Introduction to Game Development in Unity

Iftekharul Islam

Outline

- **Introduction to Unity**
- **Setting up the Game**
- **Moving the Player**
- **Moving the Camera**
- **Detecting Collisions**
- **Creating Obstacles**
- **Adding Win/Lose Condition**
- **Display Text**
- **Building the Game**

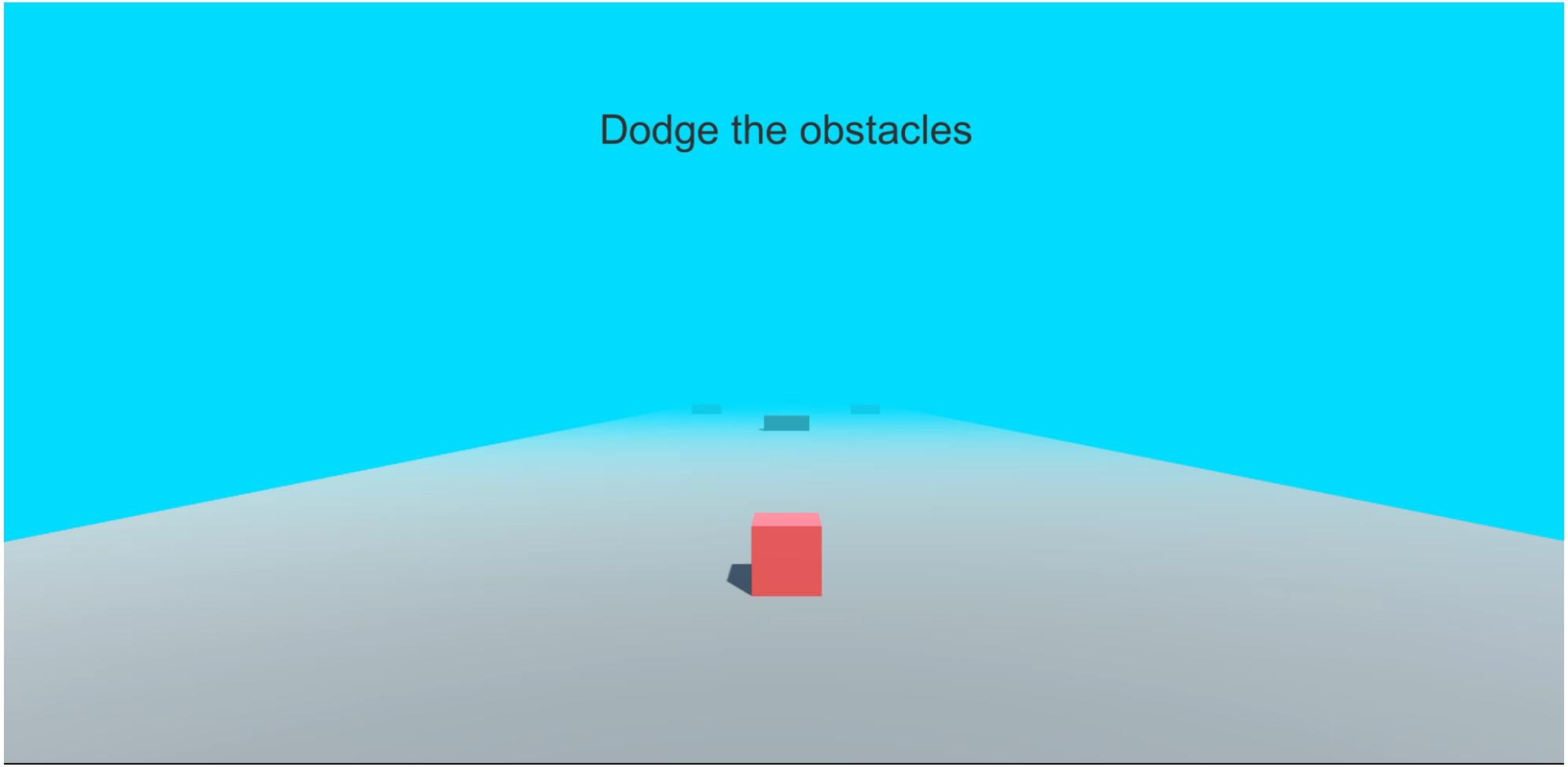
Section

Introduction to Unity

Unity

- Unity is a cross-platform game development system.
- Unity features a complete toolkit for designing and building games.
- In this lecture, we will present the basic elements of Unity.
- A good starting point to learn Unity: https://learn.unity.com/

The Game



Download and Installation

- Download: https://unity.com/download
- Installation tutorial: https://www.youtube.com/watch?v=ewiw2tcfen8
- We will be using **Unity 2022.3.19f1** for the lecture.

Section

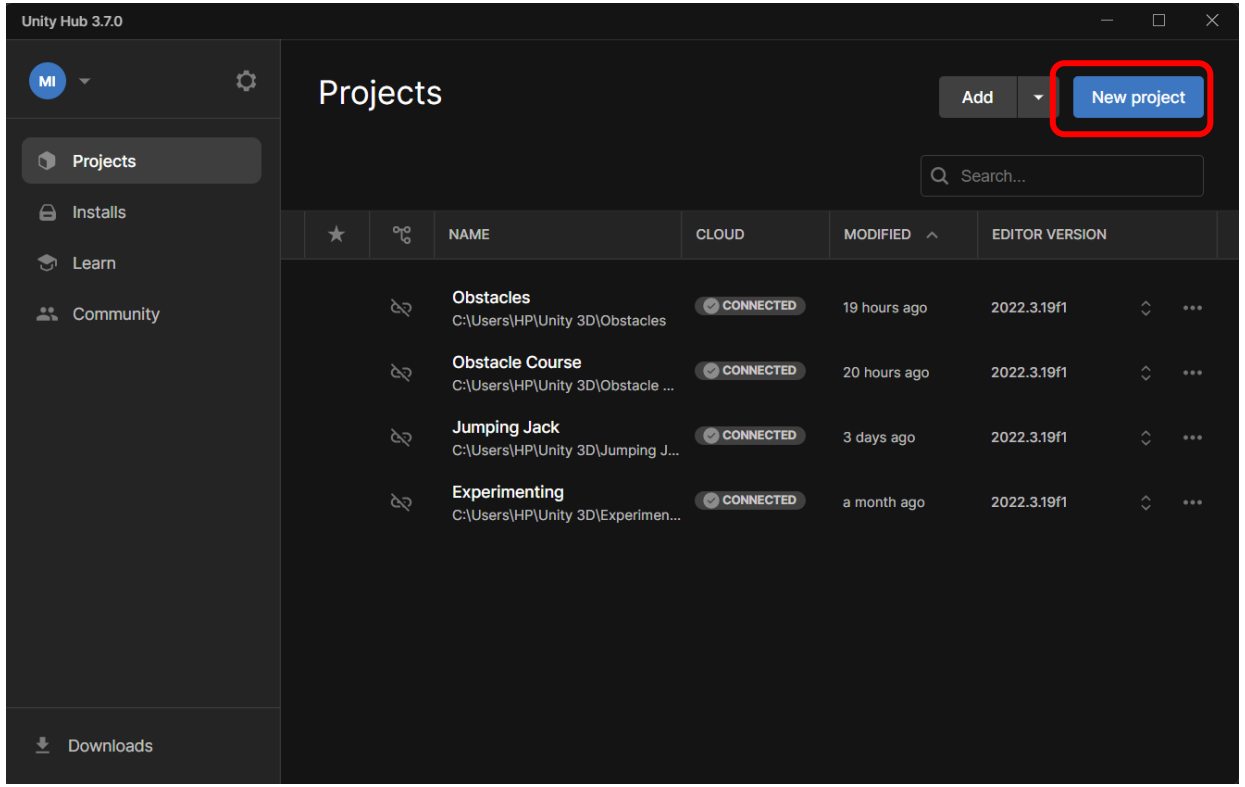
Setting up the Game

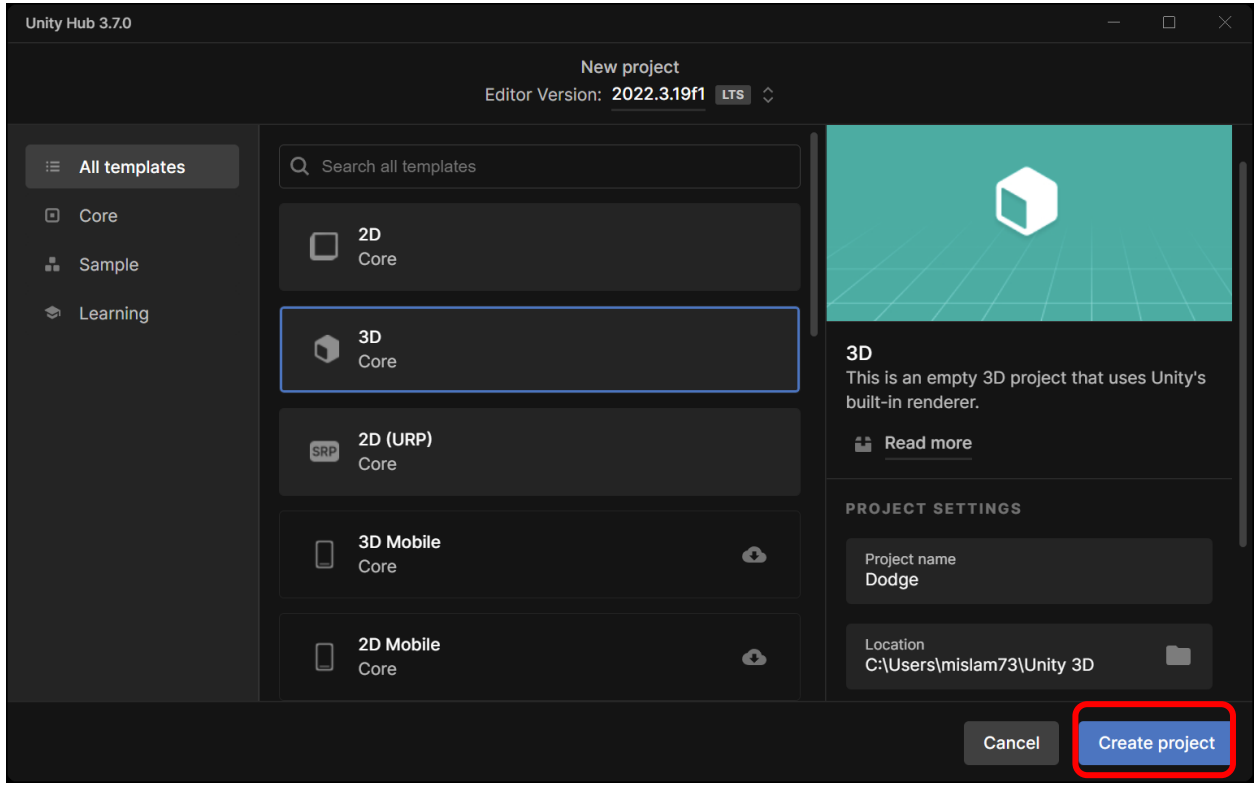
Create a new Unity project

- **Select a project template.**
  - Open Unity Hub and log in using your Unity account.
  - Select **New Project**.
  - Select the **3D** template and then select **Download template** if it is not already downloaded**.**

- **Create the new project.**
  - In the Project Settings, enter “Dodge” in the **Project Name** box, select the **Location** box, and select a local folder of your choice.
  - Select **Create Project** and wait for the Unity Editor to open.

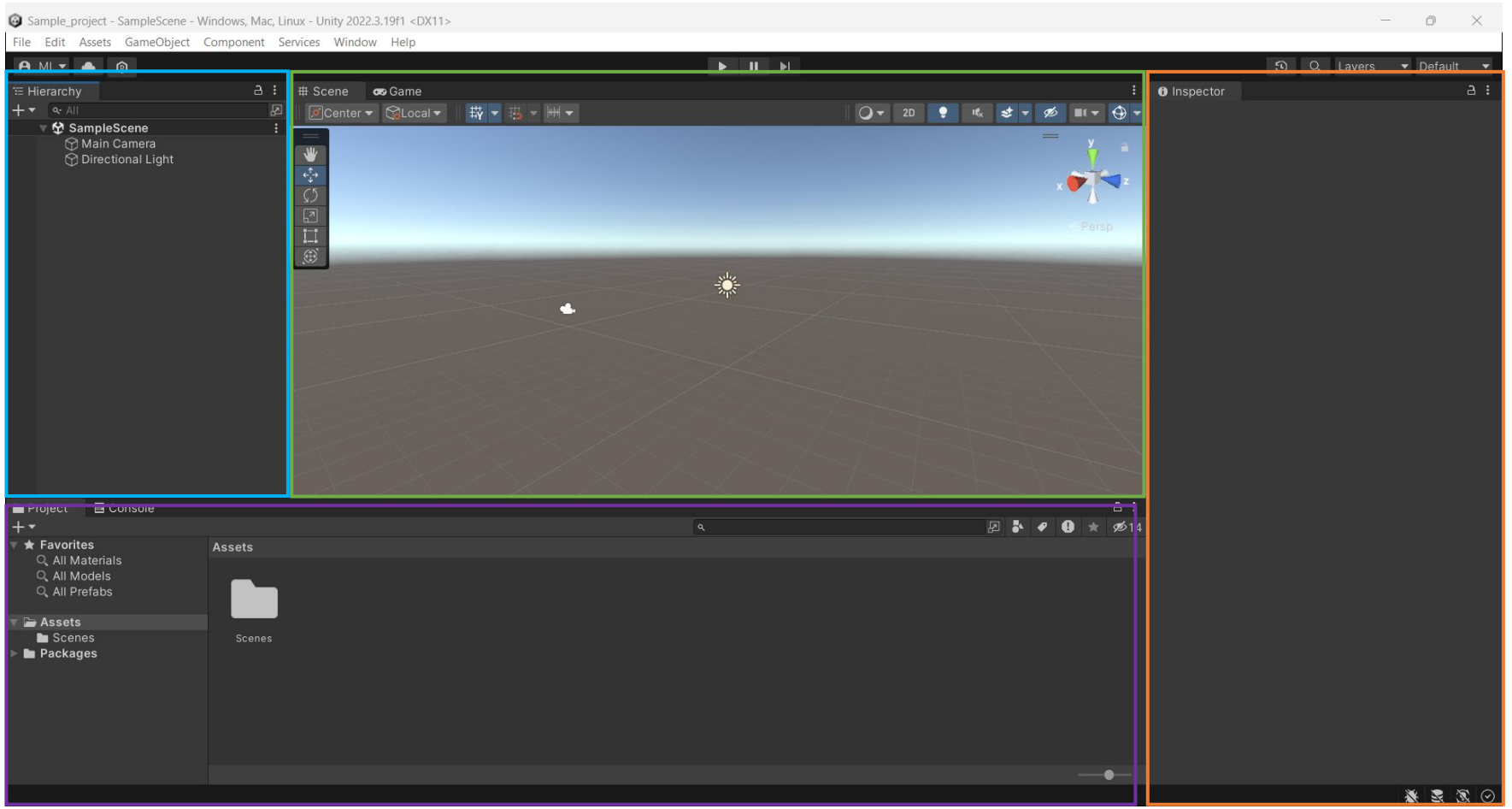
Create a new Unity project





Overview of the Unity IDE

- **Scene/game view**
  - Build/play scene
- **Hierarchy**
  - Manage game objects
- **Inspector**
  - Manage components
- **Project window**
  - Manage assets

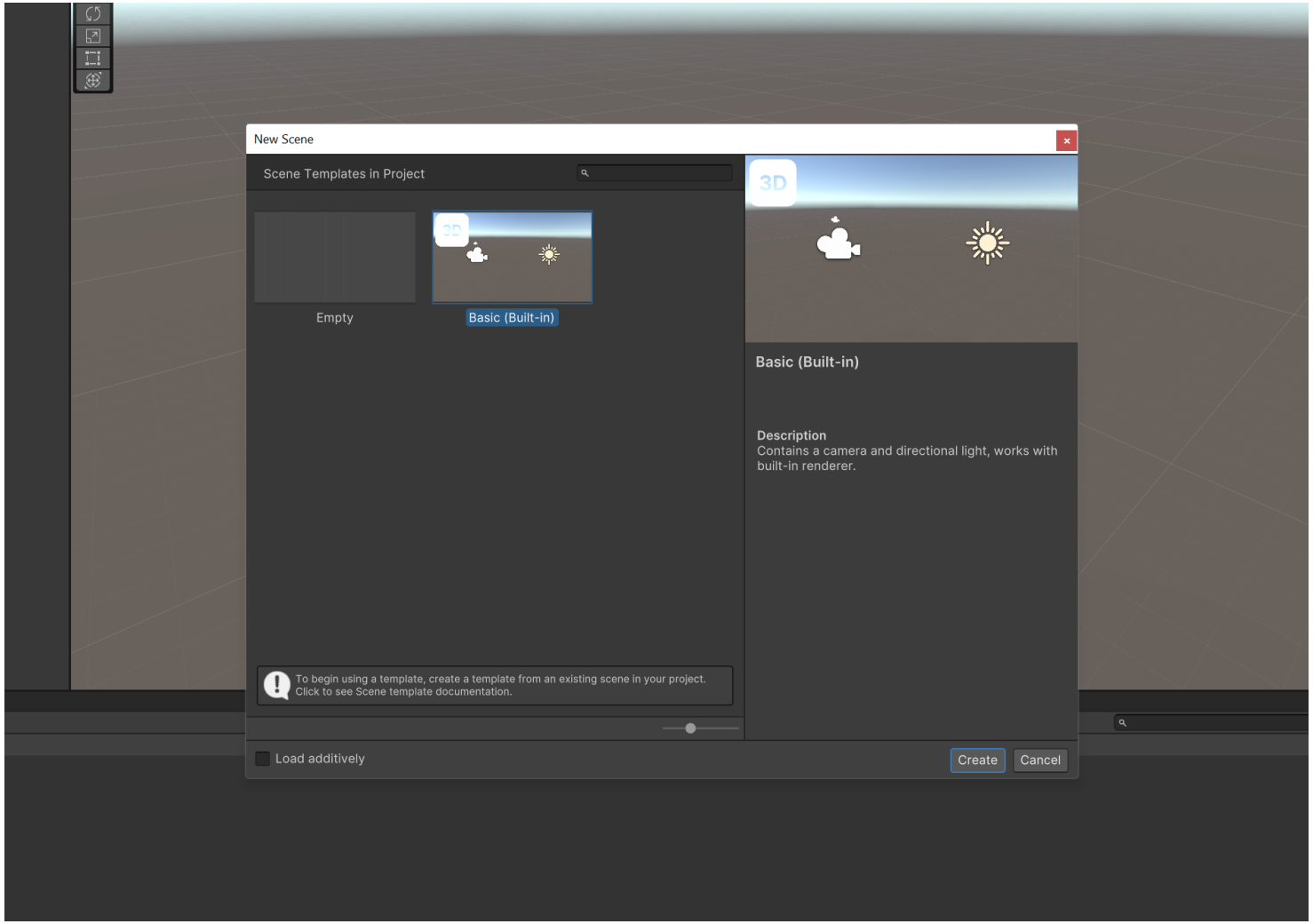


Create a new Scene

- **Set up your workspace.**
  - Make sure the **Default** Layout is selected from the **Layout** dropdown for tutorial consistency.

