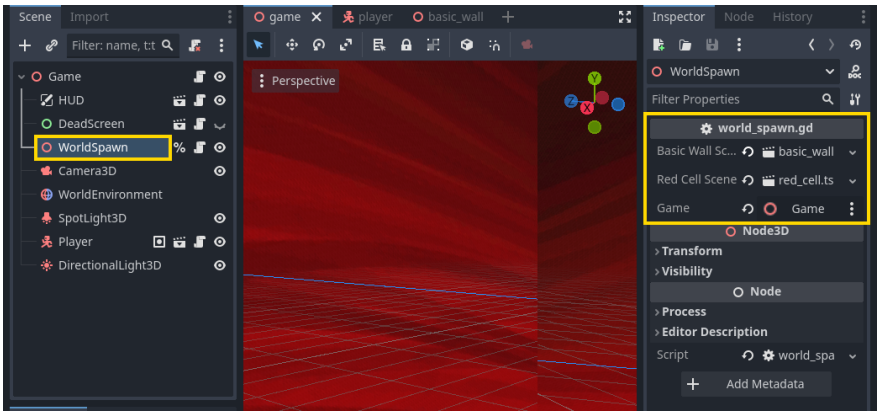

Figure 27: Game Scene with models added to the WorldSpawn

Let's check the code to see if something is missing. In the WorldSpawn Node, there is a paper symbol next to it, when clicked it opens the script. The selected script is highlighted on the left sidebar as can be seen in Figure 28.

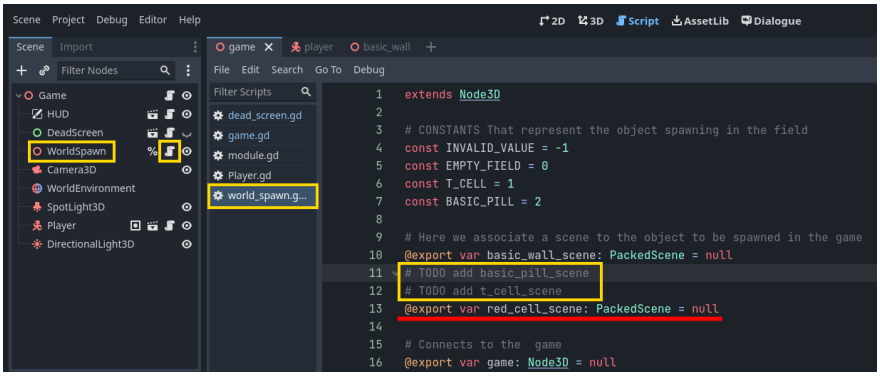

Figure 28: WorldSpawn script - Where the models are spawned inside the game

It looks like our programmer left us to work TO DO![1]

[1]TODO - It usually is a comment left by the developer to indicate that something needs to be done in this part of the code. It works as a reminder to the developer.

It looks like the code is going to be identical to the code of the red cells and t cells, but let us learn how it works by analyzing each part:

- **@Export** - This means a variable is going to appear in the inspector and can be edited there. In this case, a model can be assigned to it;
- **var** - Indicates that a variable is being declared;
- **name** - Name the variable to something you can understand easily what it is by reading;
- **:PackedScene** - It means that it is a variable that contains the indicated scene. Example of this: A Node called T Cell, that contains the model is a scene and can be assigned here so that in the future it can be spawned in the game world;
- **null** - It is a value that is attributed to the variable, in this case, it is empty;

Now that you understand how it works, try to do it for yourself. If you need help, you can check the Figure 29.

```
@export var basic_wall_scene: PackedScene = null
@export var basic_pill_scene: PackedScene = null
@export var t_cell_scene: PackedScene = null
@export var red_cell_scene: PackedScene = null
```

Figure 29: Export variables required for the game

If you check again the inspector of the WorldSpawn, you can see the field appears there, but it is empty! And you can only receive a Scene on that, not a model.

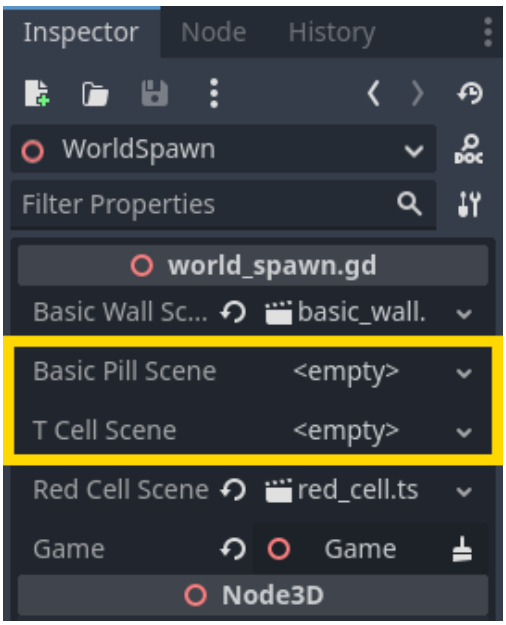

Figure 30: WorldSpawn Inspector with 2 more packed scenes

This is common in most game engines and there is a reason for it, Godot doesn't know the intentions of the developer with that model. Does it move? Does it float? Does it fall? Does it work like a wall and I can pass through, or is it a ghost model and nothing collides with it?

If the 3D model is intended to be a part of the world, it can be applied to a physics body to it. What is this physics body? It is an object that is subject to a simulation of the laws of physics, it is useful to simulate the behaviour of real physics such as gravity, movement and collisions.

In this part of the course we are going only focus on **Static Body**, but more information about the other types of physics bodies can be checked at Godot Physics Introduction.

Usually, to add a model to the game, it is required to first create a scene for that specific model, but in this case, the developer left us something already! (At least that!). So let us search for the T Cell Scene as seen in Figure 31.

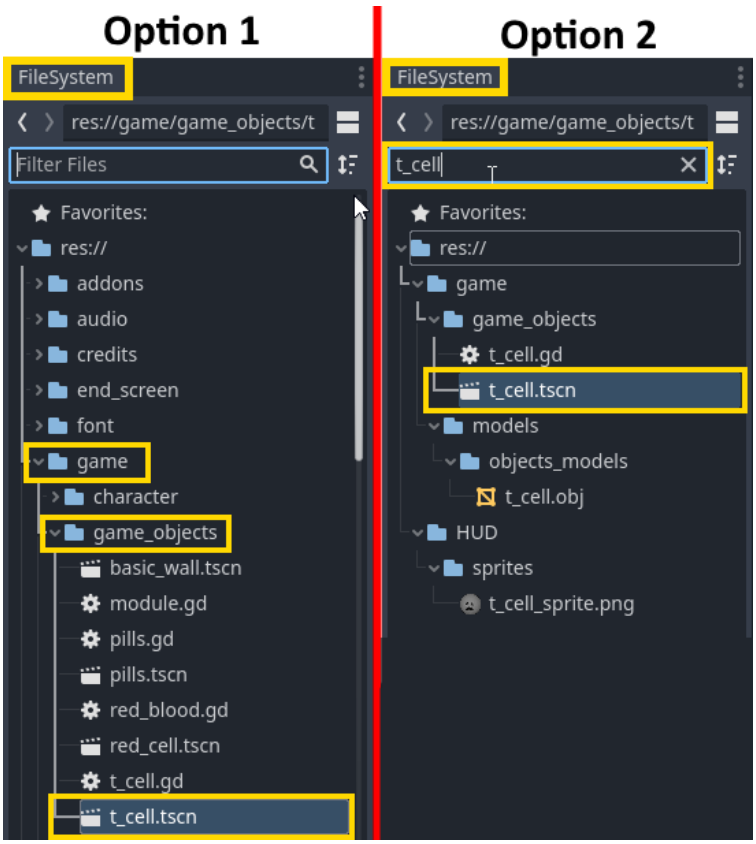

Figure 31: Search for "t_cell" scene in Filesystem

In Figure 31 there are two ways of searching for the "t_cell.tscn" scene, the first one is to go to the **game** folder, click it to expand, go to **game_objects**, click to expand and search for the "t_cell.tscn" scene, double click to open.
The second way is to write the name of the scene in the search bar and double-click to open the "t_cell.tscn" scene.