- **Create a new scene from a template.**
  - To create a new scene, select **File** > **New Scene**.
  - Select the **Basic (Built-in)** template, then select **Create**.

Create a new Scene



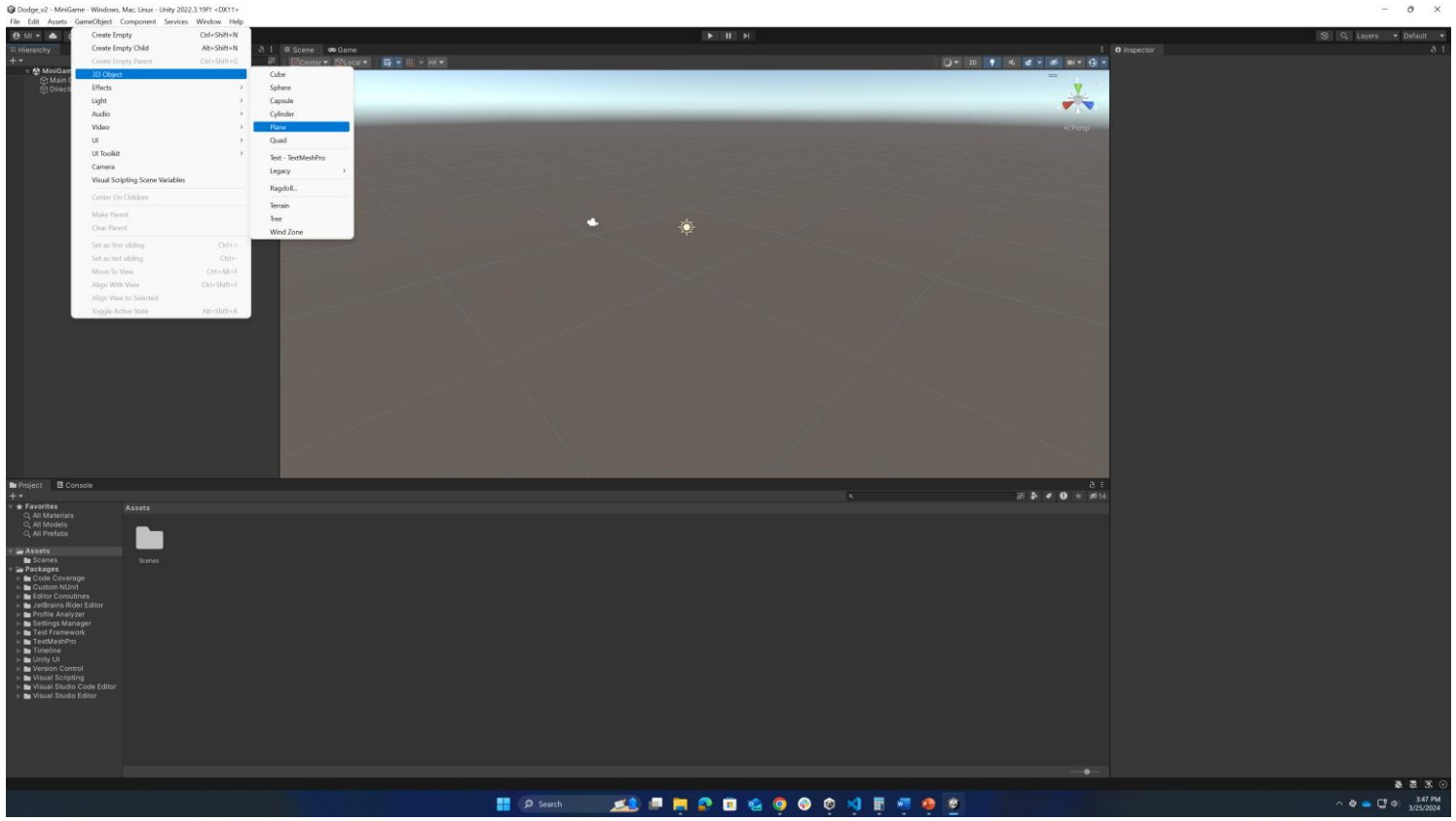
Create a new Scene

- **Save the scene.**
  - Select **File** > **Save As**.
  - Name the scene "MiniGame".
  - Save the scene in a new folder named “Scenes”.

Create a primitive plane

- **Create a Plane GameObject.**

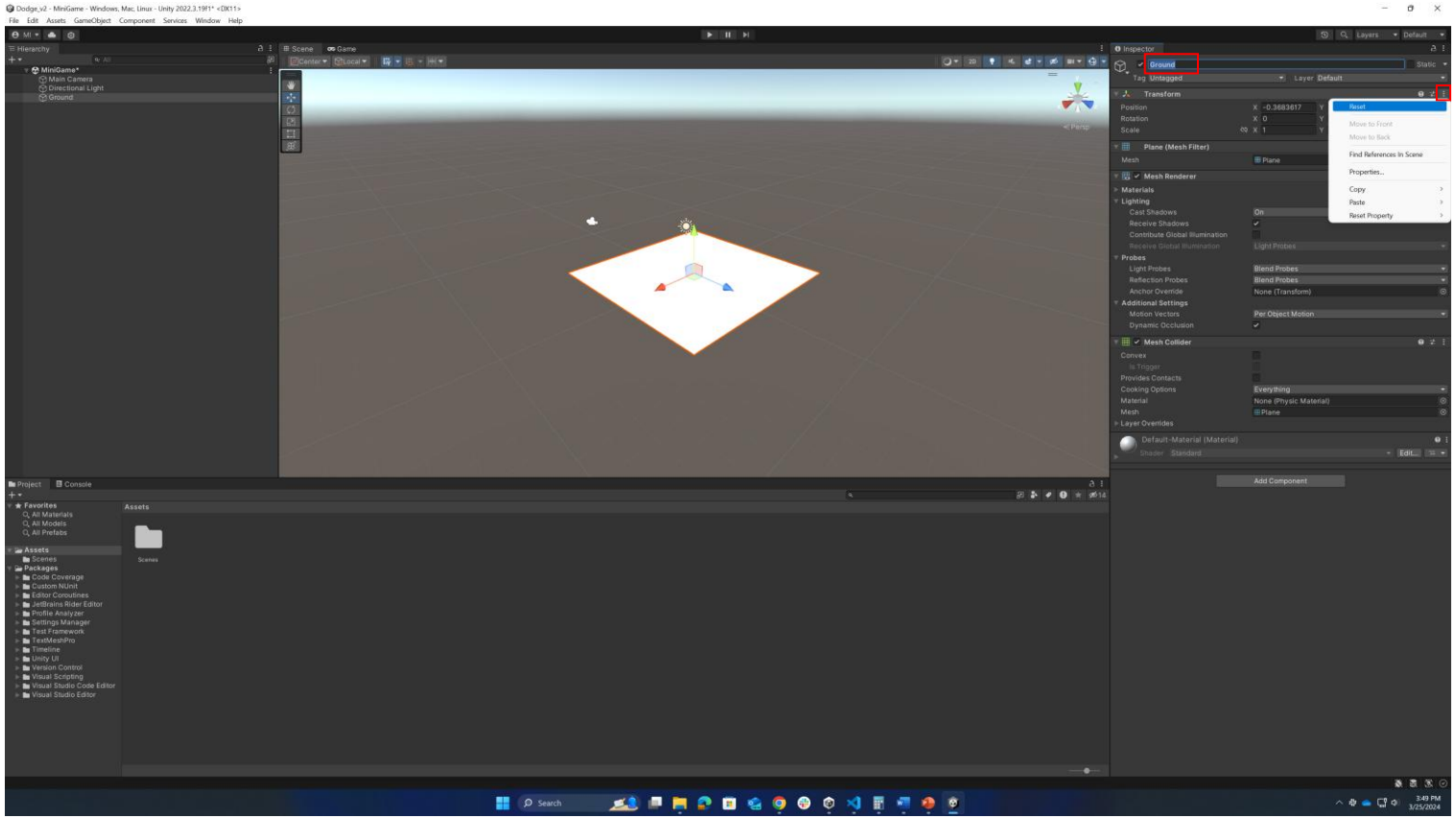
  - From the main menu, select **GameObject** > **3D Object** > **Plane**.
  - Alternatively, in the **Hierarchy** window, select the **Add** menu (**+**) > **3D Object** > **Plane**.
  - At the top of the **Inspector** window, rename the newly created **Plane** GameObject "Ground".



Create a primitive plane

- **Reset the position of the Plane.**

  - With the **Ground** GameObject selected, in the upper-right corner of the **Transform** component, select the vertical **More** (⋮) menu .
  - Select **Reset** for the **Transform** component of the **Ground** GameObject. This action places the GameObject at the origin point (0, 0, 0) in the scene.



Create a primitive plane

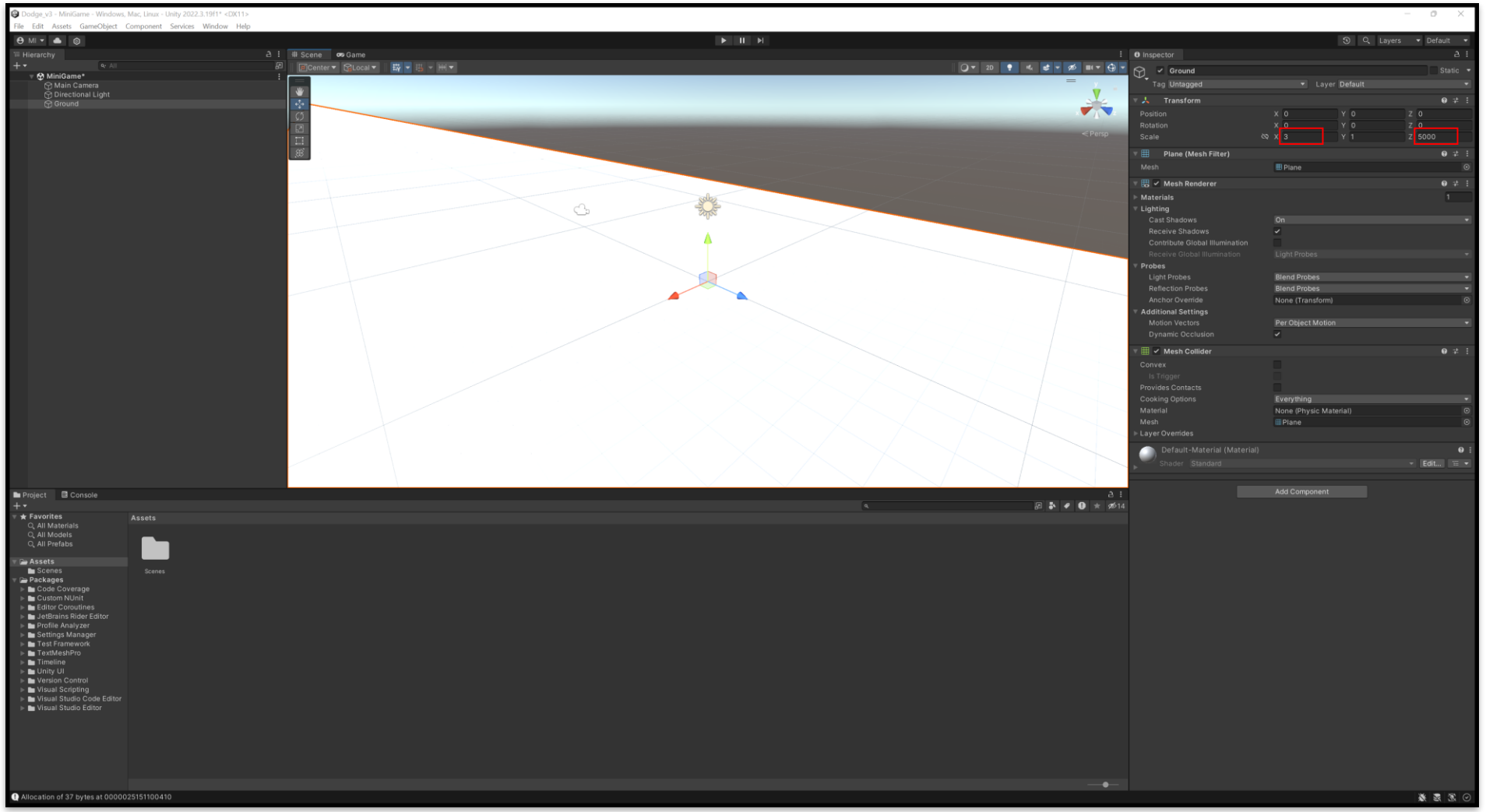
- **Frame the Plane in the Scene view.**
  - Ensure the **Ground** GameObject is selected and position the cursor in the **Scene** view.
  - Press the **F** key to frame the entire GameObject nicely within the **Scene** view.
  - Alternatively, select **Edit** > **Frame Selected** from the main menu.

Scale the Ground Plane

- **Increase the scale of the Ground Plane.**
  - With the **Ground** GameObject selected, activate the **Scale** tool by pressing the **R** hotkey.
  - We can drag the **X** (red) and **Z** (blue) handles to increase the size of the Plane.

- **Set precise scale values**
  - In the **Transform** component, set the **X** and **Z Scale** values to **3** and **5000** respectively.

Scale the Ground Plane



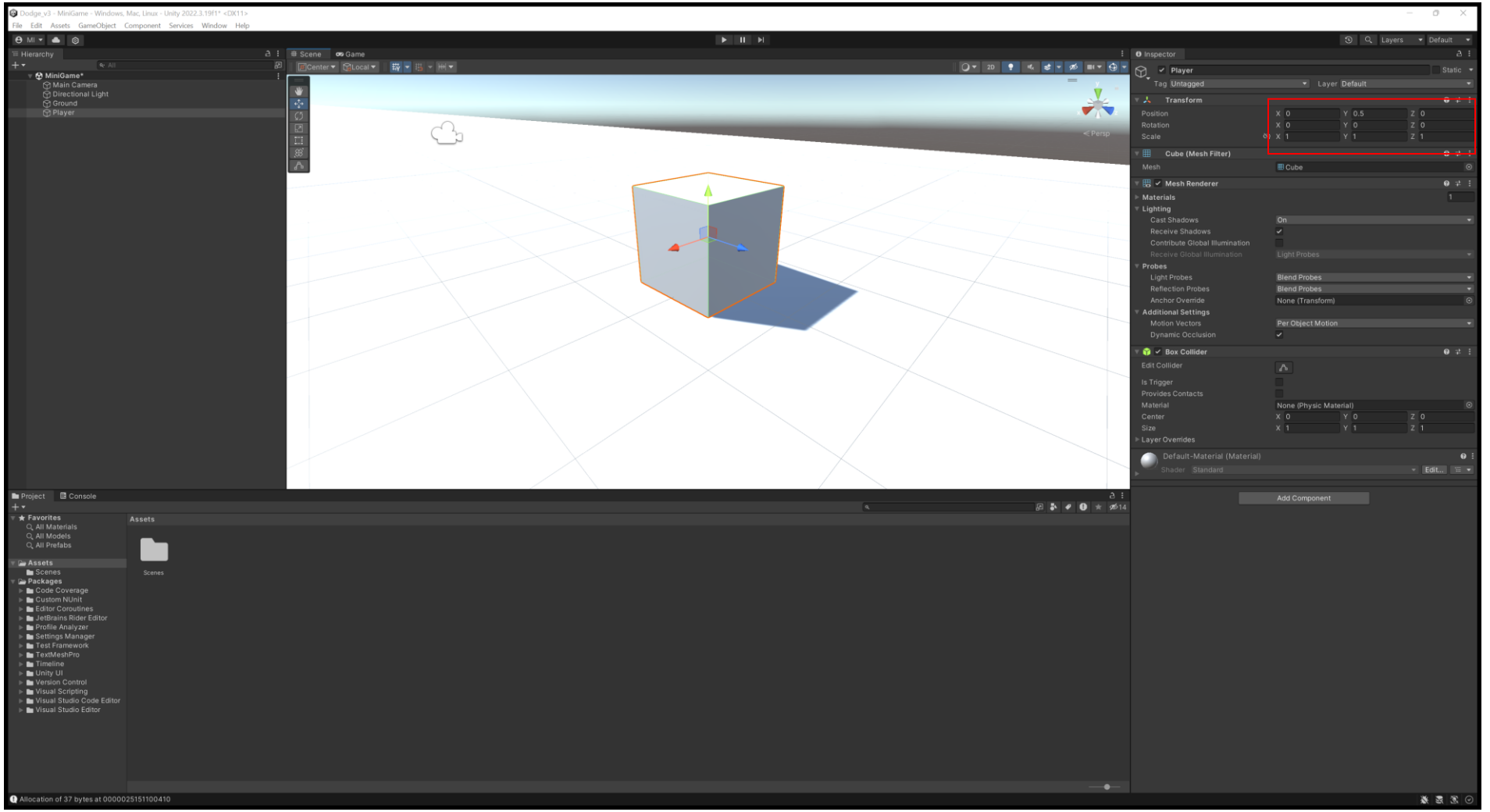
Create a Player GameObject

- **Create a Player cube.**
  - In the **Hierarchy** window, **right-click** > **3D Object** > **Cube.**
  - Rename the newly created **Cube** GameObject "Player".

- **Position the Player Sphere at the origin.**
  - In the **Inspector** window, reset the **Transform** component of the **Player** GameObject to position it at the origin point (0, 0, 0) of the scene.
  - Press the **F** key in the **Scene** view to frame the **Player** GameObject in the **Scene** view.

- **Move the Player GameObject up to sit on the Plane.**
  - In the **Transform** component for the **Player** GameObject, set the **Y Position** value to **0.5**.