Figure 32 shows the T Cell scene with the following elements:

- **TCell 3D Node** - Parent Node of all the elements of TCell, contains the script to deal with collision with the player and movement of the T Cell;
- **MeshInstance3D** - A mesh is the 3D representation of the model ready to be represented in the game space. This type of node saves the model to be displayed in the game;
- **CollisionShape3D** - It defines a 3D shape, like a sphere for example, to serve as an area of collision with other game elements. For example, when the collision shape of the player hits the floor, it cannot move anymore;
- **Area3D** - It is an area 3D with a collision shape inside of it, but instead of colliding, it only detects other collision shapes that enter or leave the area. In the case of the T Cell the collision's purpose is to detect when the player enters the 3D area of catching the T Cell, gaining points; [2]
- **AnimationPlayer** - Where a small animation of the rotation of the T Cell is playing on repeat to give it more livelihood.

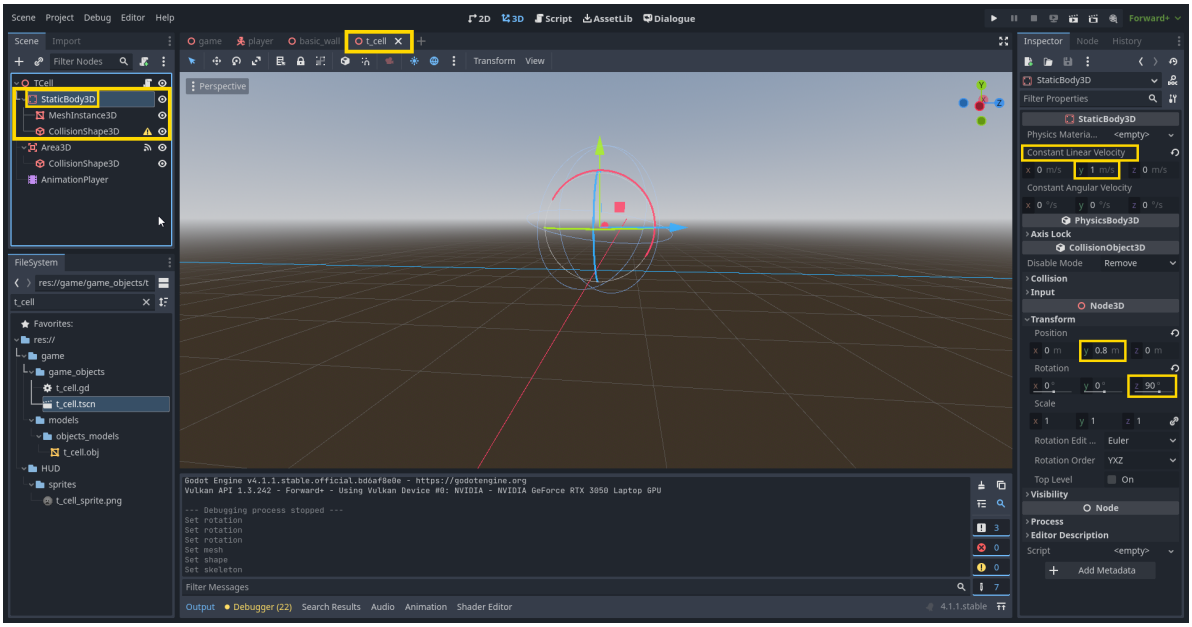

Figure 32: T Cell scene elements

With the screen open, the model is not visible, because the mesh is empty! If you click on the MeshInstance3D as seen in Figure 33, the inspector will show that the mesh is empty.

[2] This subject is more complex, for more information consult Godot Area 3D

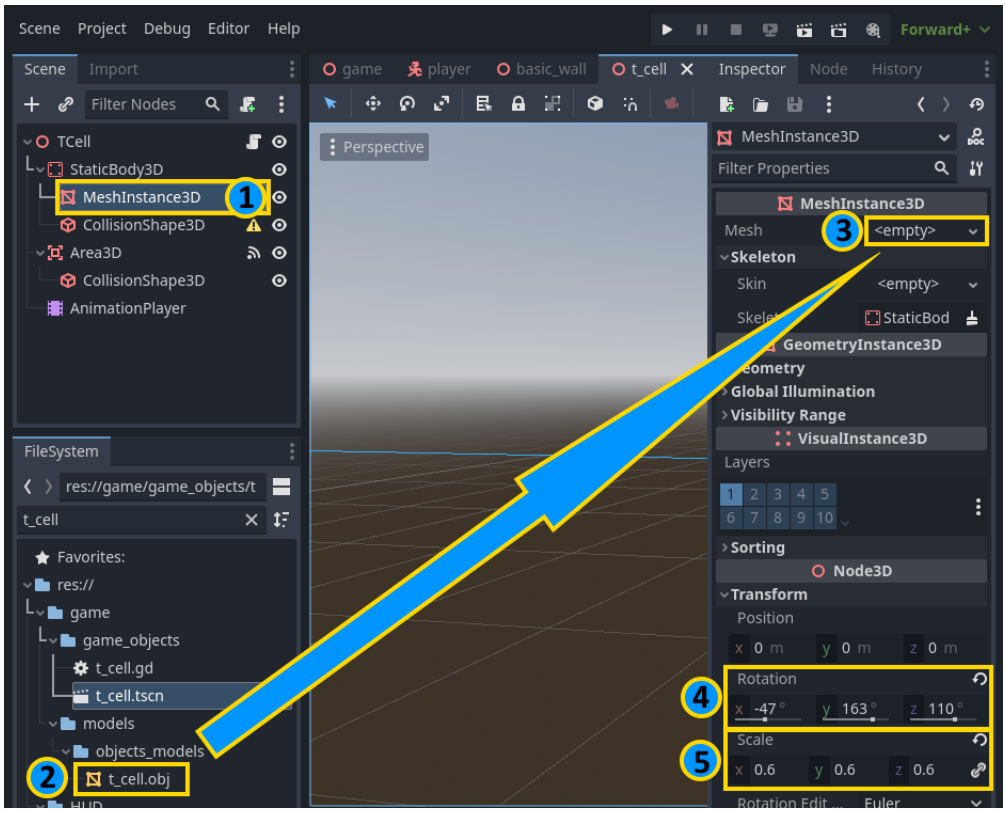

Figure 33: Assign T Cell mesh to Mesh Instance 3D

To assign the mesh to the MeshInstance3D, search for the ".obj" file that corresponds to the T Cell, in this case, it will be "t_cell.obj", and by dragging it to the Mesh "Empty" in Inspector, it will become assigned with the correct mesh and will now be **visible** in the **3D workspace.**

The developer at least left some values already set to help! But it is important to know what they do to the model.

So in Transform, because the T Cell has a scared face to show that it is scared of the virus, the rotation needs to be changed to face the player. The face is barely visible and it is hard to get the right angle, so there was much trial and error to find these values.

At last, the mesh is too big for the game field! We can adjust the size by changing the scale. 1 is the default value, if it is below 1, the size decreases, if it is above 1 the size increases, for this case 0.6 was the fitting for the game field.

To work on the **Collision Shape 3D** now. Start by selecting the **CollisionShape3D**, where in the inspector it will appear an empty shape in the collision as seen in Figure 34 .