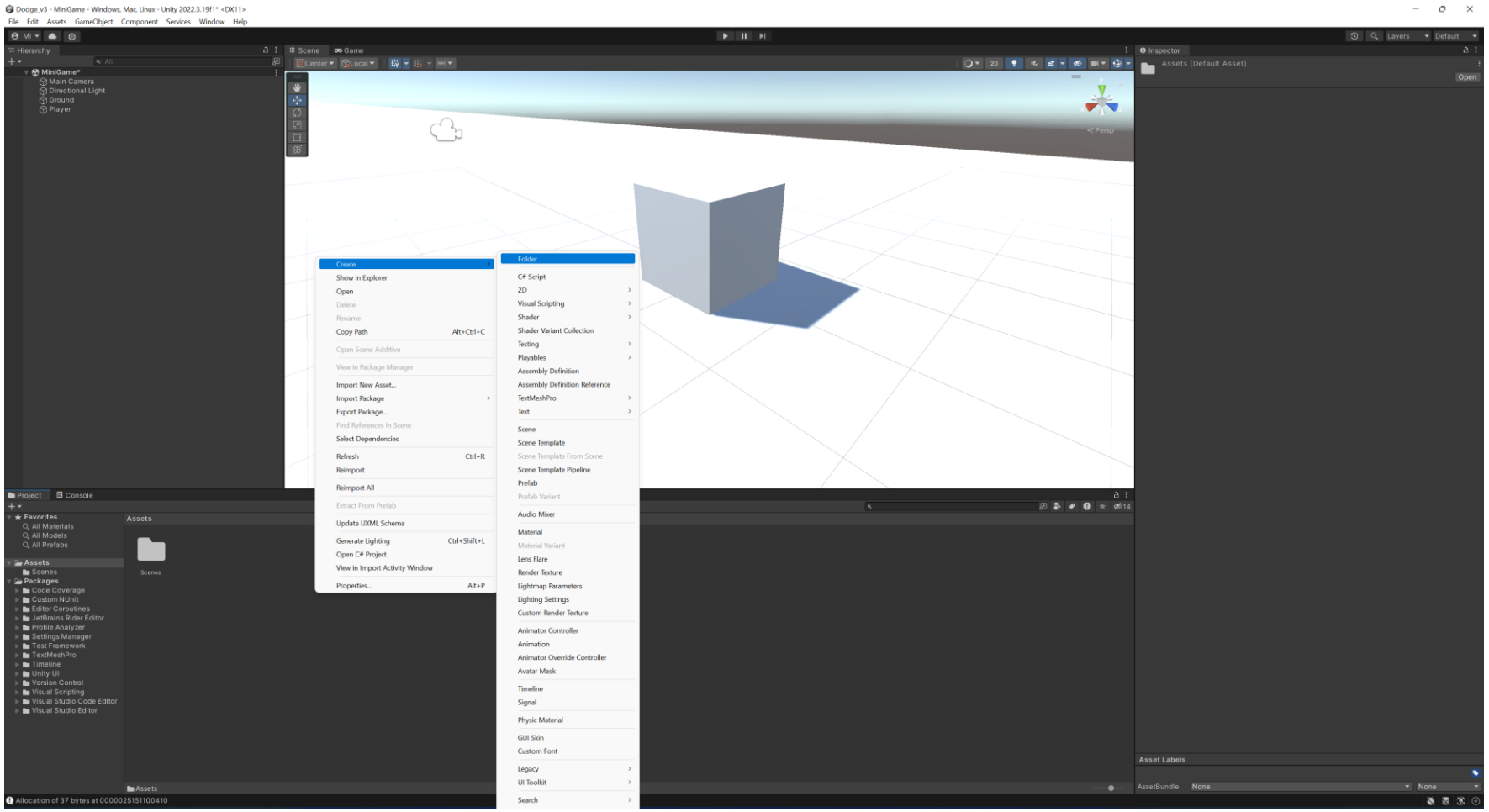
Create a Player GameObject



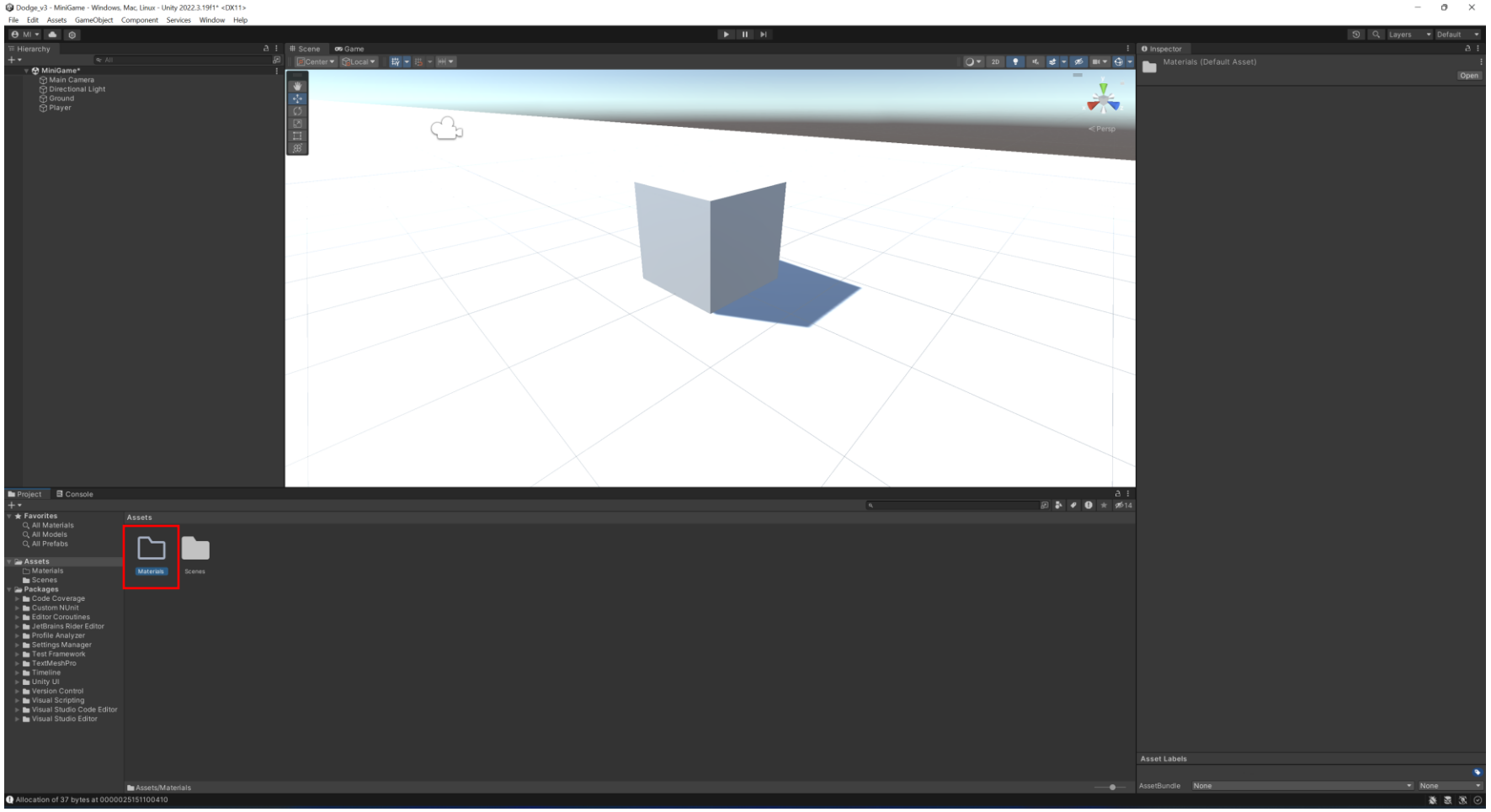
Add colors with Materials

- **Create a new Materials folder.**
  - In the **Project** window, **right-click** > **Create** > **Folder** to make a new folder.
  - Rename the new folder "Materials".

Add colors with Materials



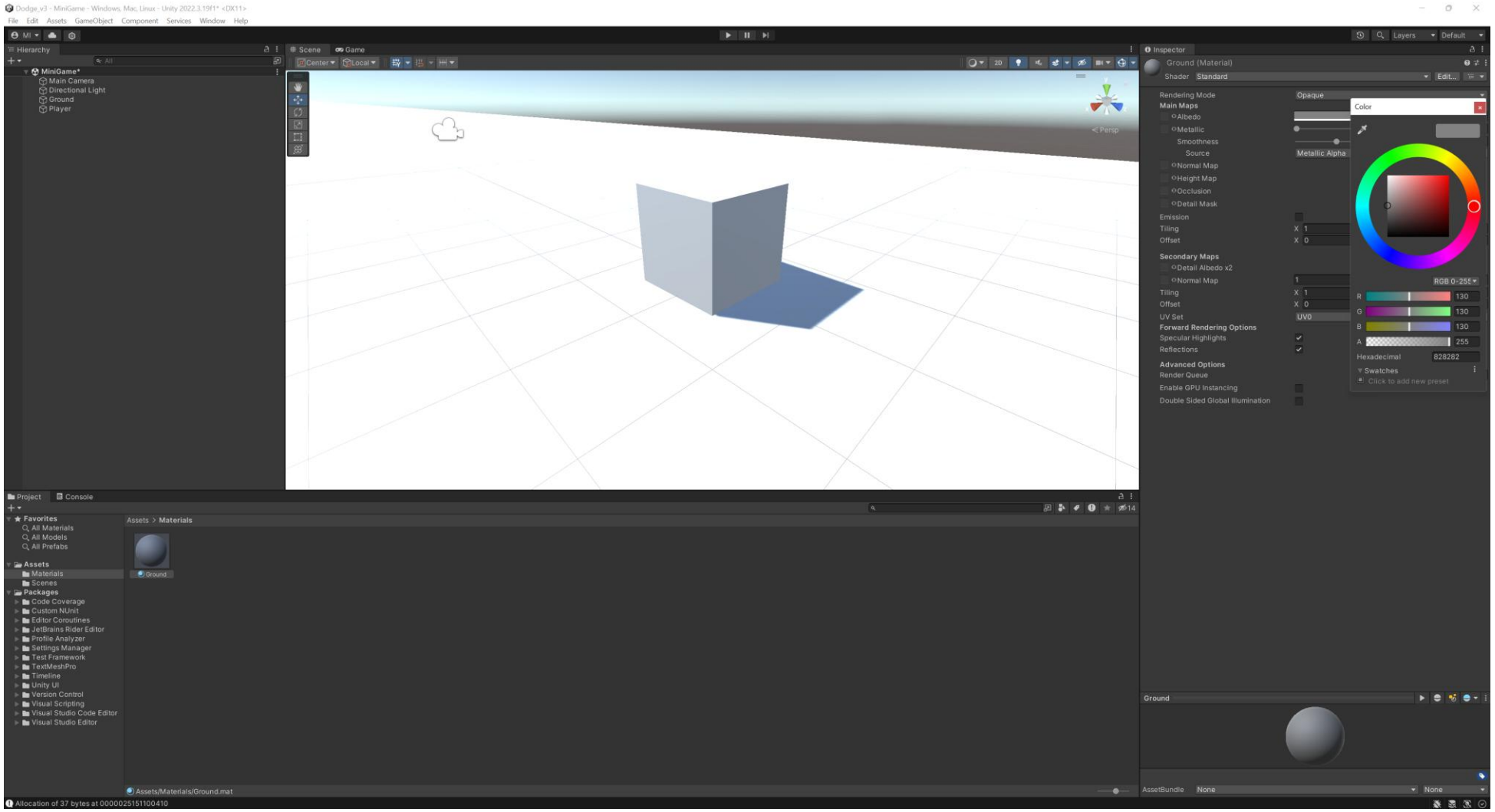
Add colors with Materials



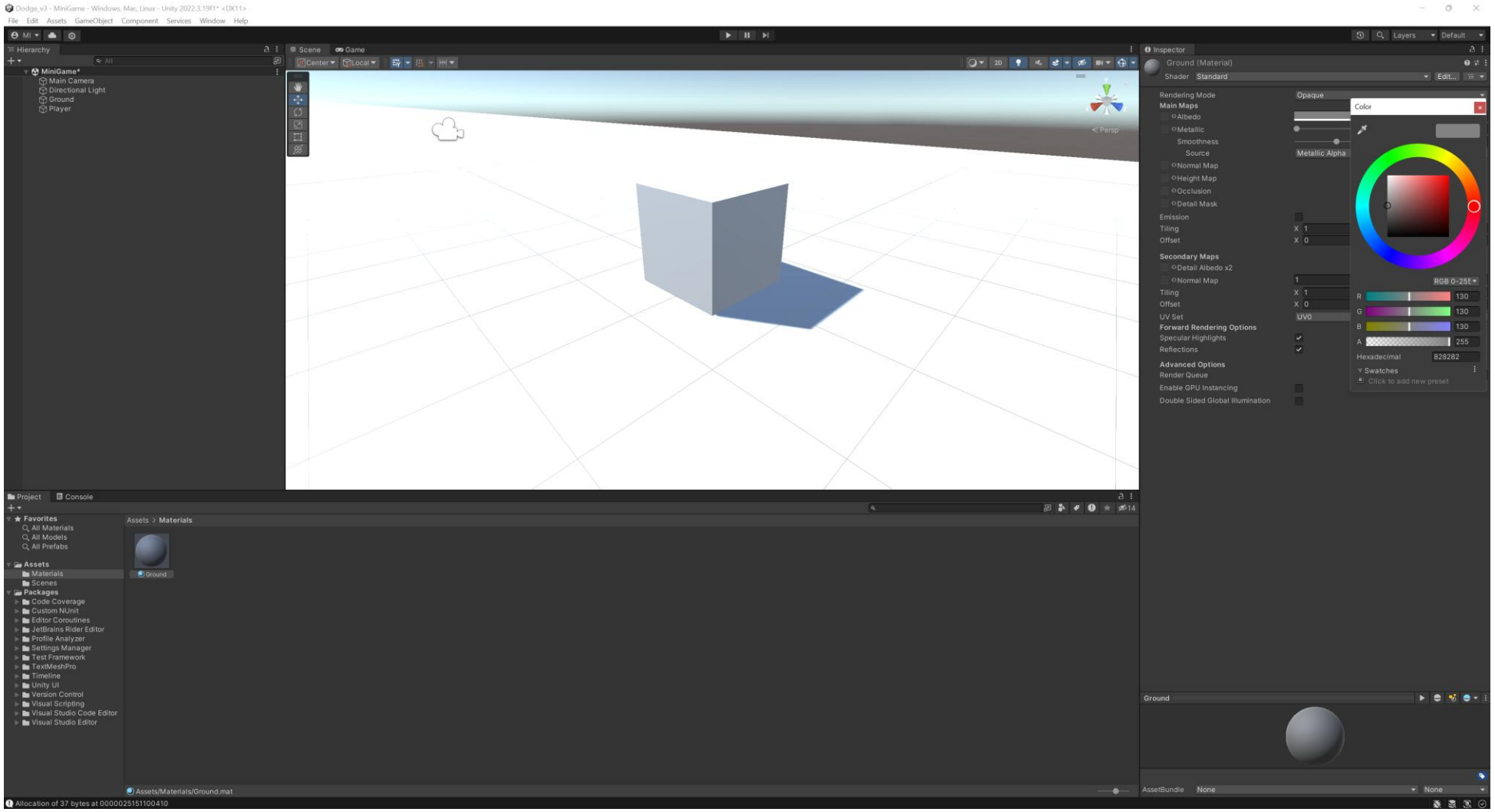
Add colors with Materials

- **Create a new Background material.**
  - In the newly created **Materials** folder, **right-click** > **Create** > **Material** to make a new material, then name it "Ground".
  - In the **Inspector** window, use the foldout (triangle) to expand the **Surface Inputs** module, and select the **Base Map** color picker.
  - Change the color to a pale gray with RGB values of **130**, **130**, and **130**.
  - Make sure the **Metallic Map** is set to **0** and the **Smoothness** is set to around **0.25** for a matte finish.
  - Apply the **Ground** material to the **Ground** GameObject by dragging it from the **Project** window onto the Ground GameObject in the **Scene** view.

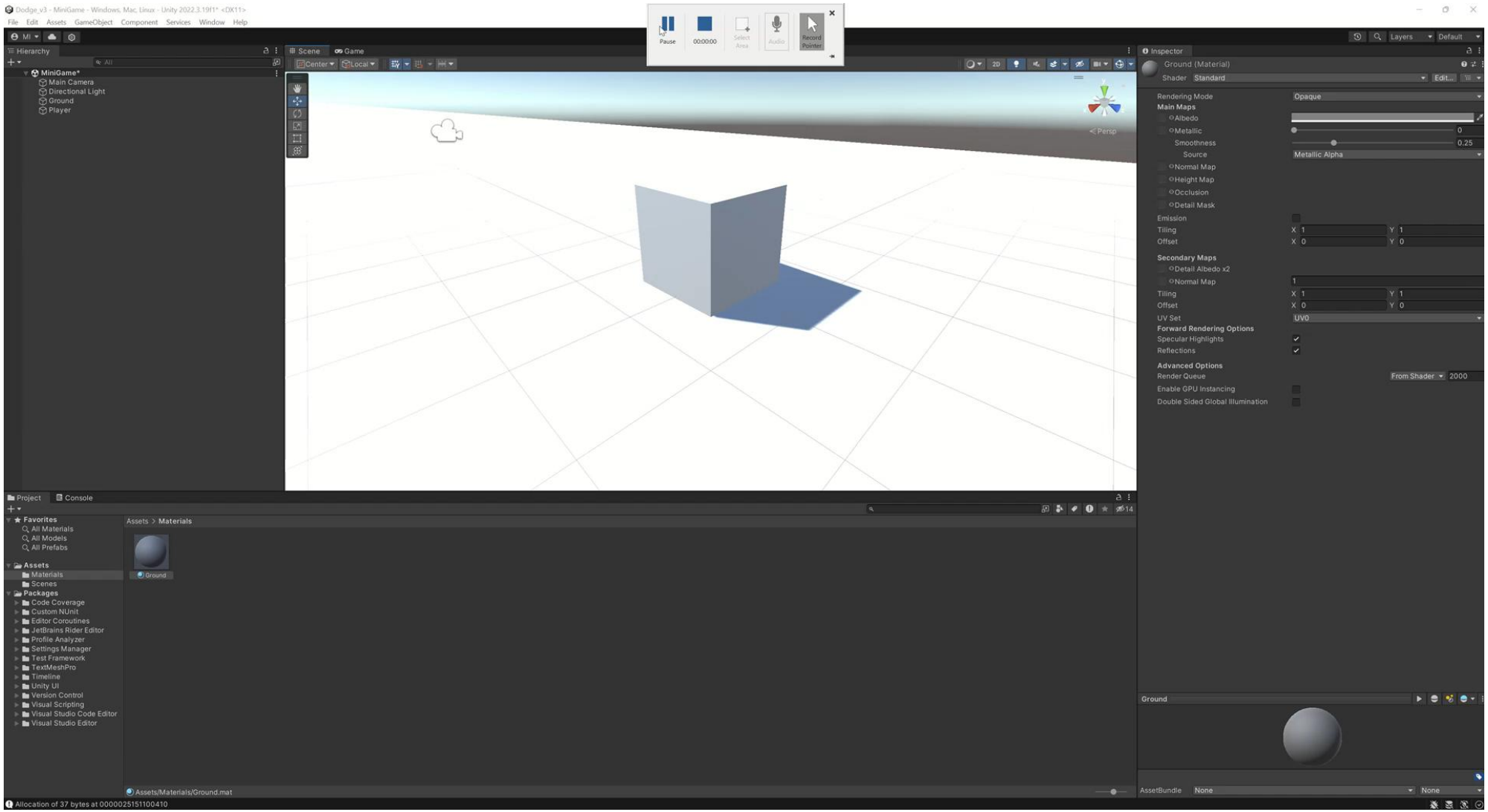
Add colors with Materials



Add colors with Materials



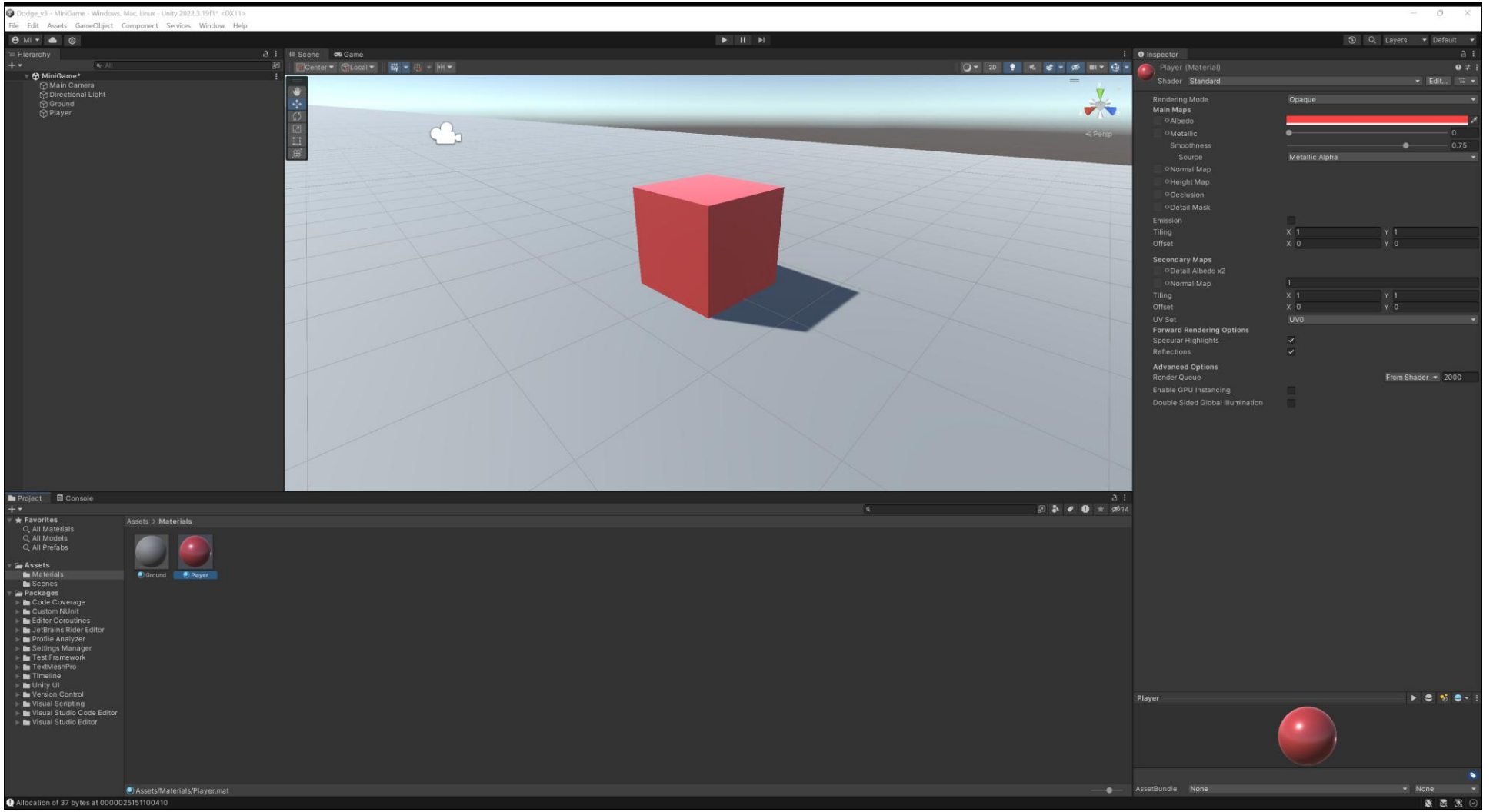
Add colors with Materials



Add colors with Materials

- **Creating a new Player Material.**
  - In the **Materials** folder, create a new material and name it "Player".
  - In the **Inspector** window, adjust the **Base Map** color for the **Player** material — set the RGB values to **255**, **65**, and **65** for a matte red.
  - Set the **Metallic Map** to **0** and change the **Smoothness** to **0.75** for a shiny finish.
  - Apply the **Player** material to the **Player** GameObject by dragging it onto the sphere in the **Scene** view.

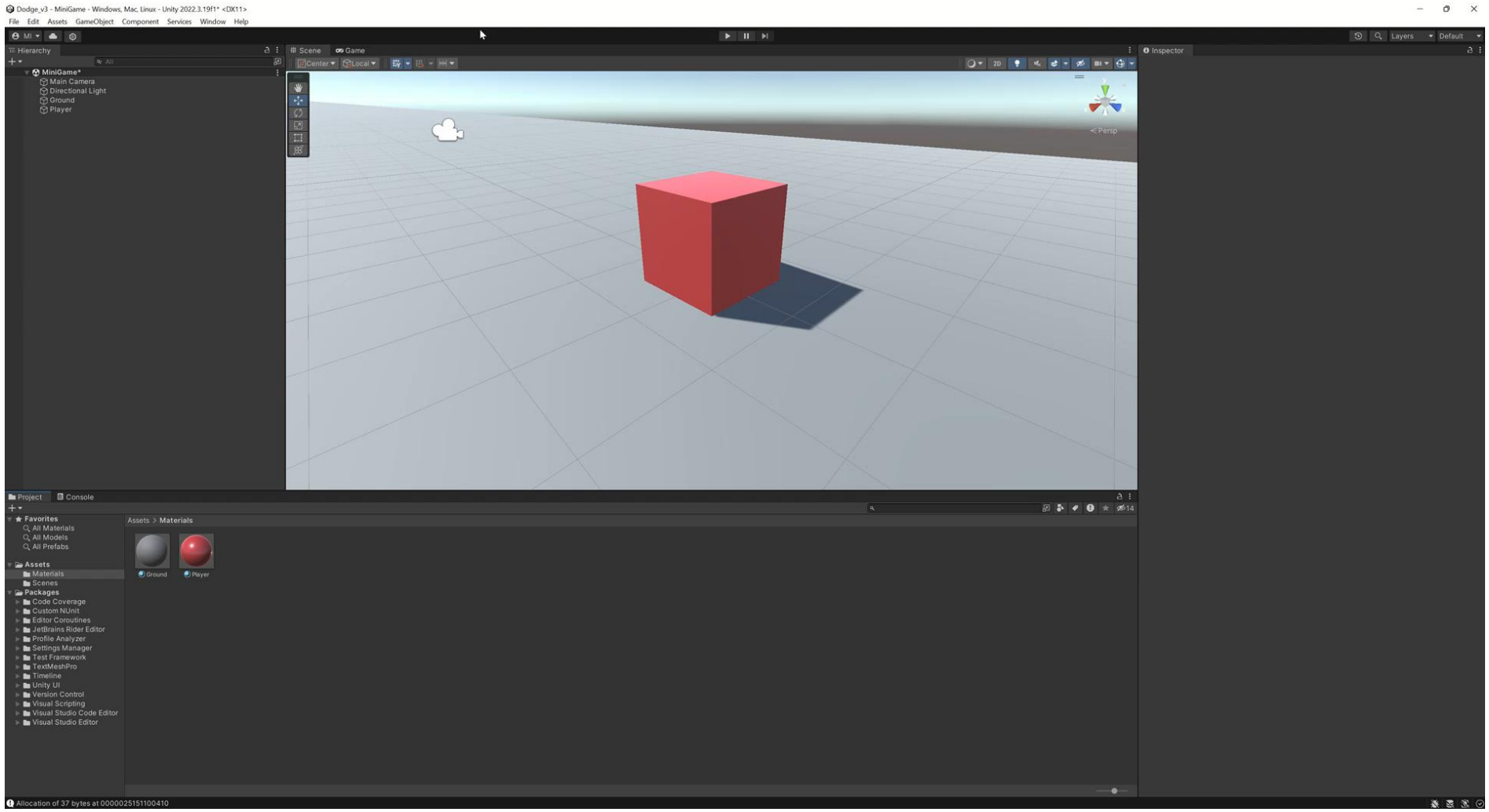
Add colors with Materials



Add colors with Materials

- **Dock the Game View to the right of Scene View.**
  - Left click and hold the game view tab. Move it to the right of the Scene view so that we can see both tabs side by side.

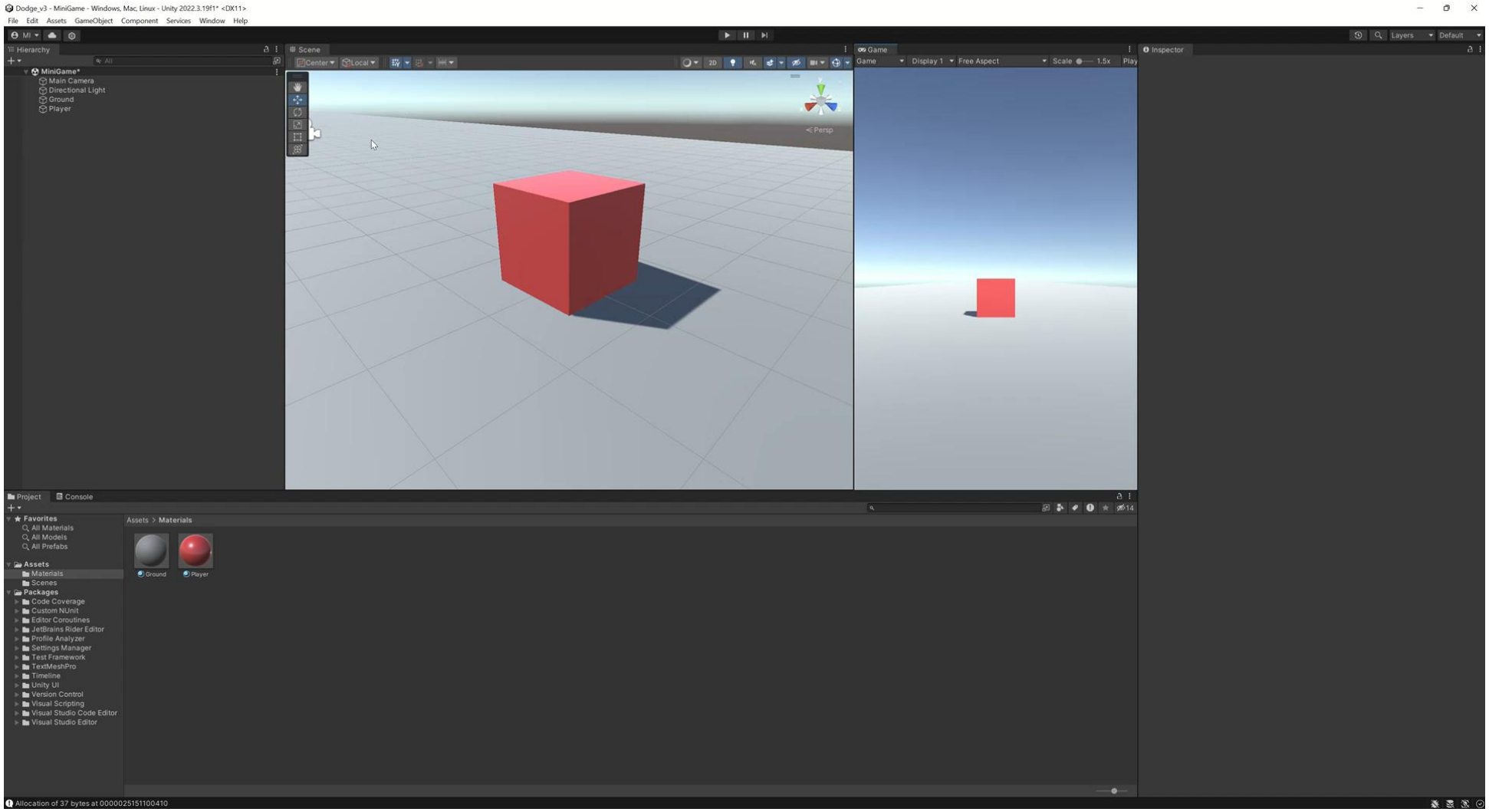
Add colors with Materials



Add colors with Materials

- **Change the skybox.**
  - In the **Hierarchy** window, select the **Main Camera**.
  - In the **Inspector** tab, change the skybox of the Camera component to **Solid Color**. set the RGB values of Background to **0**, **220**, and **255** for a light blue.

Add colors with Materials



Section

Moving the Player

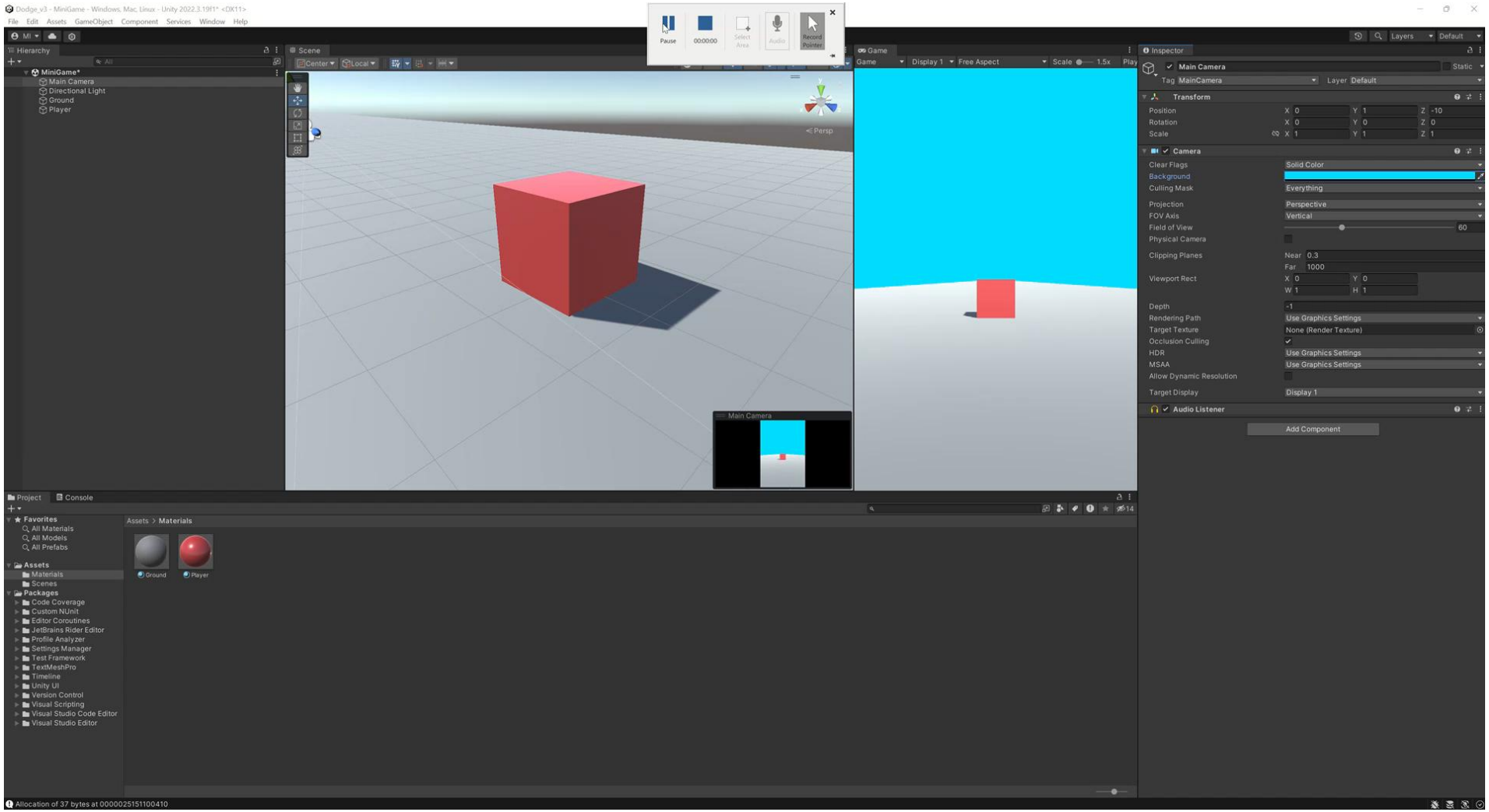
Moving the Player

- (Note: Make sure the **play button** is **unclicked**. Otherwise, all the changes we make will be lost.)

Add a Rigidbody to the Player

- **Add a Rigidbody component.**
  - Select the **Player** GameObject in the **Hierarchy** window.
  - In the **Inspector** Window, select **Add Component**, then search for "Rigidbody" and add the **Rigidbody** component to the **Player** GameObject.
- **Note**: Make sure to select **Rigidbody** and not **Rigidbody 2D**.

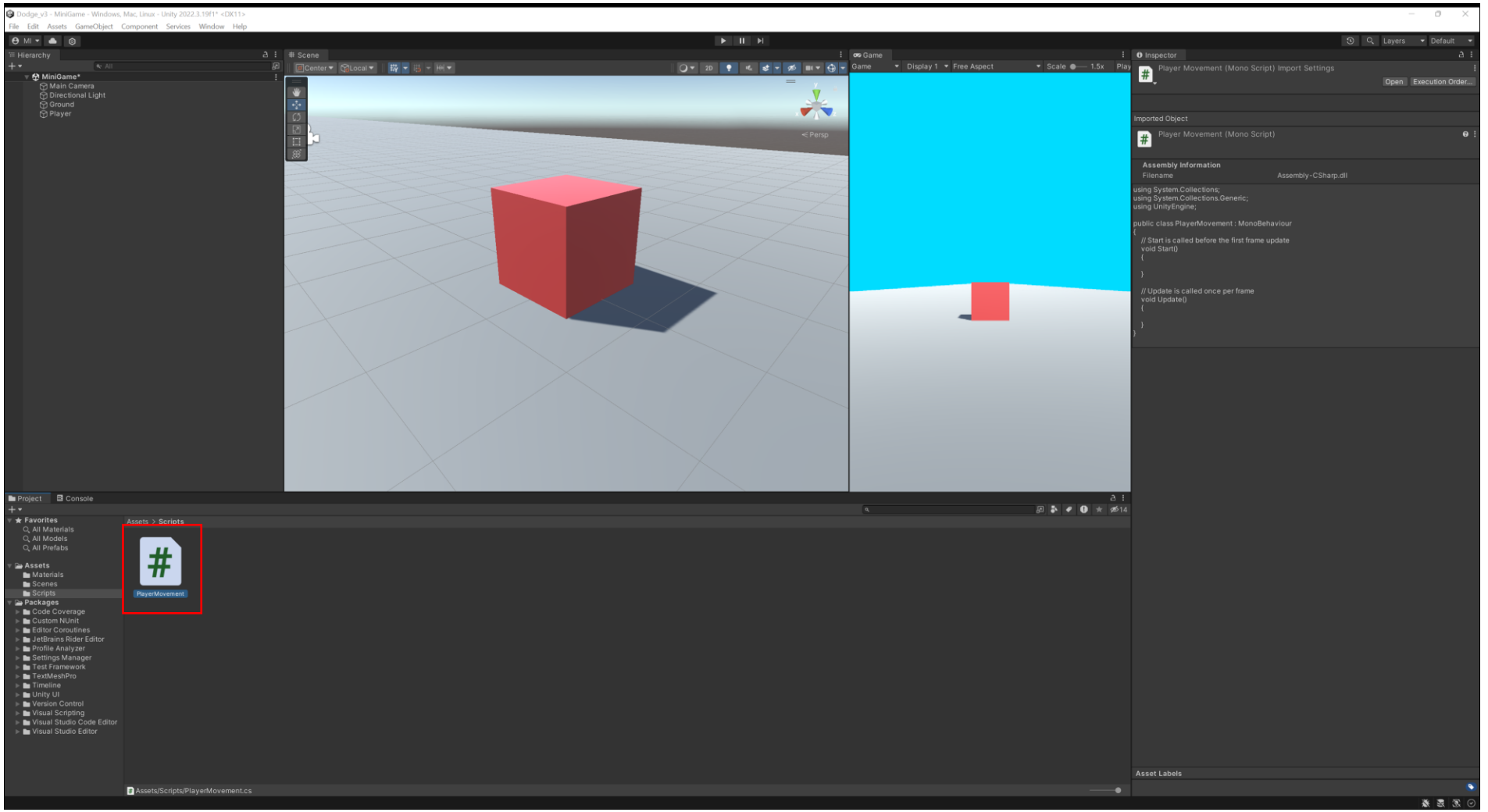
Add a Rigidbody to the Player



Create a new script

- **Create a new PlayerMovement script.**
  - In the **Project** window, create a new folder named "Scripts".
  - With the **Player** GameObject selected, select **Add Component** > **New Script,** then name the new script "PlayerMovement".
  - The created script asset will be at the root level of the **Assets** folder by default. Move the new **PlayerMovement** script asset into the **Scripts** folder.