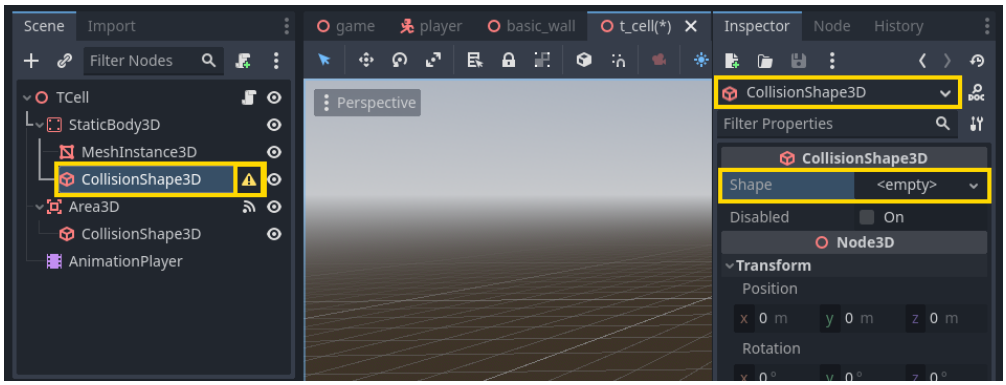

Figure 34: Define a Collision Shape to the T Cell StaticBody

Start by assigning a **New SphereShape3D** in the list of Shapes available as seen in Figure 35

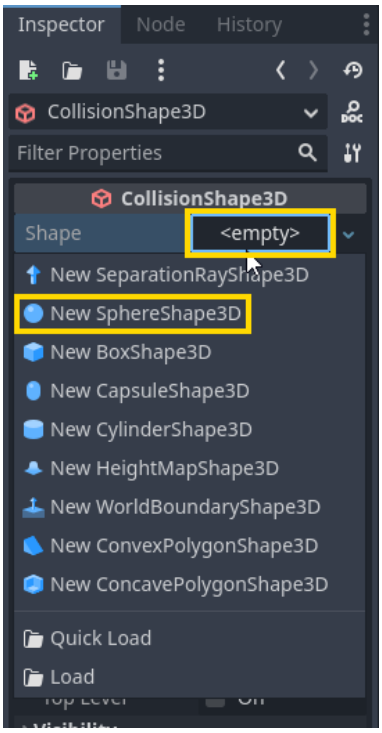

Figure 35: Select New SphereShape3D from the list of shapes

After being assigned a Sphere Shape, click on it to expand it and check different options. Right now the collision is smaller than the model and the radius should be adjusted to be a size that closely matches the size of the T Cell mesh as seen in Figure 36. It is ok to be smaller or bigger, but depending on the case, some objects might create unnatural collisions that break the immersion for the player.

Radius can be adjusted by moving the small red dot on the 3D workspace or in the inspector.

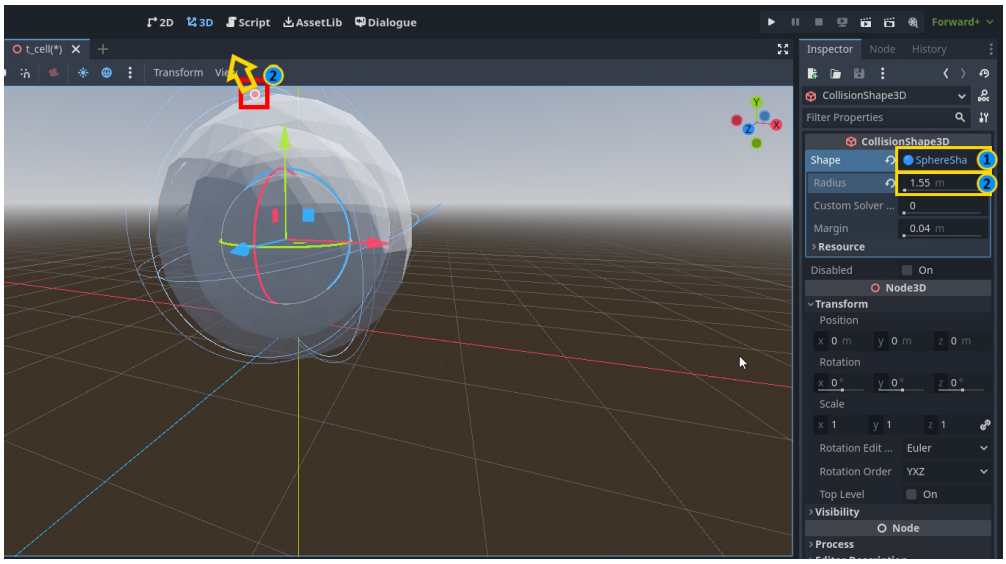

Figure 36: Select New SphereShape3D from the list of shapes

With this final step, you have learnt the elements that you need to import your own model to a Godot game engine. Right now the T Cell Scene is ready to be inserted in the game, so let us go back to our game scene and click on WorldSpawn.

To do so, you can come to the top menu and select the game scene if it is still open (if it appears "game" on the top menu as seen in Figure 37). Or search again for the game scene in the Filesystem and open it, as seen in Figure 37.

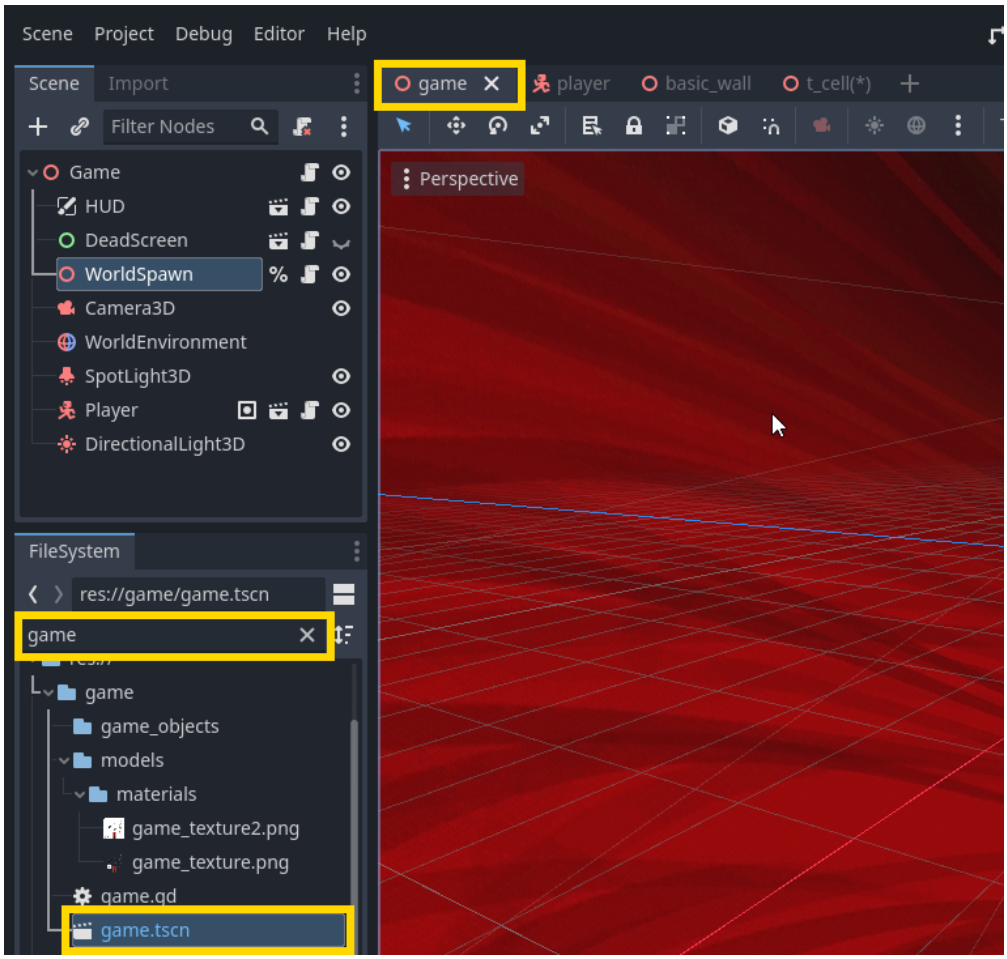

Figure 37: Reopen the game scene

With the T Cell ready to appear in the game, all that is left is to assign T Cell and the Pill to the Packed Scenes that appear in the Inspector such as it appears in Figure 38.