Create a new script



Create a new script

- **Open the script in a script editor.**
  - Double-click the script asset in the **Project** window to open it in your preferred script editor, usually **VS Code**.
  - **Start() and Update() Functions:** These are special functions within a script that Unity calls automatically during the game's lifecycle.
    - **Start()** is called only once when the object is first created or enabled in the scene.
    - **Update()** is called repeatedly every frame, making it ideal for continuously updating the object's behavior throughout the game.

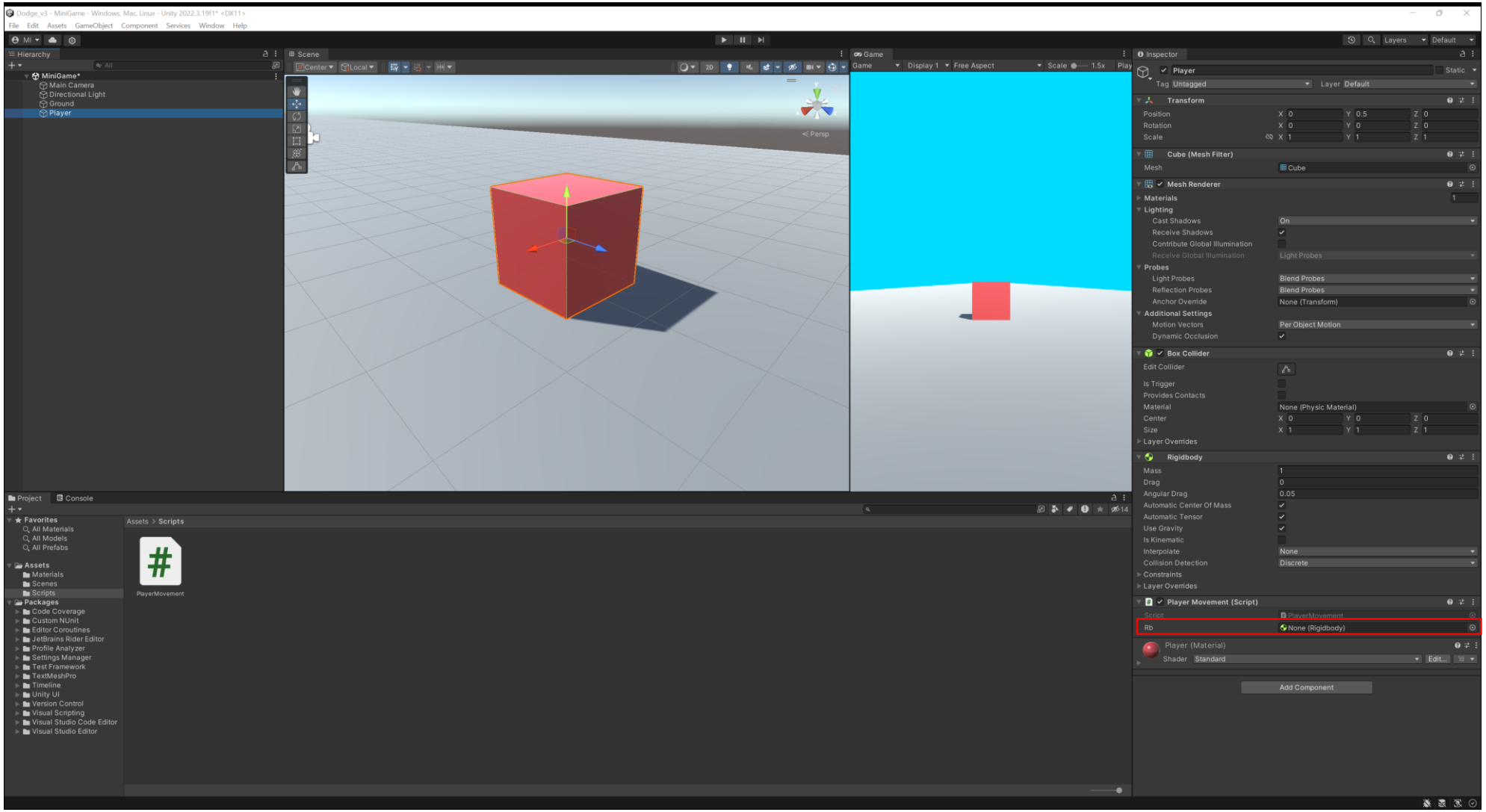
Assign a new Rigidbody variable

- Above the **Start** function, add the following line of code to declare a new variable and save the script (Ctrl + S):

```
public Rigidbody rb;
```

- Next, go to the Unity Editor. After the script gets compiled, you can see that the Player Movement component of the Player object has a Rb field added to it.

Assign a new Rigidbody variable



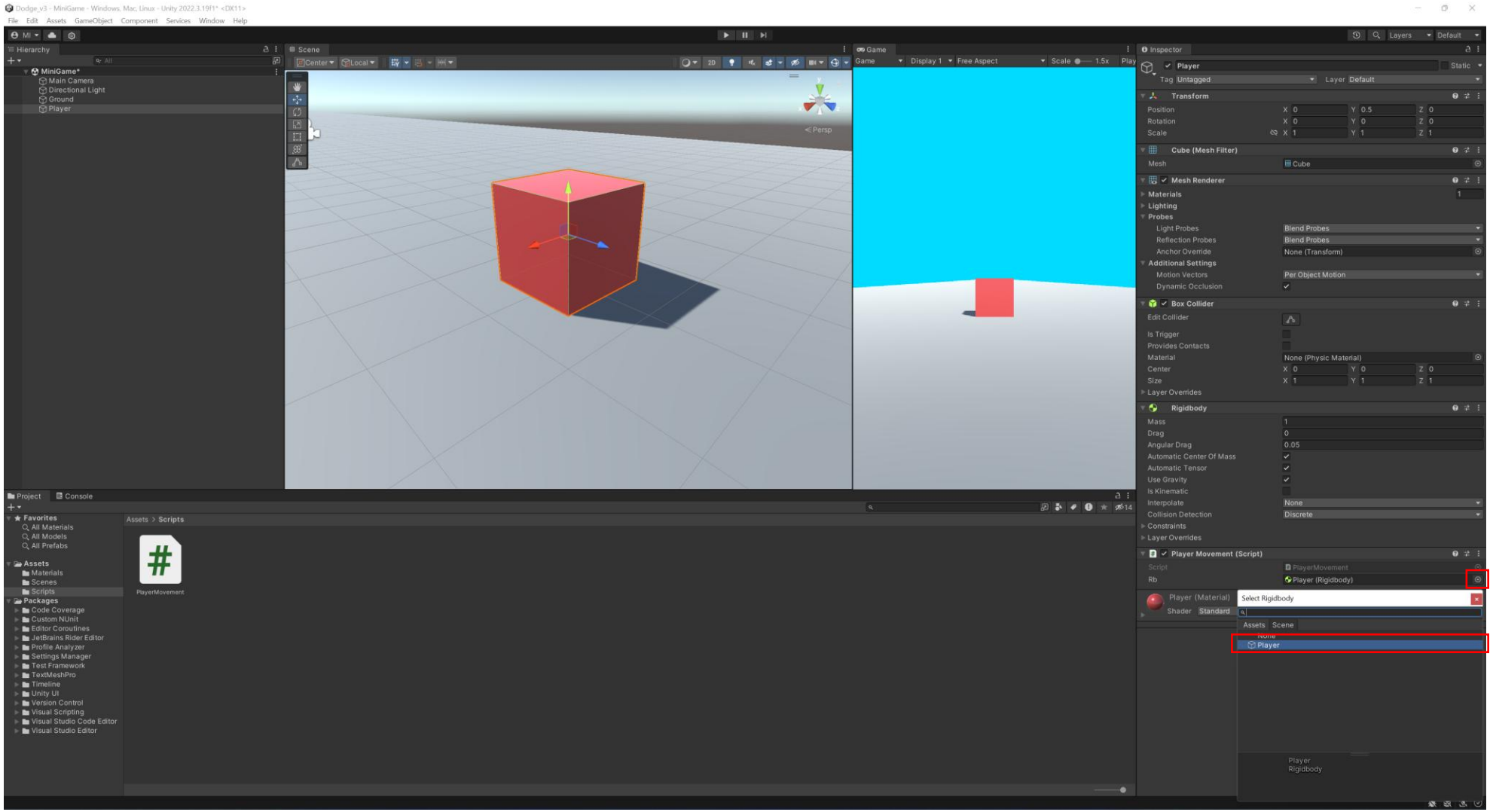
Referencing the Player's Rigidbody

- Click on the small round button on the right side of the Rb field and select Player from the list (or you can drag the Player object from the Hierarchy panel to the Rb field).

- This creates a reference to the Rigidbody component of the Player object from the rb variable in our PlayerMovement script.

- This allows us to use the rb variable to modify the Rigidbody of the Player.

Assign a new Rigidbody variable



Apply force to the Player

- In our game, we want the player to move forward automatically and move the left or to the right when we press the left or right button on our keyboard respectively.

Apply force to the Player

- **Physics Control:** Let's explore some key properties and methods of the Rigidbody component:
  - **useGravity**: Enable or disable gravity for the object.
  - **AddForce**: Apply a force to the object, causing it to move in a specific direction.
  - **Time.deltaTime**: This value is used to ensure your force calculations are frame-rate independent, leading to smoother and more consistent physics behavior across different machines.

Apply force to the Player

- Add the following lines below **`public Rigidbody rb;`**:

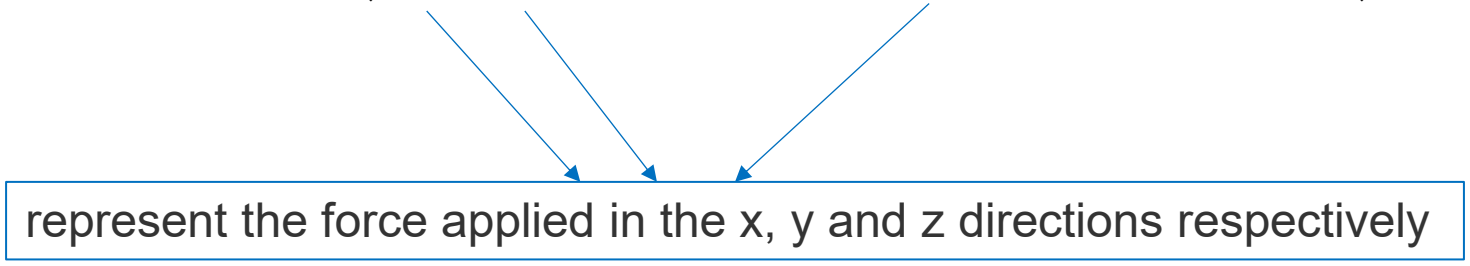
```
public float forwardForce = 1000f;

public float sidewaysForce = 100f;
```

- (Note: We can change the values of these variables in the Unity editor later if needed.)

- We will not need the Start() function here. So we can remove it.

- In the **Update** function body, add the following code:

```
rb.AddForce(0f, 0f, forwardForce * Time.deltaTime);
```

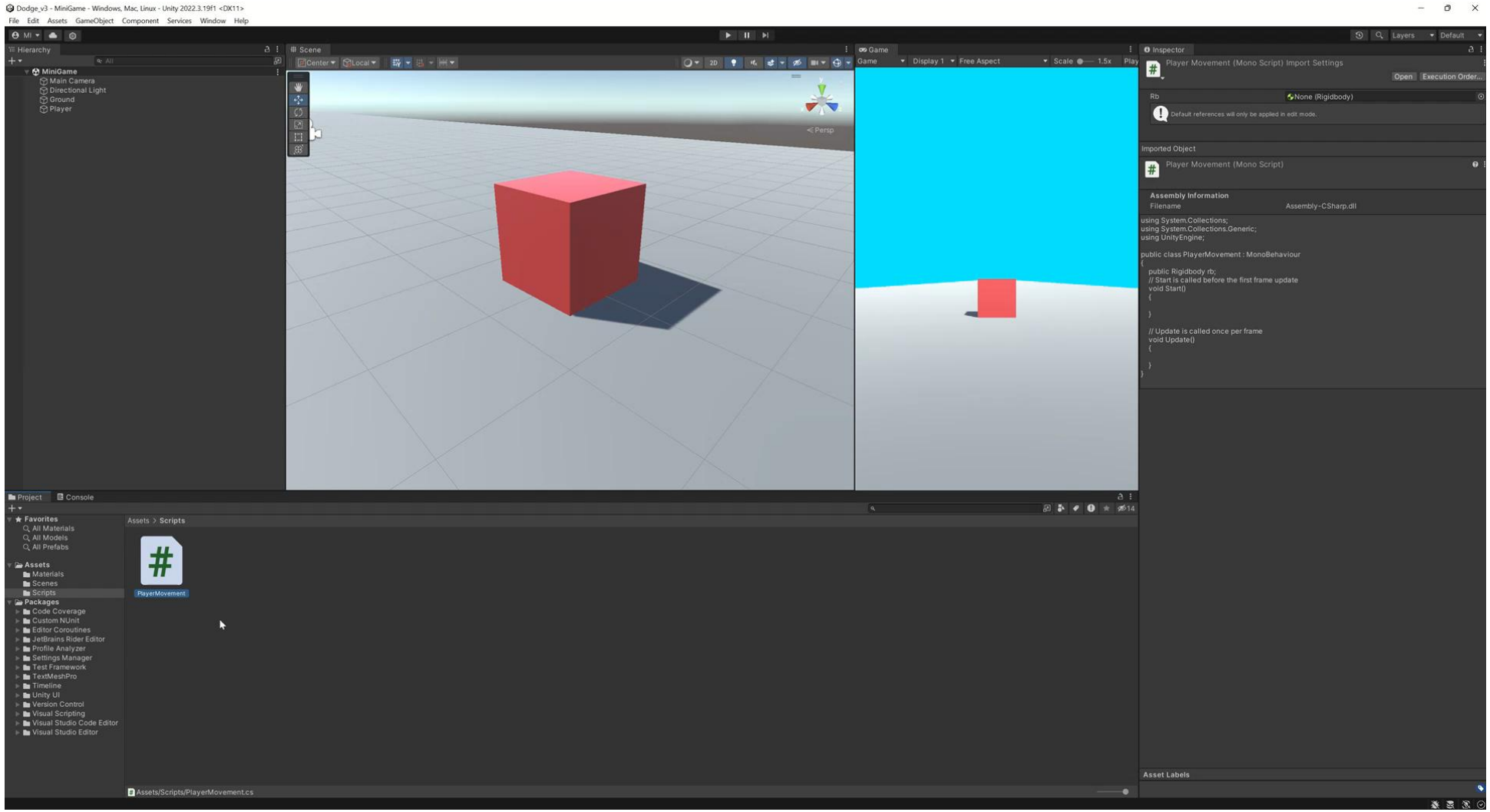


Apply force to the Player

- If we now go to the Unity editor and click the play button, we will see the player moving forward.

- Make sure to unclick the play button afterward.

Apply force to the Player



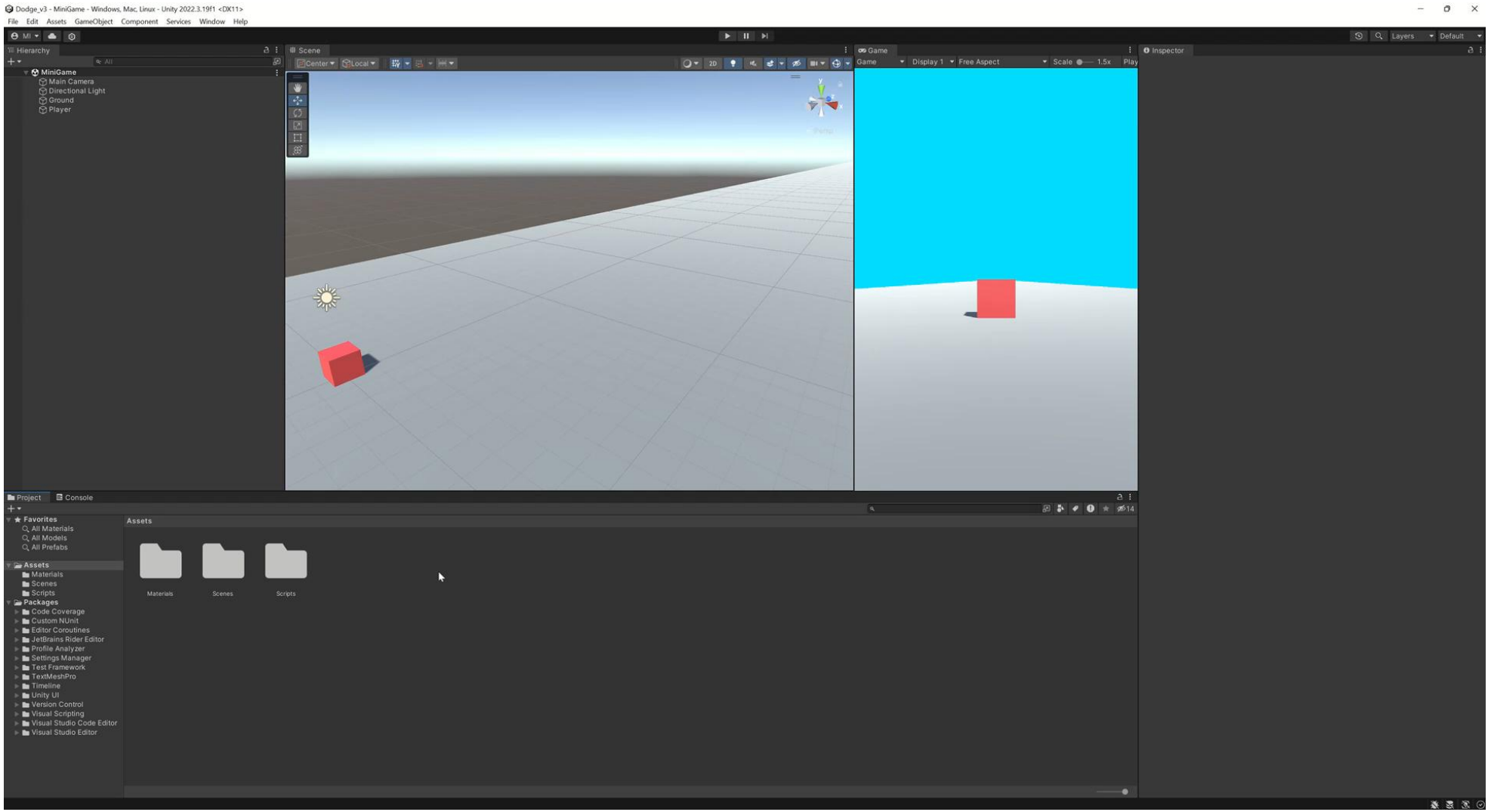
Apply force to the Player

- To address the uncontrolled spinning or rotating of the player object, follow these steps:

1. Create a Slippery Physic Material:
   - In the Project panel, make sure you are in the Asset folder
   - Right-click in the Project panel and choose "Create" > "Physic Material".
   - Name it “Slippery" and set both dynamic and static friction values to 0.