To do so:

- Make sure the Node WorldSpawn appears in the Inspector;
- In the filesystem search for the "t_cell.tscn" and "pills.tscn" scenes;
- Drag each scene to the correspondent in WorldSpawn.

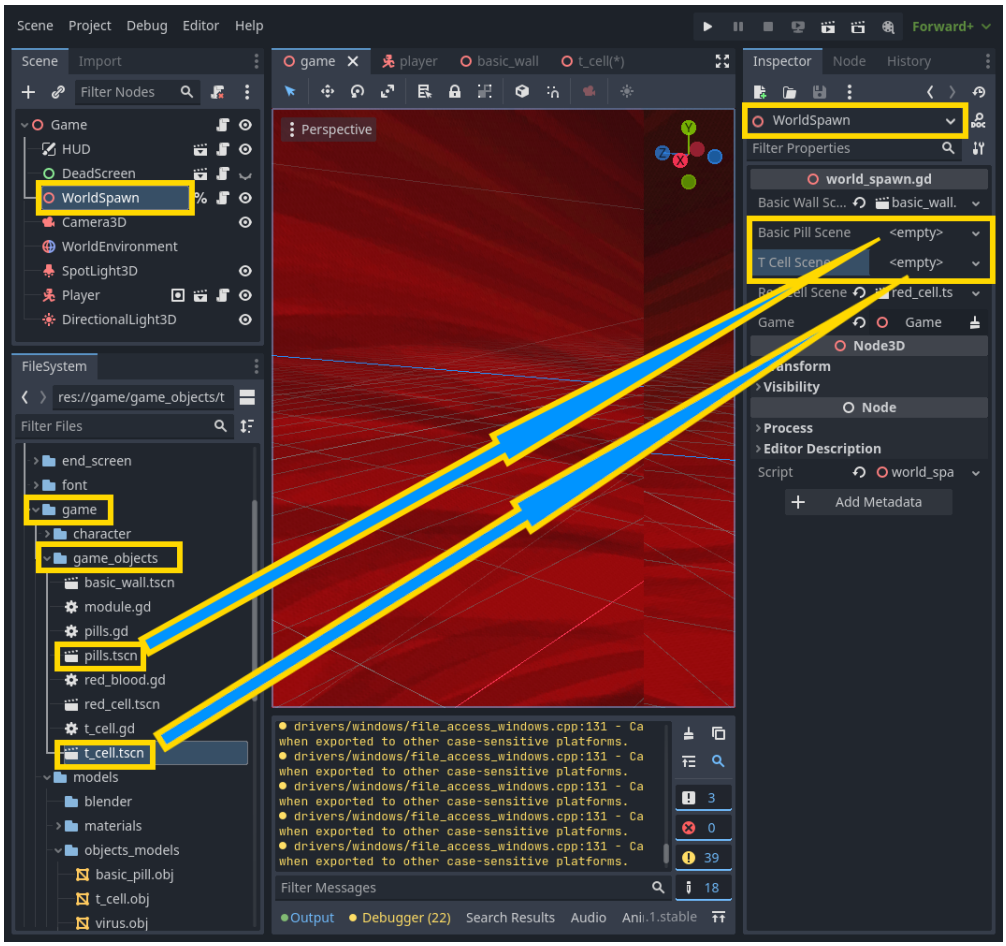

Figure 38: Assign T Cell and Pills to WorldSpawn

Now if you try to run the game... The game elements still don't appear!!! Why!?!?!? Well, it looks like some code is still missing, let us check the "world_spawn.gd" script and search for missing elements.

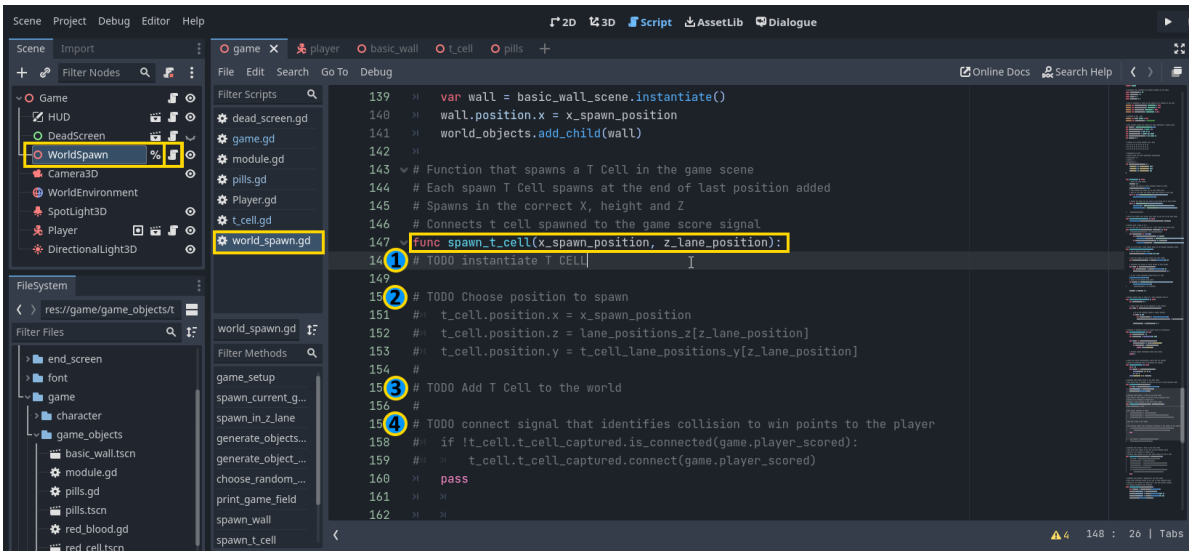

Figure 39: Fix spawn of T Cells in "world_spawn_t_cell.gd"

The first step is to search for the function that spawns the t cells ("spawn_t_cell") and identify what is missing. Looks like our developer left us some TODO's again!! I dunno about you, but it is starting to look to me that it is on purpose to make you learn the important elements. Anyway, looks like some code is missing and other is just commented, so let us check each element of the function:

1. It is required to **instantiate the t-cell object**, but what does this mean? To instantiate means to create a new copy of the scene inside the game world. And because it is a unique copy, it can have its unique position and create various copies.
   To create a new instance of the t_cell object, the code goes "var t_cell = t_cell_scene.instantiate()", where the t_cell_scene was the element defined on top of the script that we assigned previously and the instantiate creates an instance of that scene, saving it in a variable named t_cell.

2. After the object is instantiated, it needs to be put in its **position in the world**, the X, Y and Z are attributed to the object but these positions are not defined inside the function, but they are sent into the function where the Z and Y are pre-defined values in a list. Meaning there is only a specific number of defined positions. In this case, you can just uncomment the code.

3. Next comes the **addition of the object to our world scene**. This is done by using the following code "world_objects.add_child(t_cell)", where world_objects is the node where the objects are spawned and the t_cell is added to it as a child node. This is important to keep the code organization and making the object belong to the main scene. Else it wouldn't appear.

4. This part of the code is responsible for **associating a signal** emitted from the T Cell when it is **captured** so that the game registers as the score to the player on being captured. This part works as the signals mentioned in Level 2 but is only used in code. It is a little more

complex and will not be approached here, but more information can be seen on Godot Signals.