2. Apply the Material to the Ground:
   - Select the ground object in the Scene view.
   - Drag and drop the “Slippery" material onto the ground object.
   - This reduces friction between the player and the ground, allowing for smoother movement and preventing uncontrolled spinning.

Apply force to the Player



Add sideways movement

- To incorporate sideways movement to our player, add the following code segment in the Update function:

```
if (Input.GetKey("right"))
{
    rb.AddForce(sidewaysForce * Time.deltaTime, 0f, 0f, ForceMode.VelocityChange);
}
if (Input.GetKey("left"))
{
    rb.AddForce(-sidewaysForce * Time.deltaTime, 0f, 0f, ForceMode.VelocityChange);
}
```

Add sideways movement

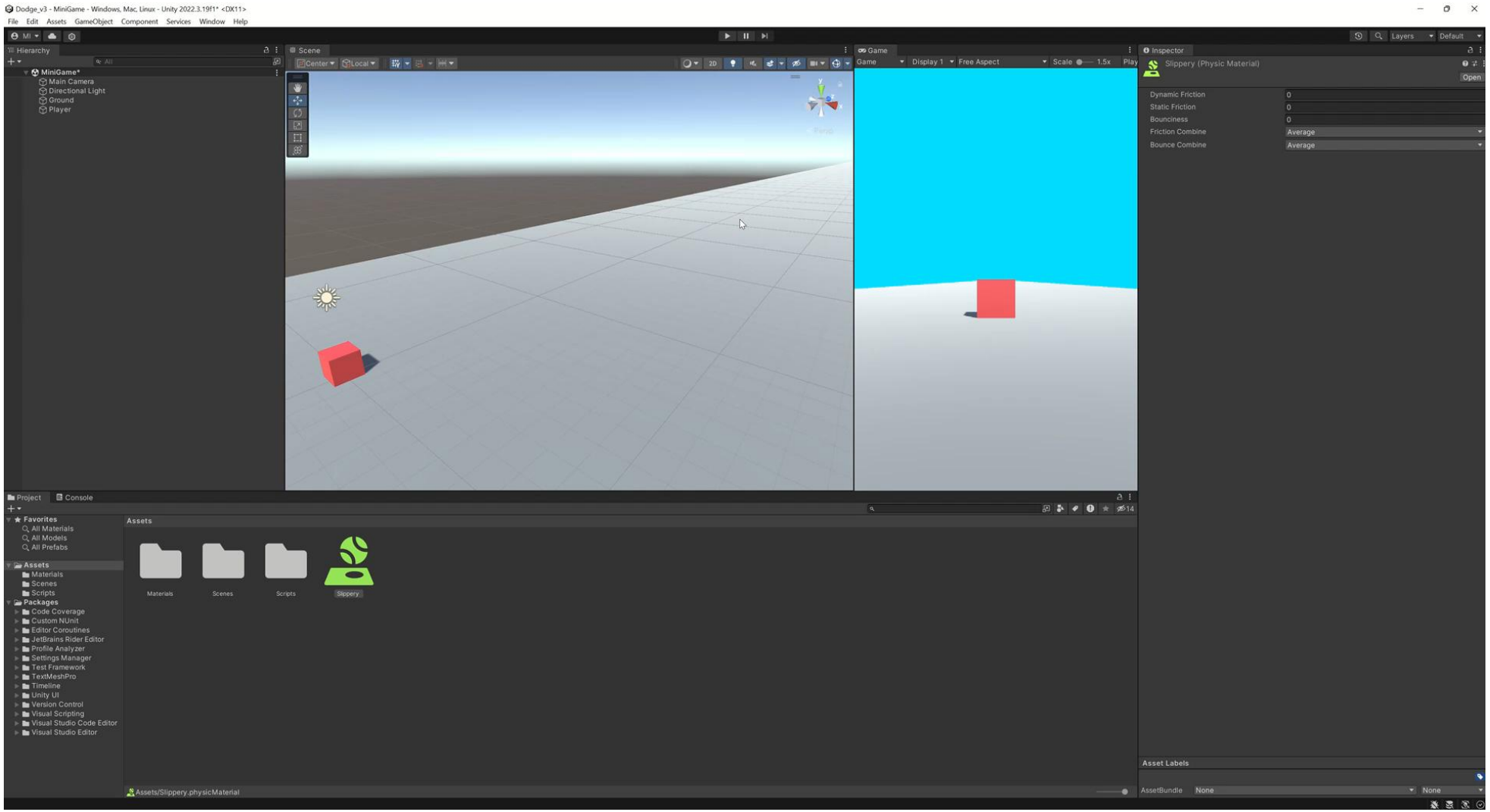
- To add sideways movement to our player, we've introduced a new variable **`sidewaysForce`** with a value of **`100f`**, which determines the strength of the sideways force.

- In the **`Update()`** function, we've added conditions to check if the 'right' or 'left' arrow keys are pressed. If the 'right' arrow key is pressed, we apply a force to the right using **`rb.AddForce(sidewaysForce * Time.deltaTime, 0f, 0f, ForceMode.VelocityChange);`**. Similarly, if the 'left' arrow key is pressed, we apply a force to the left using **`rb.AddForce(-sidewaysForce * Time.deltaTime, 0f, 0f, ForceMode.VelocityChange);`**.

Add sideways movement

- Here's what's happening in these lines:
  - **`Input.GetKey("right")`** and **`Input.GetKey("left")`** check if the 'right' or 'left' arrow keys are being pressed.
  - **`rb.AddForce()`** applies a force to the player object.
  - **`sidewaysForce * Time.deltaTime`** determines the strength of the sideways force applied.
  - The **`0f, 0f`** values in the **`AddForce()`** method mean there's no force applied in the vertical or forward/backward direction, ensuring the movement remains purely sideways.
  - **`ForceMode.VelocityChange`** ensures that the force is applied instantly, allowing for immediate movement in response to key presses."

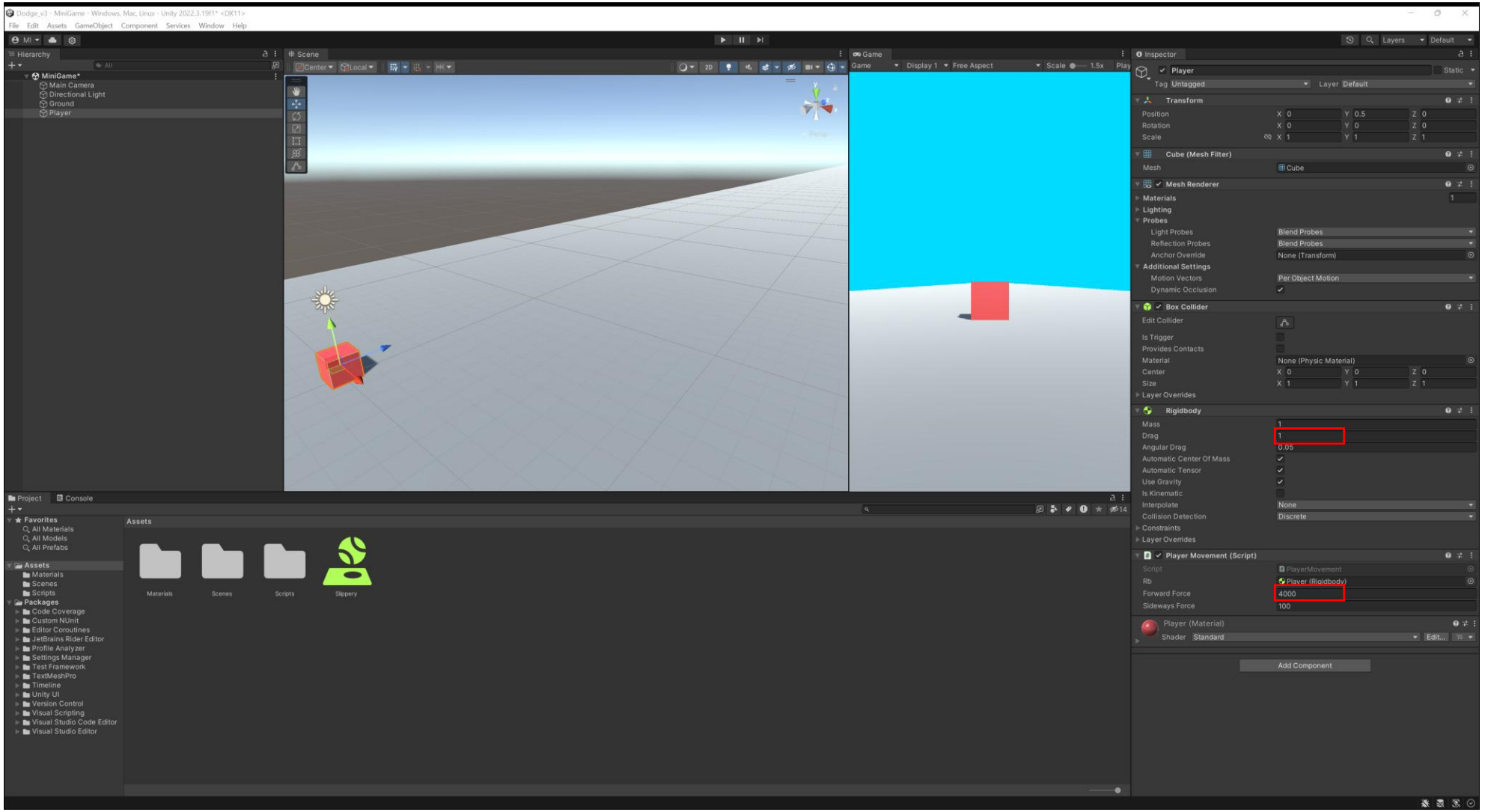
Add sideways movement



Add sideways movement

- The forwardForce looks to be slow when we hit the play button. Let's change it to 4000.
- Also set the drag value to 1 from 0 in the Rigidbody component of the Player. This adds air resistance and make the player's movement smoother.

Add sideways movement



Section

Moving the Camera

Making the Camera follow the Player

- **Approach 1:** Attach camera as child of the player object.
  - **Drawback:** Whenever the player rotates, the camera will also rotate, potentially causing a disorienting experience for the user.

- **Approach 2:** Script for camera follow
  - **Drawback:** Whenever the player rotates, the camera will also rotate, potentially causing a disorienting experience for the user.

Making the Camera follow the Player

- **Create the CameraMovement script.**
  - With the **Main Camera** GameObject selected in the **Hierarchy** window, select **Add Component** > **New script** in the **Inspector** window.
  - Name your new script “CameraMovement”.
  - In the **Project** window, move the script from the root **Assets** folder into the **Scripts** folder.
  - Open the new script for editing.

- **Declare player and offset variables.**
  - Inside the first curly brace, add the following lines of code to declare two new variables:

```
public Transform player;
 Public Vector3 offset;
```

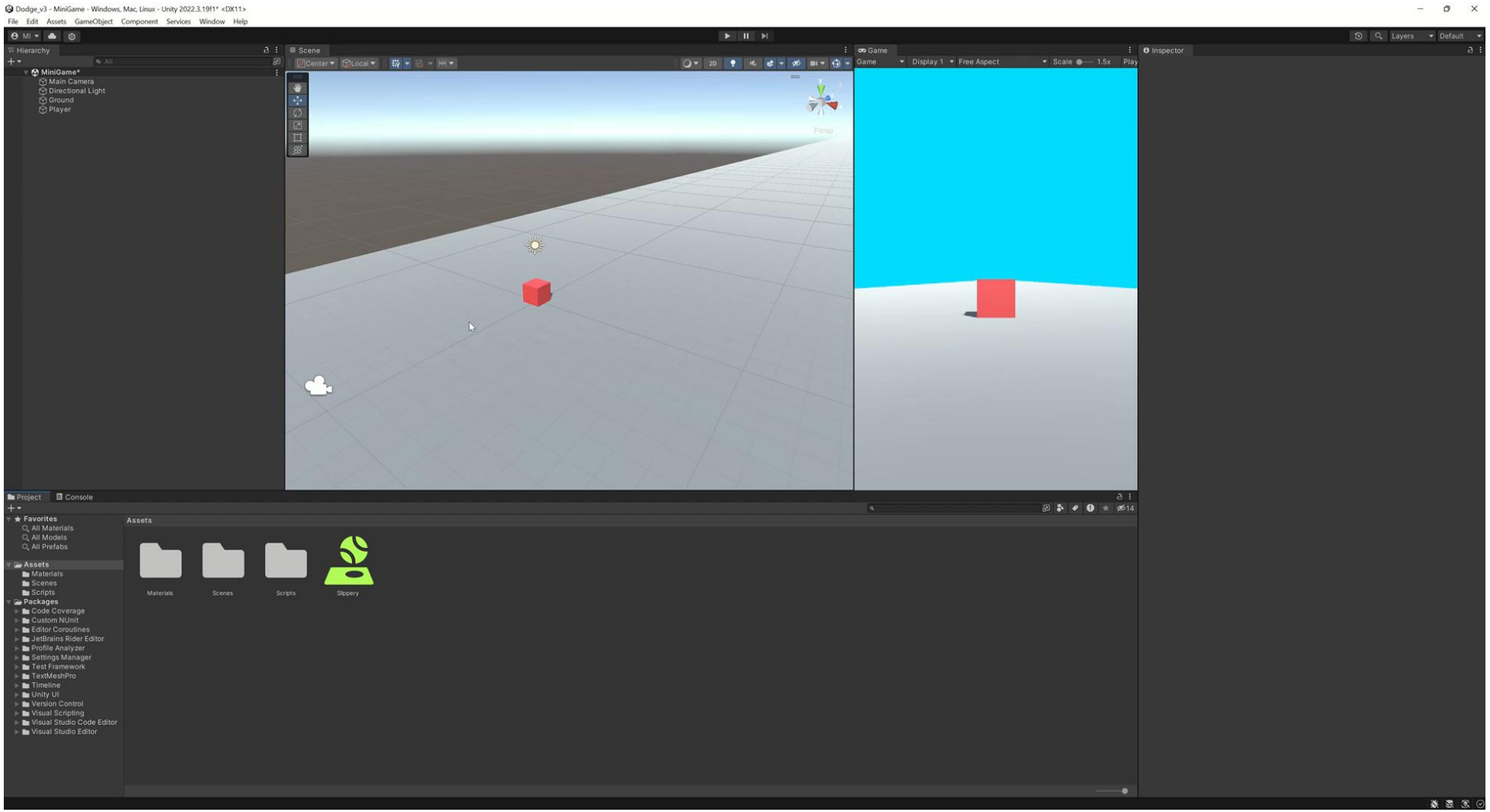
Making the Camera follow the Player

- **Set the camera position in Update.**
  - In the **Update** function, inside the first curly brace, add the following line of code:

```
transform.position = player.transform.position + offset;
```

- **Assign the Player GameObject variable in the Inspector window and Set the offset value.**
  - Make sure you have saved the **CameraMovement** script, then return to Unity.
  - Drag the **Player** GameObject from the **Hierarchy** window into the **Player** slot in the **CameraController** component.
  - For the offset, Set X = 0, Y = 2 and Z = -8.

Making the Camera follow the Player



Section

Creating Obstacles and Detecting Collisions

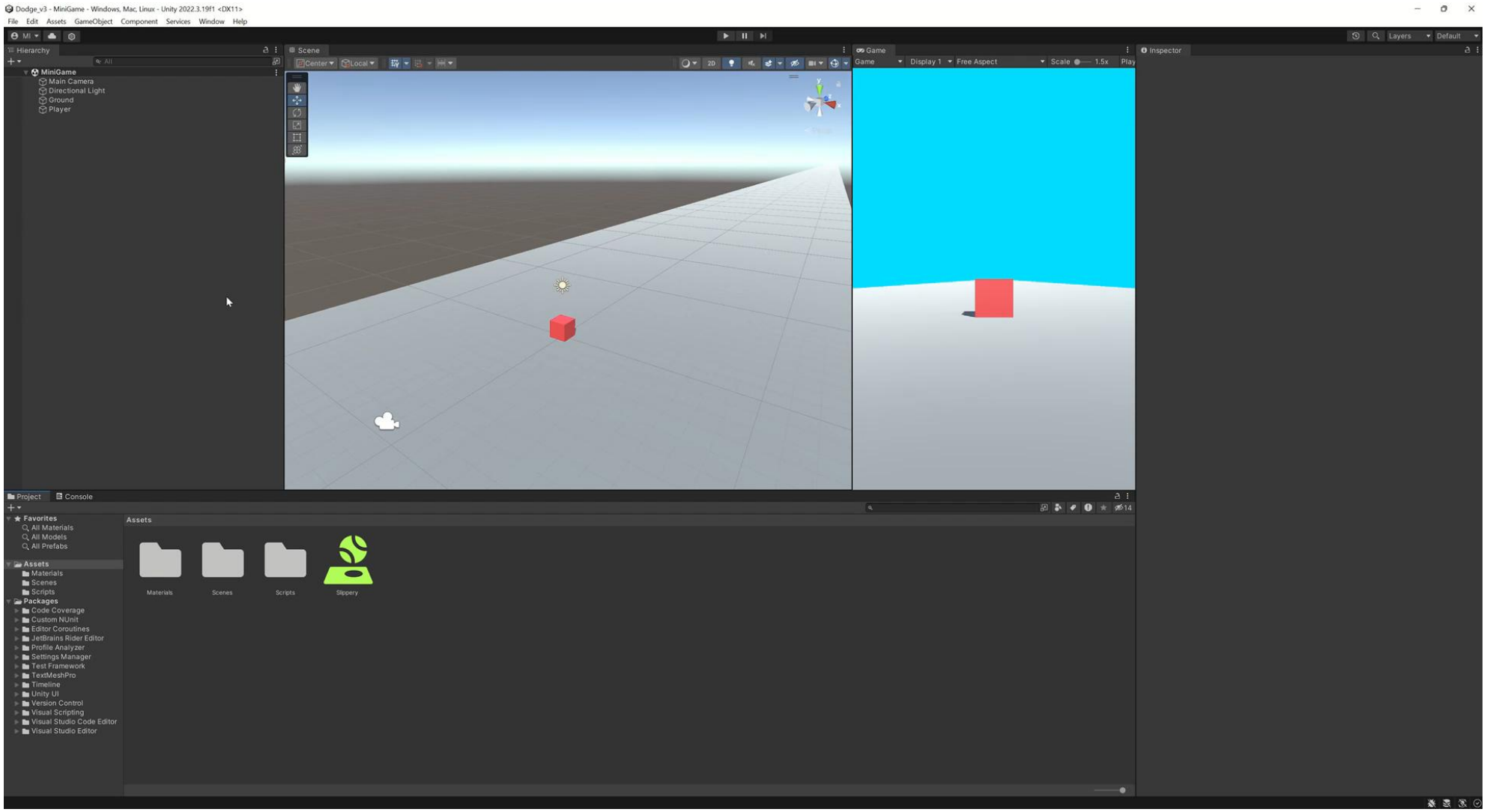
Creating an Obstacle

- **Create an Obstacle:** Create a new cube object in the scene to represent an obstacle the player can collide with.
  - (Select the main camera and set its Y position = 3 to get a better view in the Game View.)
  - Reset obstacle's position in the (Transform Component in the Inspector tab).
  - Then set its Y position = 0.5 and Z position = 35.

- **Obstacle Properties:** Customize the obstacle's appearance by adjusting its color (R = 60, G = 60, B = 60), smoothness (= 0.75, optional), and size (Scale → X = 3).

- **Adding Rigidbody and Mass:** Attach a Rigidbody component to the obstacle. Adjust the obstacle's mass to influence its behavior during collisions. A higher mass (such as 2) will make the obstacle less movable upon impact with the player.

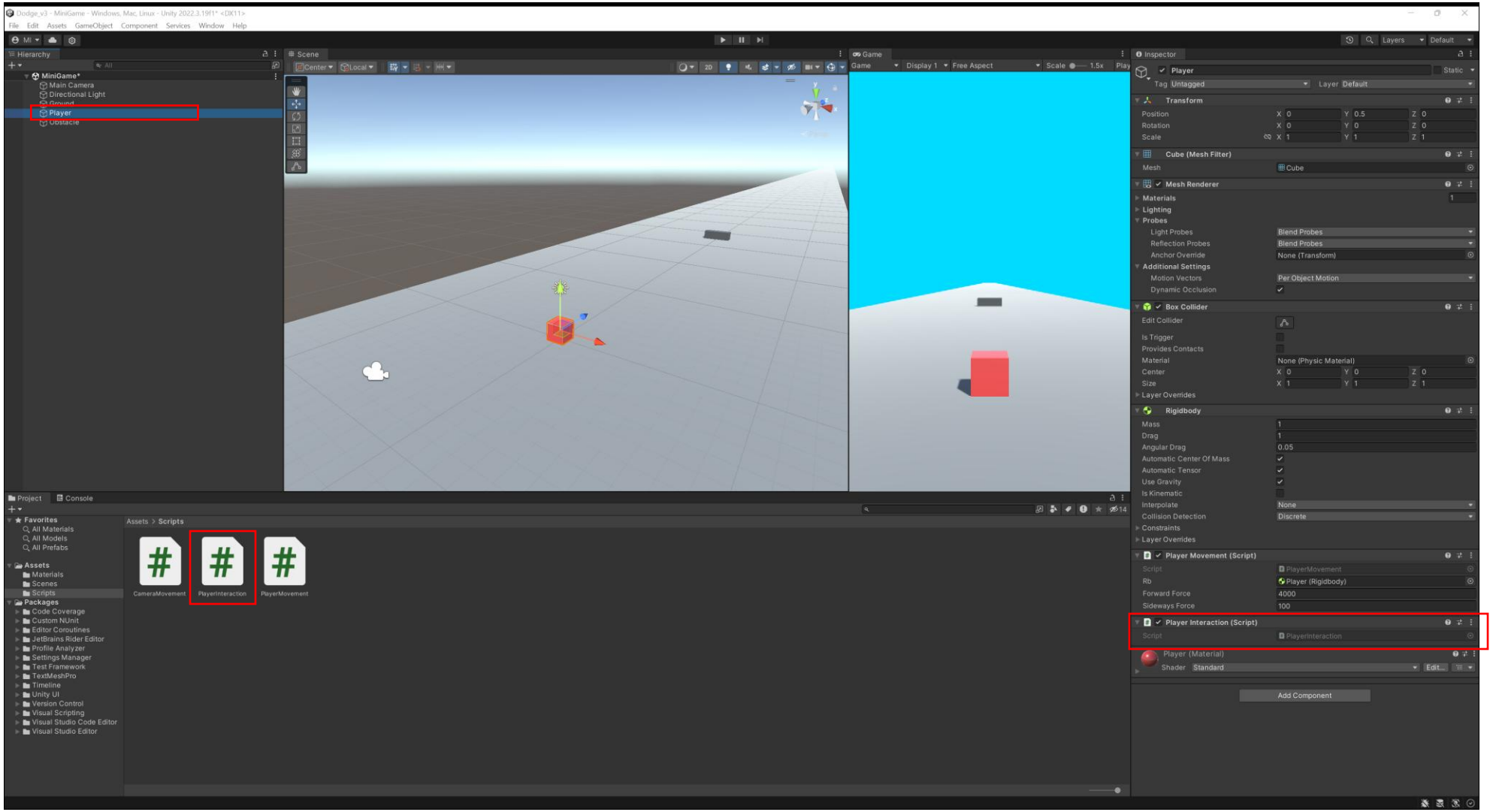
Creating an Obstacle



Detecting Collision

- **Scripting Collisions (Separate Script)**
  - While collision detection can be implemented in the player movement script, it's often cleaner to create a dedicated script for better organization.
  - Select the Player object and add a new script to it named **PlayerInteraction**.
  - Place **PlayerInteraction.cs** in the Scripts folder.

Detecting Collision



OnCollisionEnter Function

- Unity provides built-in functions for handling collisions. Its basic structure is as follow:

```
private void OnCollisionEnter(Collision other) {
    // Collision detection code goes here
}
```

Refers to the object with which the player collided/came into contact.

- The OnCollisionEnter() function gets called whenever the player object collides with another object in the scene.
- We can use Debug.Log statements within this function to verify collision detection. Initially, this might show a collision with the ground since it is a game object as well.

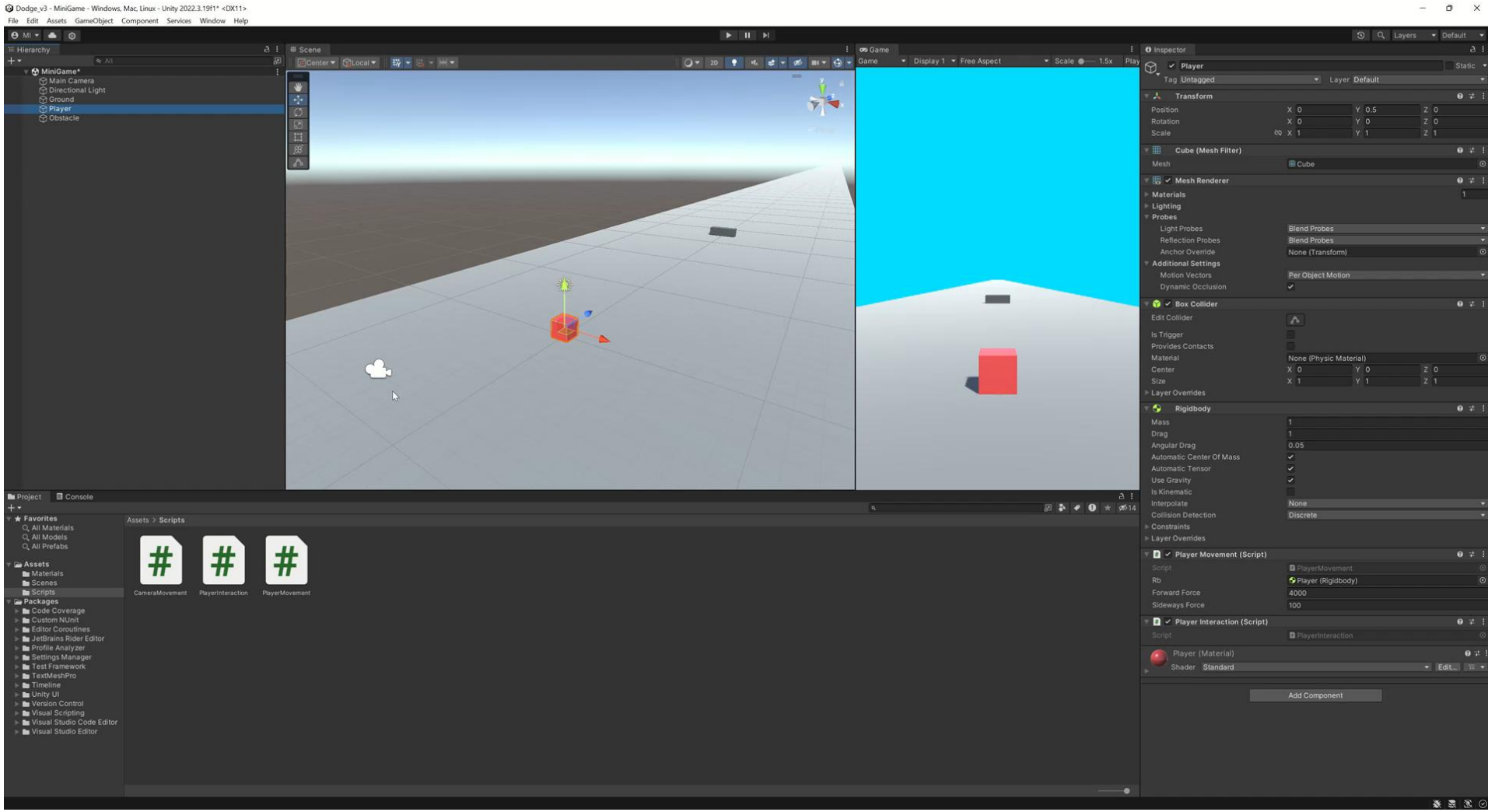
OnCollisionEnter Function

- We can use Debug.Log statements within this function to verify collision detection. Initially, this might show a collision with the ground since it is a game object as well.
- In the PlayerInteraction.cs script:
  - Remove the Start function (We won't need it here).
  - Within the first curly braces, add the following code segment (above Update fucntion).

```
private void OnCollisionEnter(Collision other) {
    Debug.Log("Player hit something!");
}
```

- Head back to Unity and click the play button. Select the console tab on the lower portion of the editor.
- We will see that the message "Player hit something!" is displayed. The message gets displayed twice, one for the ground (which we don't want for the game) and the other one for the obstacle.

OnCollisionEnter Function



Identifying Colliding Objects

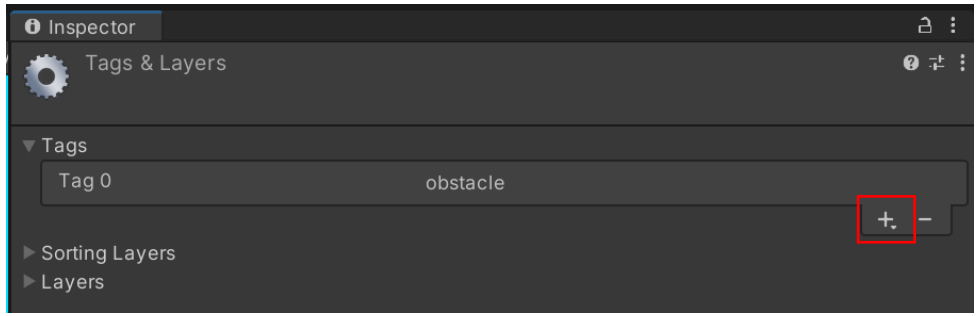
- In our game, we need to detect when the player collides with obstacles. This allows us to respond accordingly, such as halting player movement upon collision.

- While using the collider's name for identification can be a starting point, it's not a reliable approach for complex games.
  - Names can be easily duplicated or forgotten, leading to potential issues.

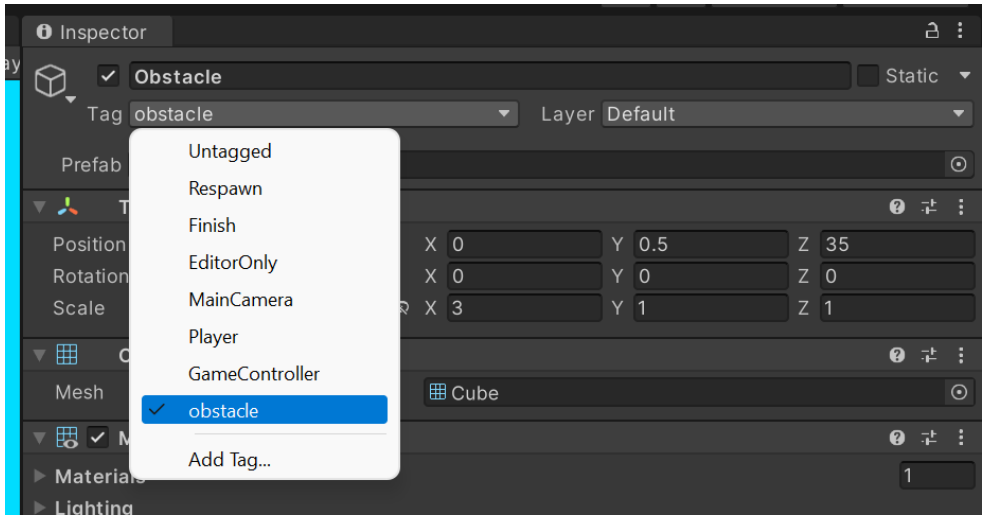
- **Tags** provide a more robust way to identify objects in Unity.
  - Assign a unique tag to the obstacle object (e.g., “obstacle").

Identifying Colliding Objects

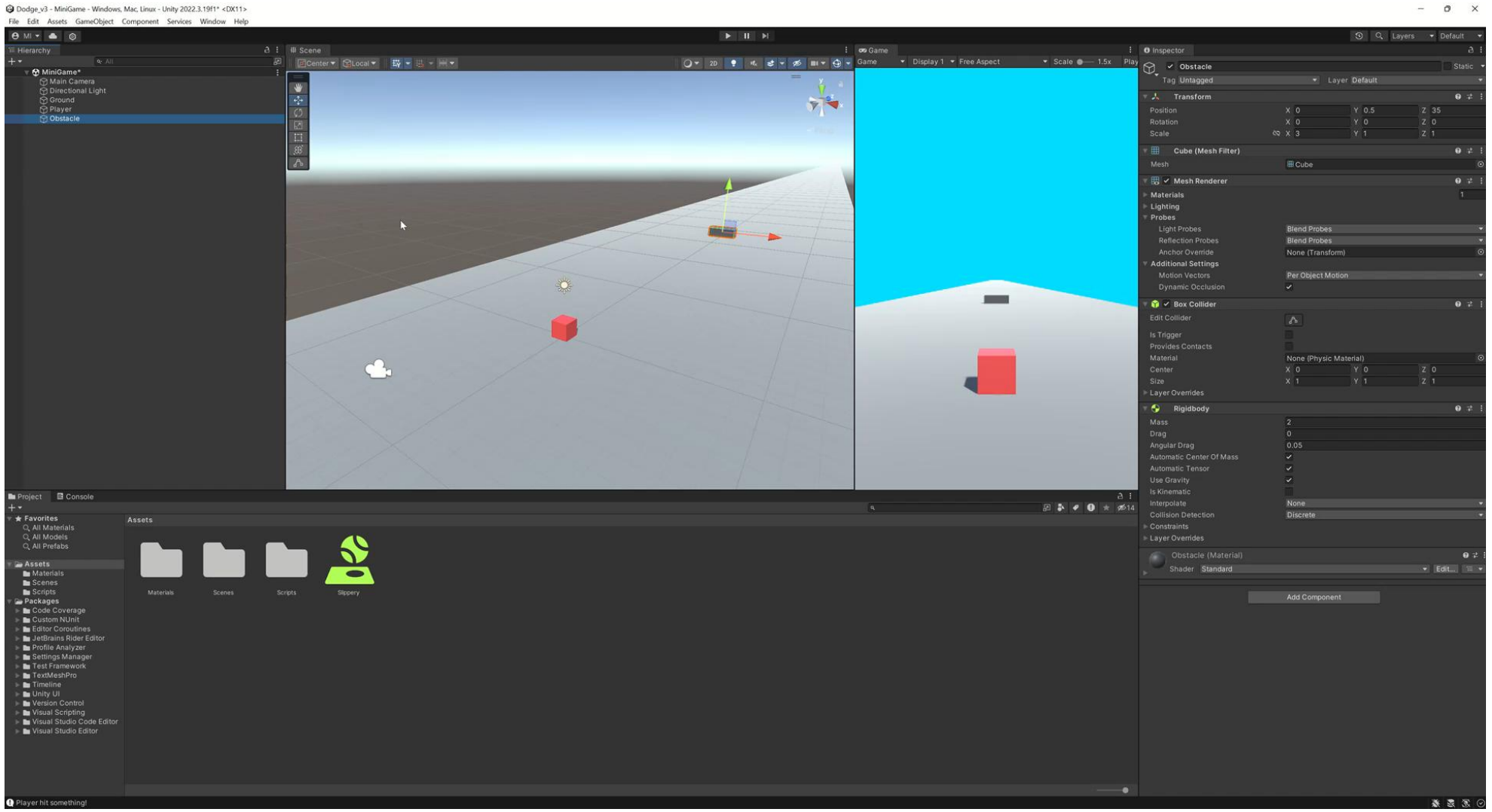
- **Create a new tag.**
  - Select the Obstacle object in the Hierarchy panel.
  - At the top of the **Inspector** window, from the **Tag** dropdown menu, select **Add Tag**.
  - Select the **Add** (**+**) button to add a new tag, then name it “**obstacle**”.
  - **Important:** This is case sensitive, so be careful — it needs to be exactly the same spelling and capitalization that you use in the script later on.



- **Apply the tag to the Obstacle object.**
  - With the **Obstacle** object still selected, use the **Tag** dropdown menu to select the new “obstacle” tag from the list.



Identifying Colliding Objects



Stopping Player Movement on Collision

- **Using other.gameObject.CompareTag**
  - Within OnCollisionEnter(), we'll use other.gameObject.CompareTag("obstacle") to check if the collided object is tagged as an 'obstacle'.

- To disable player movement upon collision with an obstacle:
  - Inside **PlayerInteraction.cs**, declare a public variable of type PlayerMovement named 'movement' within the first curly braces.

```
public PlayerMovement movement;
```

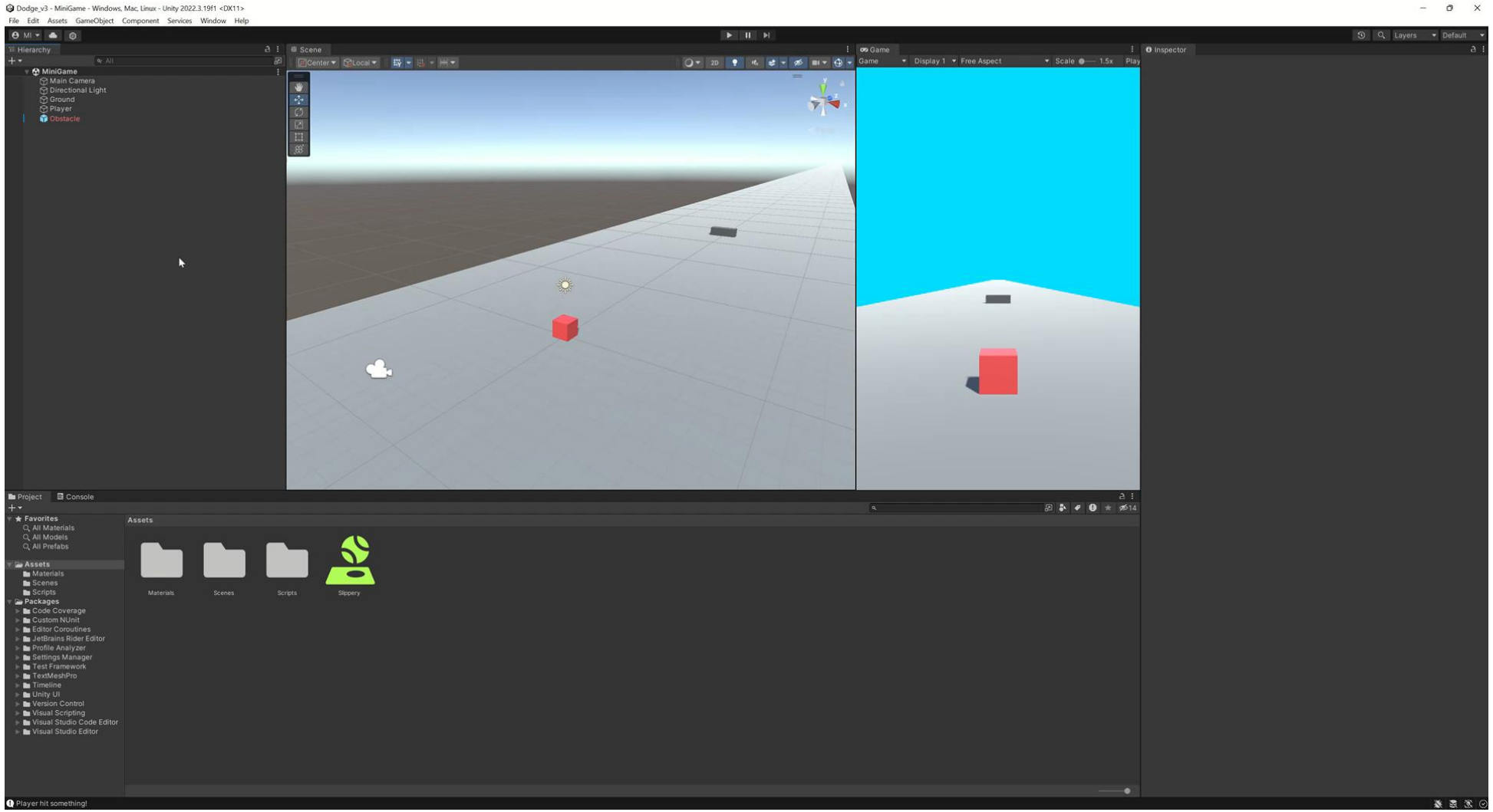
  - Replace the code within **OnCollisionEnter** function. Changed **OnCollisionEnter** looks like this:

```
private void OnCollisionEnter(Collision other) {
    if (other.gameObject.CompareTag("obstacle"))
    {
        Debug.Log("Player hit an obstacle!");
        movement.enabled = false;
    }
}
```

Stopping Player Movement on Collision

- Save the script.
- Go back to Unity editor.
- Finally, in the unity editor, drag the Player object from Hierarchy panel to the movement field of the Player Interaction component of the Player object.
- Click the play button and check the console panel to verify that the “Player hit an obstacle!” is displayed when the Player collides with the obstacle.
- (Make sure to unclick the play button afterwards.)