Your code should look like Figure 40. In this code, the "pass" and the "# TODO" can be removed, since they served only guidelines. The pass keyword is used to represent an empty statement. It is often used in the body of a function to indicate that there is no code to be executed.

```
# Function that spawns a T Cell in the game scene
# Each spawn T Cell spawns at the end of last position added
# Spawns in the correct X, height and Z
# Connects t cell spawned to the game score signal
func spawn_t_cell(x_spawn_position, z_lane_position):
# TODO instantiate T CELL
    var t_cell = t_cell_scene.instantiate()

# TODO Choose position to spawn
    t_cell.position.x = x_spawn_position
    t_cell.position.z = lane_positions_z[z_lane_position]
    t_cell.position.y = t_cell_lane_positions_y[z_lane_position]


# TODO Add T Cell to the world
    world_objects.add_child(t_cell)


# TODO connect signal that identifies collision to win points to the player
    if !t_cell.t_cell_captured.is_connected(game.player_scored):
        t_cell.t_cell_captured.connect(game.player_scored)
    pass
```

Figure 40: Final code of function spawn_t_cell

If you run the game now, the T-cells will spawn! At last! The game is starting to work as intended! Looks like you are starting to be ready to go on your own journey as a game developer!

To make the Pills appear, it is just necessary to do the same procedure for the function "spawn_pill". As it can be seen in Figure 42, the function is identical with the same logic that was applied to spawn_t_cell.

Figure 41: Fix spawn of T Cells in "world_spawn_pill.gd"

To verify your code you can use Figure 42.

```
# Function that spawns a Pill in the game scene
# Each spawn Pill spawns at the end of last position added
# Spawns in the correct X, height and Z
# Connects Pill spawned to the game score signal to lose a life
func spawn_pill(x_spawn_position, z_lane_position):
# TODO instantiate T CELL
    var pill = basic_pill_scene.instantiate()

# TODO Choose position to spawn
    pill.position.x = x_spawn_position
    pill.position.z = lane_positions_z[z_lane_position]
    pill.position.y = lane_positions_y[z_lane_position]

# TODO Add Pill to the world
    world_objects.add_child(pill)

# TODO connect signal that identifies collision to win points to the player
    if !pill.pill_contact.is_connected(game.life_lost):
        pill.pill_contact.connect(game.life_lost)
    pass
```

Figure 42: Final code of function "spawn_pill"

In the final version of the code, to keep your code clean and organised, unnecessary comments and code should be removed, such as the "pass" and "TODO" as referred before.

With this, your gameplay loop is almost complete! You already die and restart, but the movement of the player it's still missing! On the next chapter, you're going to learn how to create player input! And with that, a full gameplay will be ready!

2.3 Rewards

Current XP: 70

Congratulations! Another challenge was done and a step closer to the end. You now have learned how to import models to your game and can do it on your own! Now, rest for a while, and enjoy the victory, the last mission is up next, where you are going to learn how to add controls to your player!

2.4 Summary

Table 2: Documentation and Help - Where you can find answers to your questions.

Godot's updated documentation	Available at Godot Docs
Godot Engine Forum	Available at Godot Engine Q&A
Stack Overflow for anything related to programming	Available at Stack Overflow
Godot Signals	Available at Godot Docs Signals
Godot Import 3D Models	Available at Godot Docs Import 3D Models
Godot Physics	Available at Godot Docs Physics Introduction

In this chapter, you discover how to import any 3D model into your game, preparing you to import your own models to any game you wish to develop! Here is a small summary of all the elements you learned in this chapter:

Game Element	**Summary**
Scenes for 3D models	A dedicated scene for each of the 3D model should exist. This scene should be composed by the **physical body** (game examples: **Static Bodies**) and as a child of the physical body a Mesh Instance 3D and a Collision Shape.
Mesh Instance 3D	A mesh instance is a 3D graphical representation of a model, the shape and colours that came from the original 3D model.
Collision Shape	A collision shape is like a virtual boundarie that defines how objects interact in a game. It can be simple (like a sphere) or complex (like a custom mesh). It is used to create a collision between objects (like hitting the floor) or areas.
Instantiate	Create a new copy of an existing scene or other resource with its own unique properties, such as position, rotation and others. Useful to create multiple copies of the same object in a game with variations in properties.
Add Instance to Scene	To bring the instance of the object into the game world, it must be added to the scene. It is also useful to keep the node structure organized.

3 I would like to Move Now!

3.1 Mission description

This chapter will contain information about how to control the player's movement through an input key or keys of your choosing, learn how input mapping works and how a character can jump.

3.2 Goals

The first step to add input control to our playable character is to define what keys should be the controls of the player. For that, let us start by opening Project Settings, as seen in Figure 43.

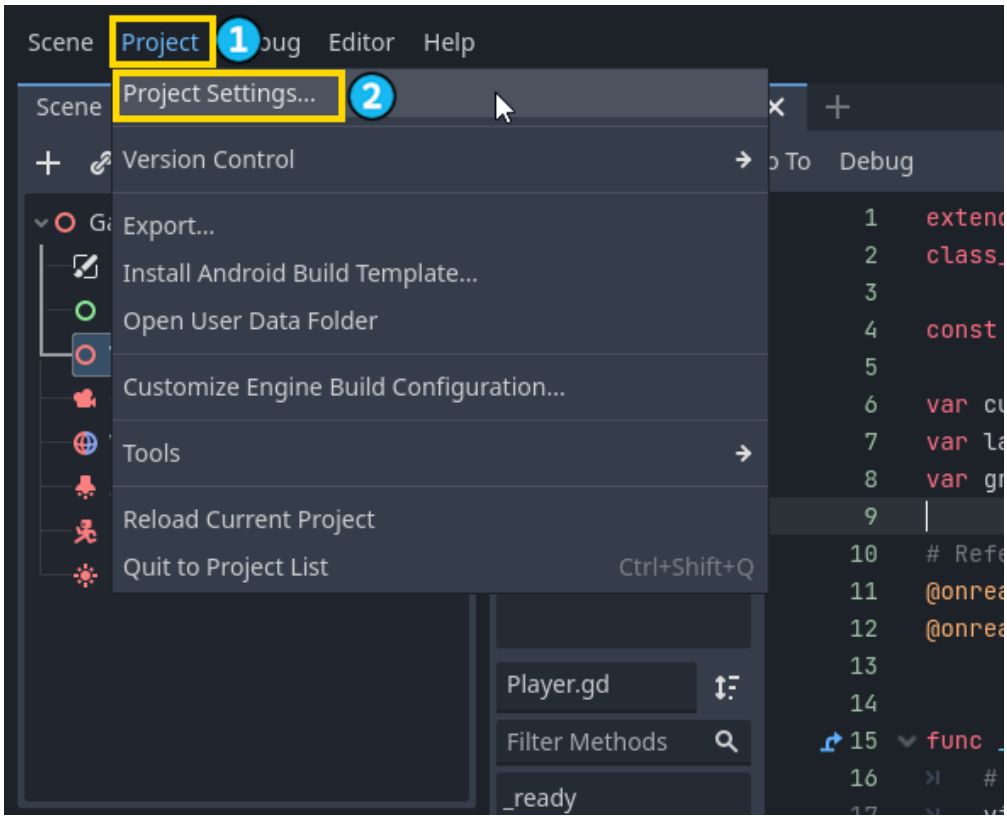

Figure 43: Open project settings

A new window will pop up when selecting Project Settings and by choosing the tab "Input

Map", the available actions already added in the game will appear. Right now, there is only the action to "move_right", with the Right Arrow to dispute this action.

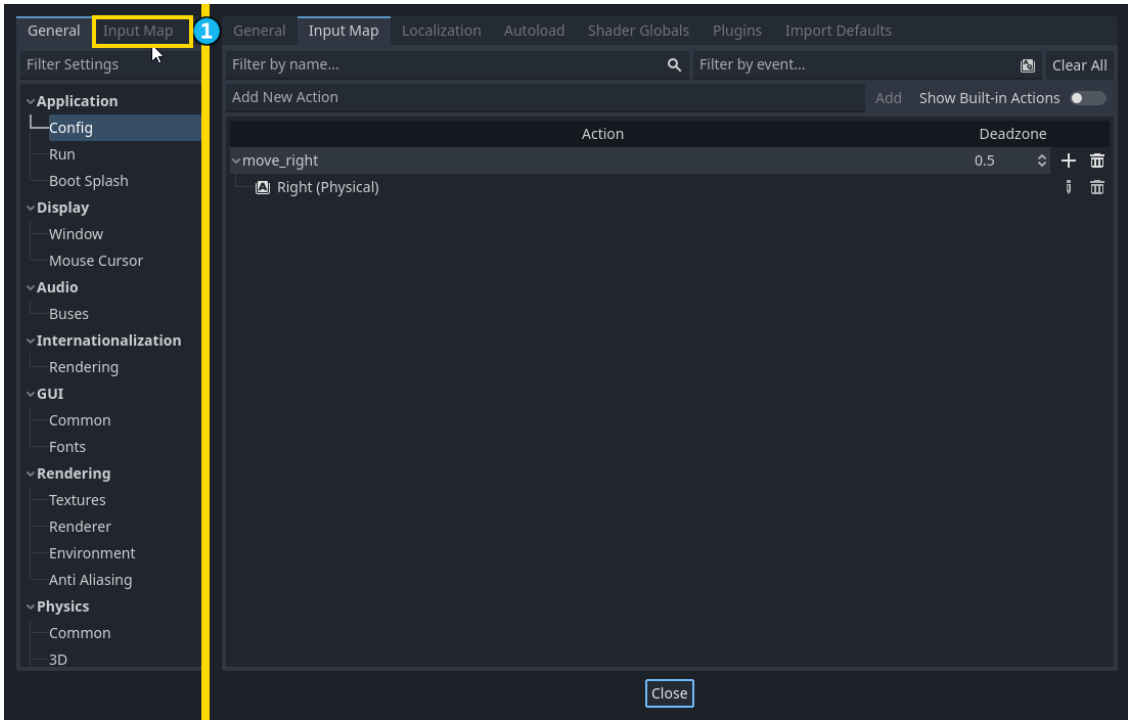

Figure 44: Select Input Map in Project Settings

Let us start by adding more action so that the player can move left too! In Figure 45, there is an "Add New Action", select it and write the name you want to give to the action, it is recommended to keep consistency between names, so that you can remember it and so that other people can understand it by intuition. The recommendation given is "move_left", but feel free to add or change to your name. After writing the name, click the button add as seen in Figure 45, and it will appear in the list of actions.

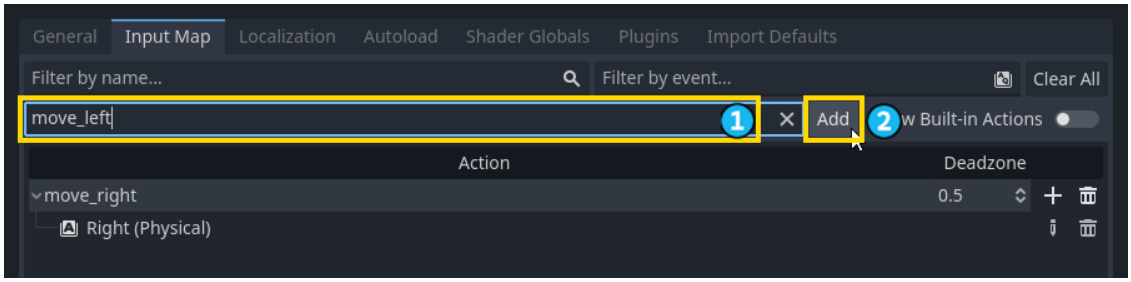

Figure 45: Add "move$_{l}eft$"$action$

To now register the key you intend to use to move left, you click on the "+" button, as seen in Figure 46 to open a new window, where you can select the input.

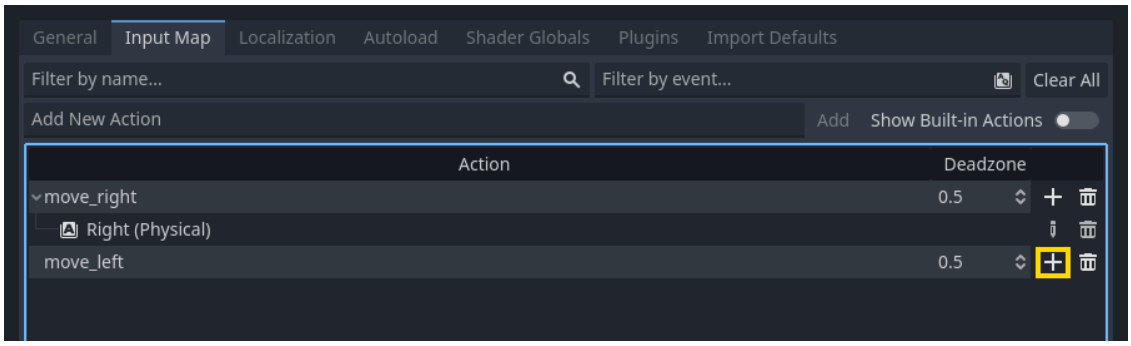

Figure 46: Press the "+" to add the intended key to the action

A new window appears and by clicking the first field with the text "Listening for Input", you may choose the input you wish. In Figure 47, "Left" appears, due to the user clicking the Left Arrow key. After choosing the key, click "ok" to save it.

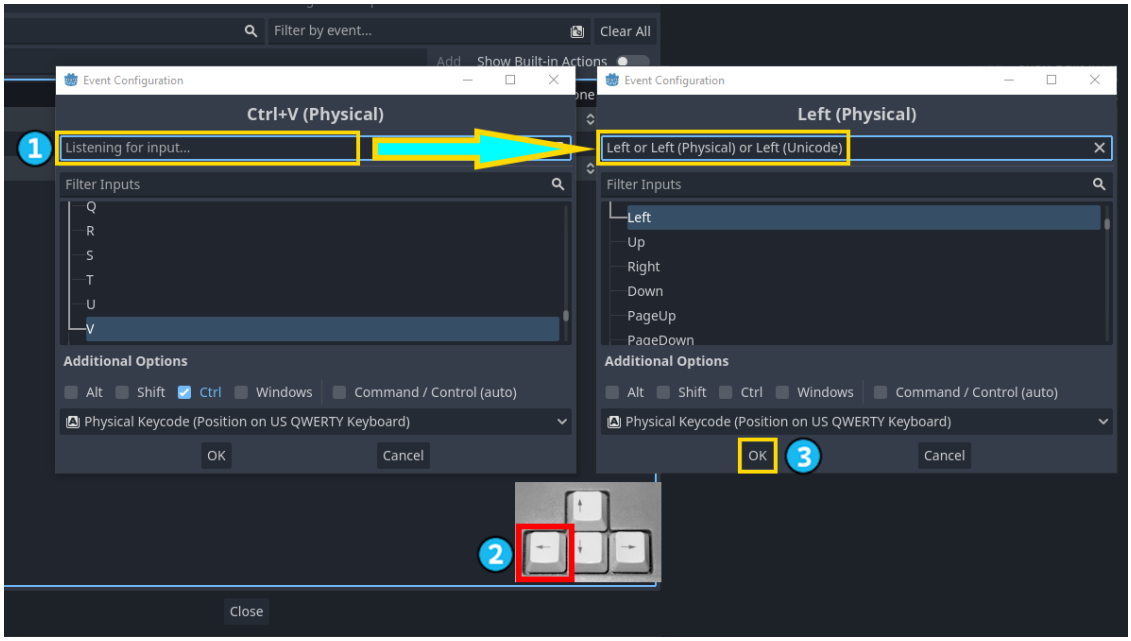

Figure 47: Select the input of your choosing to add to the action

After saving it, the "move_left" action is now associated with clicking the Left Arrow key. Now it is necessary to code the action happening, to what we want to happen in the game. But, before going right to the code, let us add the input for the jump action. For that, repeat the process to "add a new Action", the name should indicate the jump action, in this case, it was named "jump", but you can name it something of your liking. Check Figure 48.

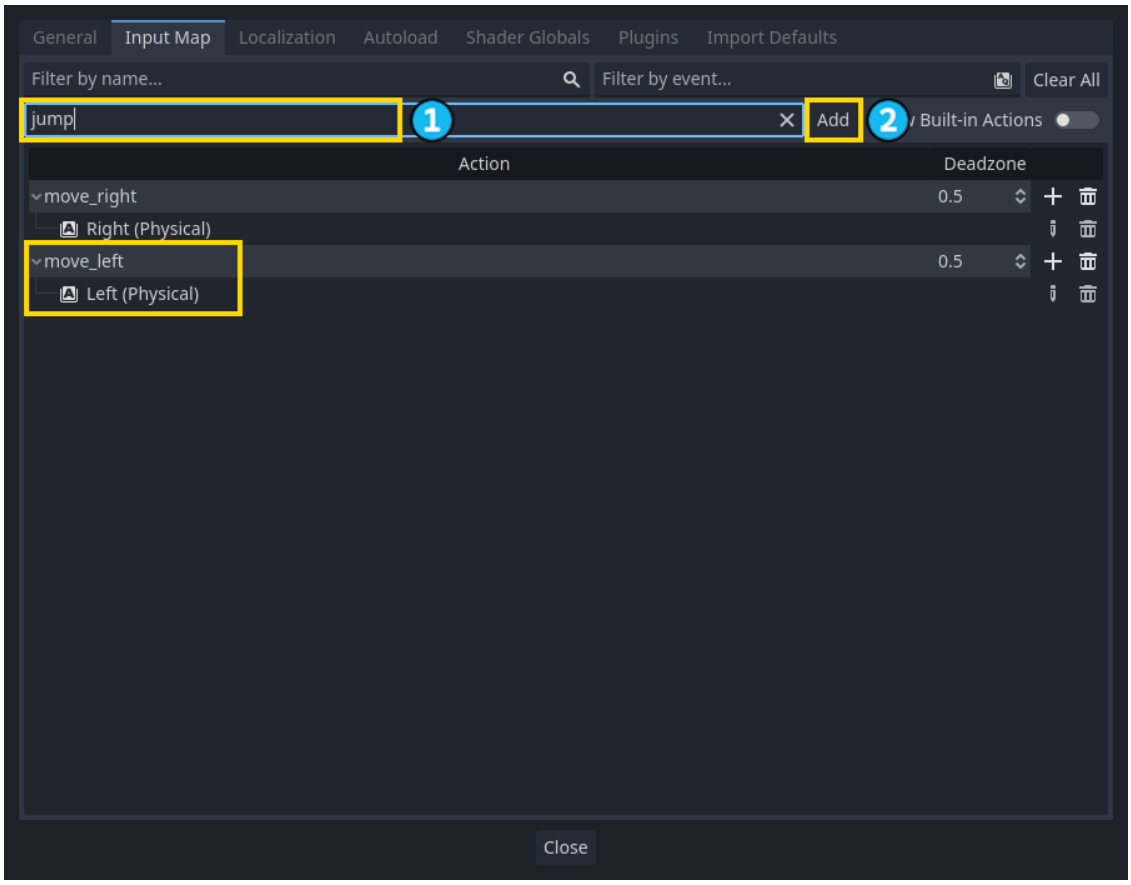

Figure 48: Add jump action to the input map

Choose a key to add to the as jump action, in the case of Figure 49, it was the "Space Key".

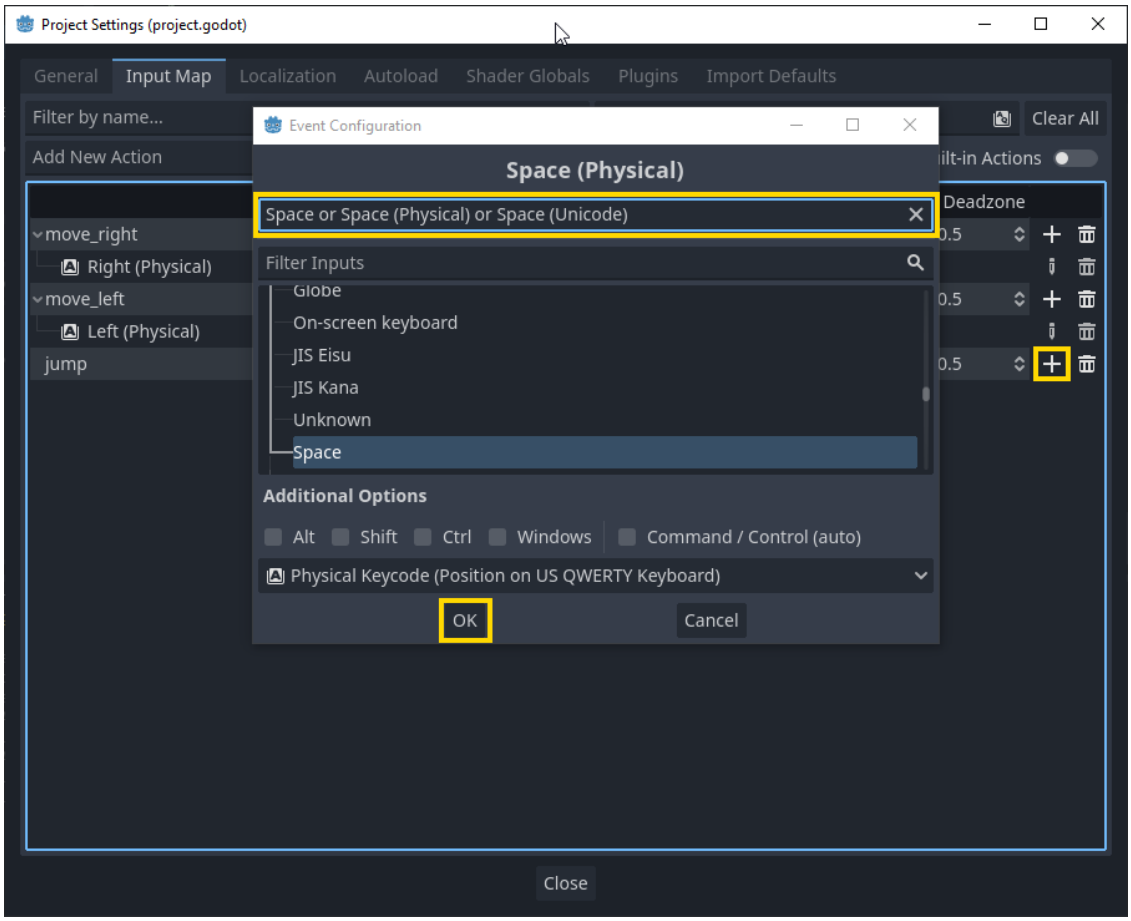

Figure 49: Select space key to add to the action

Now, with all action set, let us open the code where the player is controlled. To do so, open the game scene, and click on the script as seen in Figure 50.

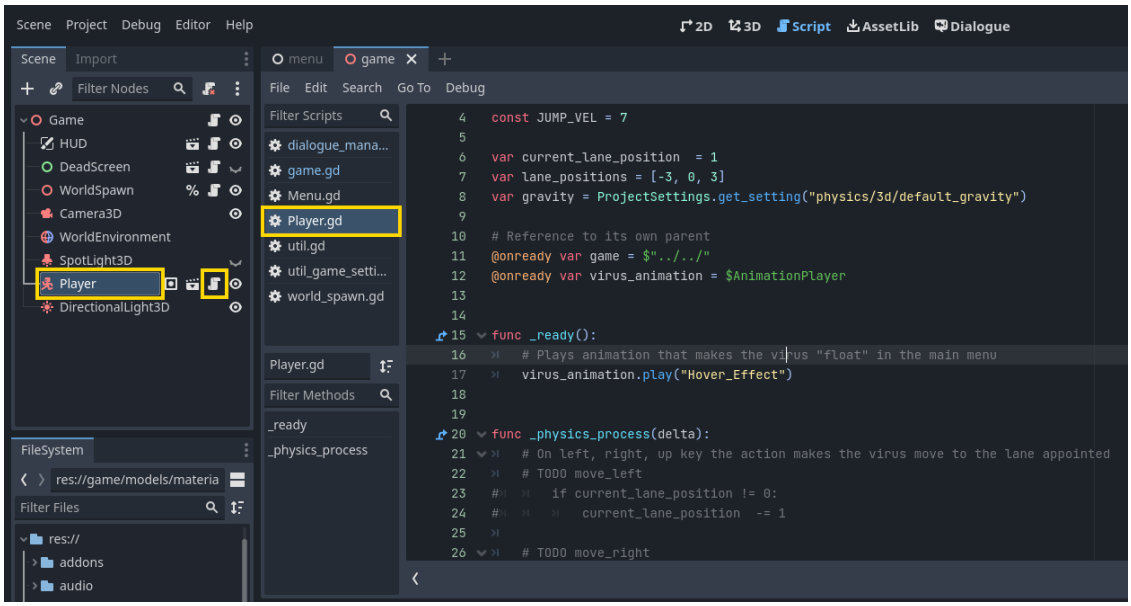

Figure 50: Navigate to the game scene and open the player script

After opening the script, if you explore it a bit, you will find that the physics_process function is all commented, looks like another trap set by the developer for you to fix! In Figure 51, it is visible some "TODO's", corresponding to the move left and move right actions, but what is the missing code?

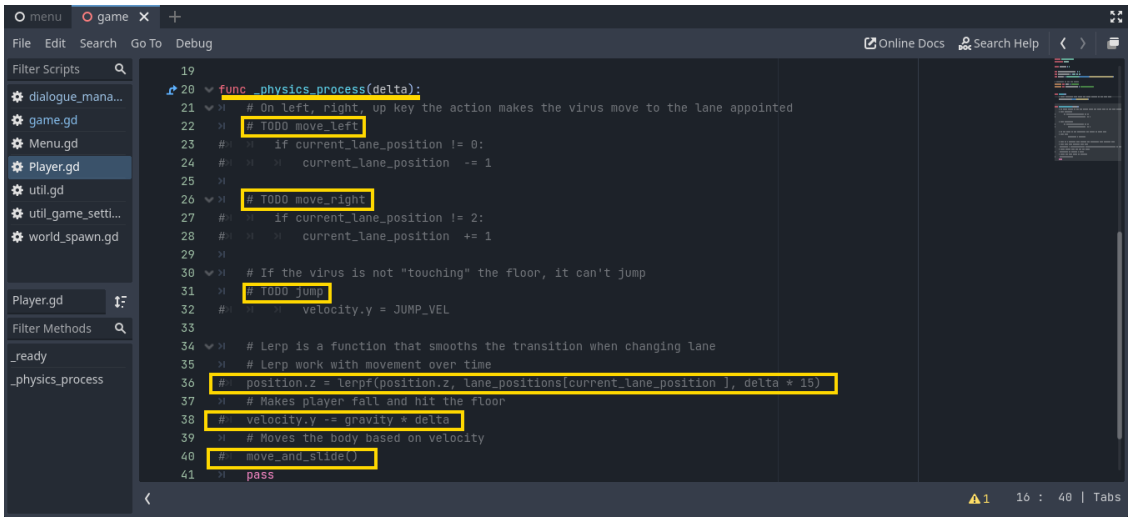

Figure 51: Notice the TODO's left by the developer

There are some different ways to handle the input of the player and you can check more here Godot Input examples, but in this case, let us focus on checking if the player used an Input each _physics_process cycle[3]. So to check if the player clicked a key or a button, we do "Input.is_action_just_pressed()", which checks if the player pressed some key that corresponds to

[3]Each Physics process cycle is the in-game cycle that controls physics, it has 30 updates each 1 second (30 Fps), meaning that it is constantly checking if there was an Input from the player

an action, but there is the name of the action missing. So, to check if the player did an action like "move_left", we can do "if Input.is_action_just_pressed("move_left"):", meaning that each time a key is pressed that represents move left, the code will run.

The code below needs to be uncommented so that it can check the current position of the virus in the lane. If the virus is currently at the far left (value 0) it won't move further, but if it isn't, it will move one value to the left.

The move_right action has the same logic: "if Input.is_action_just_pressed("move_right"):" the code will check if the virus is positioned at the far right (value 2) and it will only move to the right if it isn't.

Finally, the "jump" action is verified by "if Input.is_action_just_pressed("jump"):". When it jumps, the code gives the velocity.y (where y represents the height axis) some value, so it goes up. Each updates the velocity.y, goes done due to the gravity math being applied below "velocity.y -= gravity x delta", where the delta represents the in-game time passed.

The Lerp code is just a mathematical equation, that creates a smooth transition between two places, instead of instantly changing the position, so that the player virus moves.

The final _physics_process function should look like this:

```
func _physics_process(delta):
    # On left, right, up key the action makes the virus move to the lane appointed
    if Input.is_action_just_pressed("move_left"):
        if current_lane_position  != 0:
            current_lane_position  -= 1
    if Input.is_action_just_pressed("move_right"):
        if current_lane_position  != 2:
            current_lane_position  += 1
    # If the virus is not "touching" the floor, it can't jump
    if Input.is_action_just_pressed("jump") && is_on_floor():
            velocity.y = JUMP_VEL

    # Lerp is a function that smooths the transition when changing lane
    # Lerp work with movement over time
    position.z = lerpf(position.z, lane_positions[current_lane_position ], delta * 15)
    velocity.y -= gravity * delta
    # Moves the body based on velocity
    move_and_slide()
```

Figure 52: Final code result of "func _physics_process():"

Notice that in the code, there is a " is_on_floor()", this code makes sure the virus is on the floor before you can jump again.

3.3 Rewards

Current XP: 100

Congratulations! You have reached the end! Your game is now fully running and you are ready to continue your game development journey alone. Some challenges are going to be rough, but persistence is your friend and if you insist, nothing will stop you the same way you did all the workshops! The Playmutation team wishes you the best of luck on your journey!

3.4 Summary

Table 3: Documentation and Help - Where you can find answers to your questions.

Godot's updated documentation	Available at Godot Docs
Godot Engine Forum	Available at Godot Engine Q&A
Stack Overflow for anything related to programming	Available at Stack Overflow
Godot Input Examples	Available at Godot Docs Input Examples
Godot Input Map	Available at Godot Docs Input Map

In this chapter, you have learned how to create your game's input to control the player, here is a summary of the components taught:

Game Element	**Summary**
Input Map	System that allows to map input events to actions in the game. Input events can be anything from keyboard presses to mouse clicks, and actions can be anything from moving a character to firing a weapon.
Detect Input from player	Verifying in the physics process function the type of input that can control the output of that action. Code: "if Input.is_action_just_pressed("action_name"):"
Is On Floor	It is a Godot built-in function that checks whether a physics-based object is currently in contact with a surface that is considered to be the "floor".

Well played!

You have reached final level!

Congratulations!!!