Stopping Player Movement on Collision



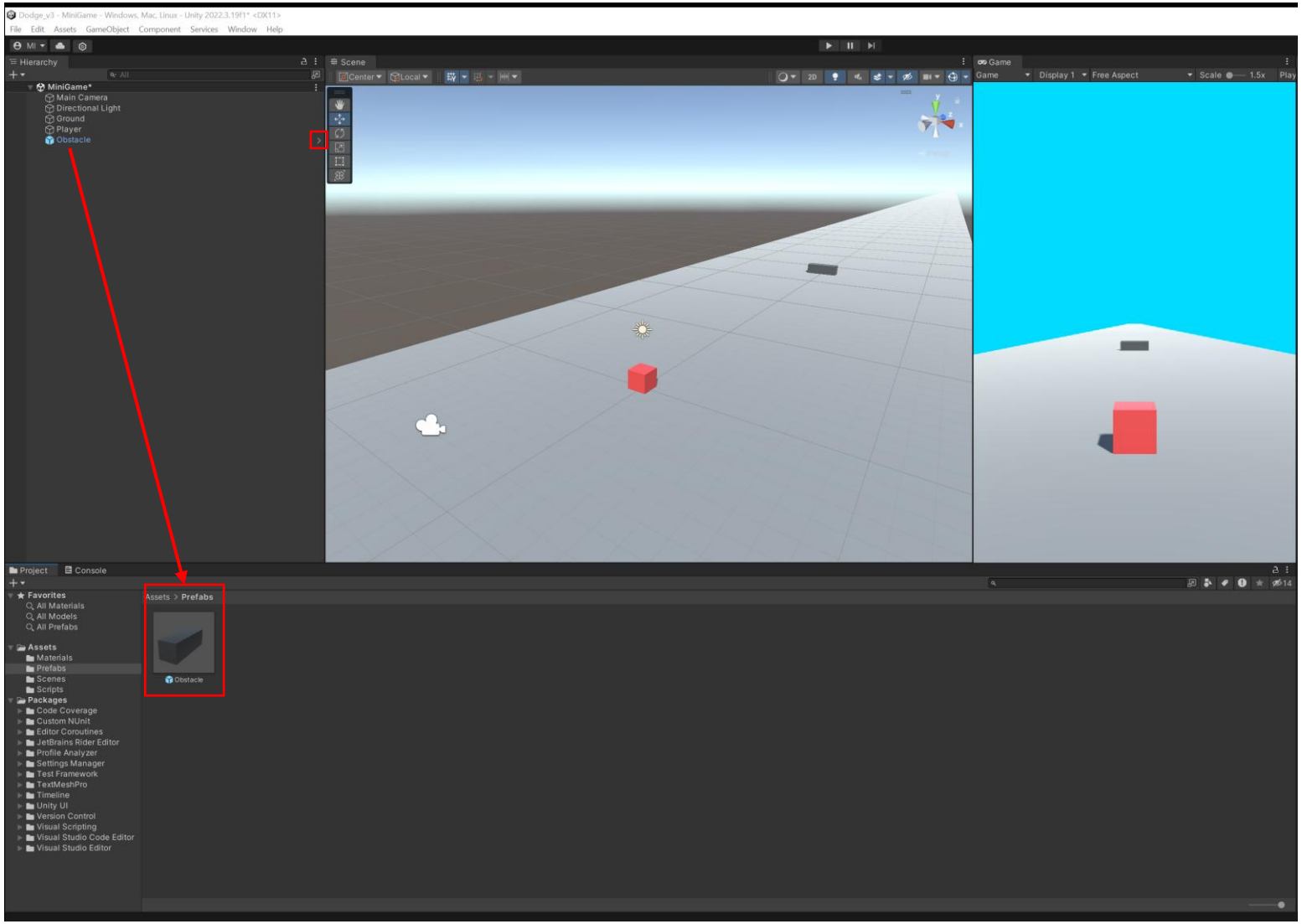
Prefabs: Creating More Obstacle Objects

- **Prefabs** are reusable game object templates

- **Turn the Obstacle GameObject into a prefab.**
  - In the Assets folder, **right-click** > **Create** > **Folder**, then rename this new folder “Prefabs”.
  - Drag the **Obstacle** GameObject from the **Hierarchy** window into the **Prefabs** folder.
  - If prompted, select **Original Prefab**.

- **Enter and exit prefab editing mode.**
  - Select the arrow to the right of the **Obstacle** GameObject in the **Hierarchy** window to open prefab editing mode.
  - To return to the normal **Scene** view, select the back arrow at the top of the **Hierarchy** window.

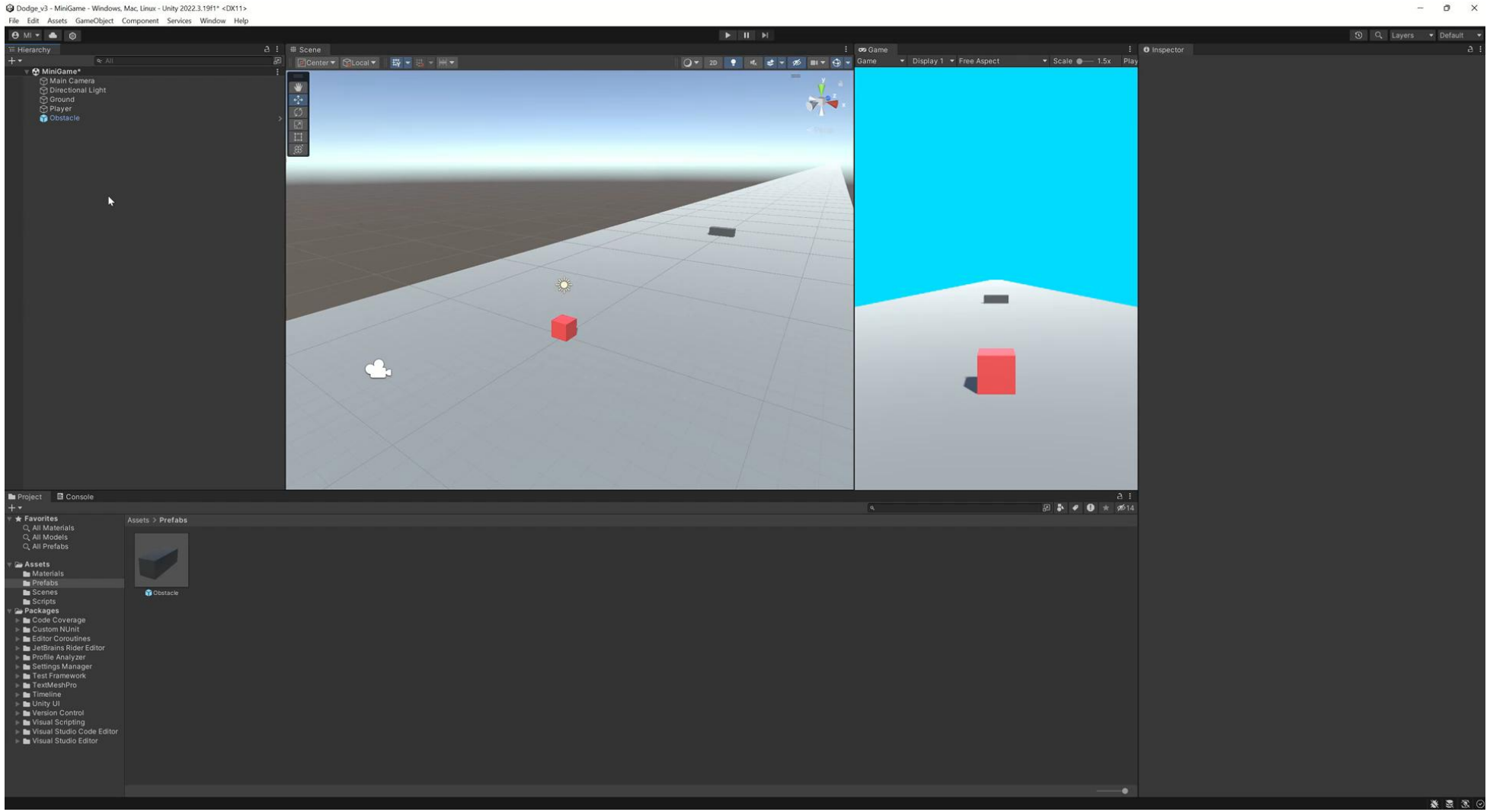
Prefabs: Creating More Obstacle Objects



Prefabs: Creating More Obstacle Objects

- **Make an empty parent GameObject for the Obstacle GameObjects.**
  - In the **Hierarchy** window, create a new empty GameObject and name it “Obstacle Parent”.
  - Reset the **Transform** component.
  - In the **Hierarchy** window, drag the **Obstacle** GameObject onto the **Obstacle Parent** GameObject.

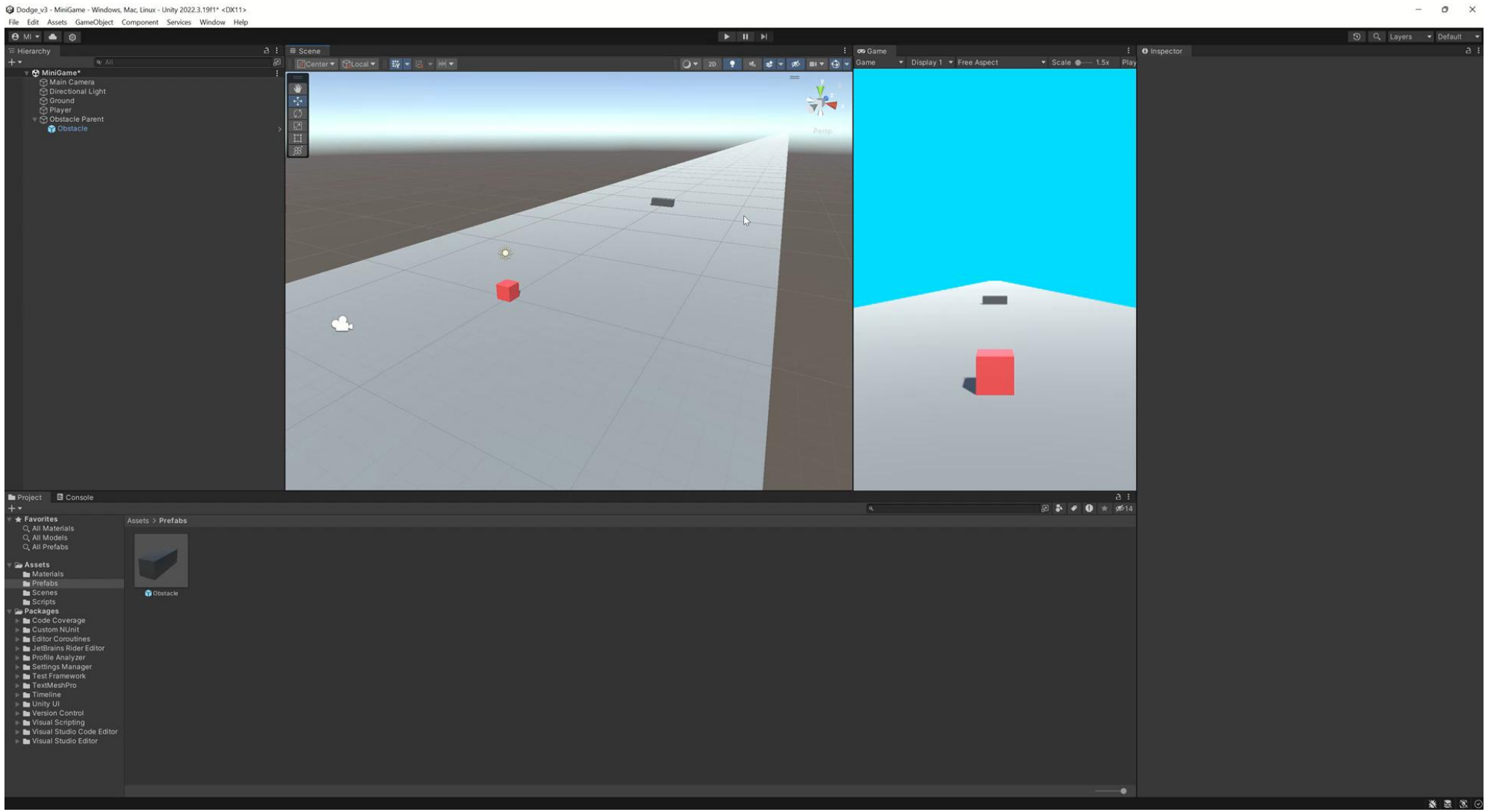
Prefabs: Creating More Obstacle Objects



Prefabs: Creating More Obstacle Objects

- **Align to a top view of the scene.**
  - Select the Gizmo in the upper right of the **Scene** view to switch to a top-down view.
  - Zoom out a little so you can see the entire play area.

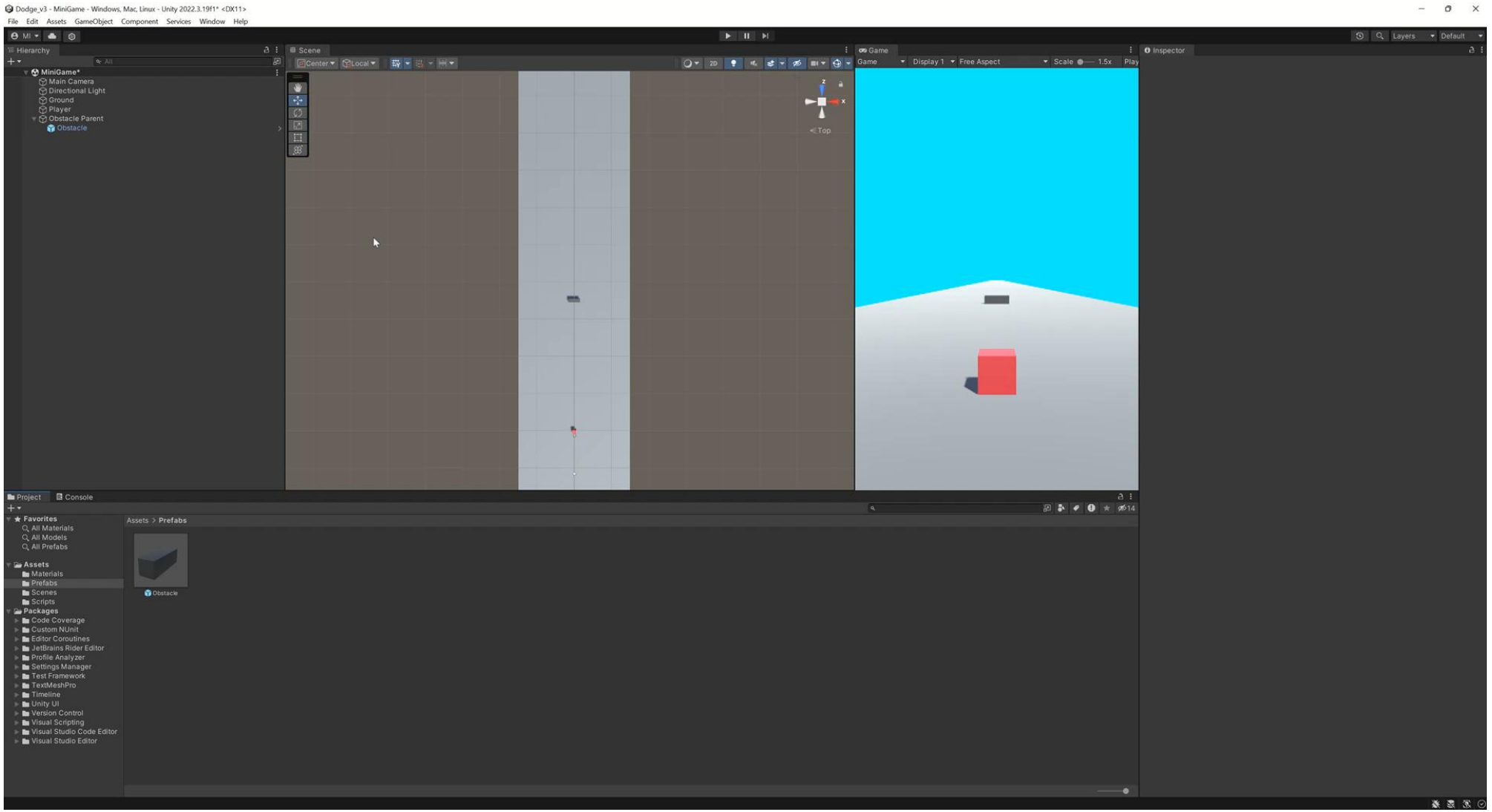
Prefabs: Creating More Obstacle Objects



Prefabs: Creating More Obstacle Objects

- **Place duplicate pickups around the scene.**
  - Move the first **Obstacle** GameObject somewhere you like.
  - With the **Obstacle** GameObject selected, duplicate it with **Ctrl**+**D** (macOS: **Cmd**+**D**).
  - Use the **Move** tool to relocate the second instance of the prefab.
  - Repeat this process to place as many **Obstacle** GameObjects as you like in the scene.

Prefabs: Creating More Obstacle Objects



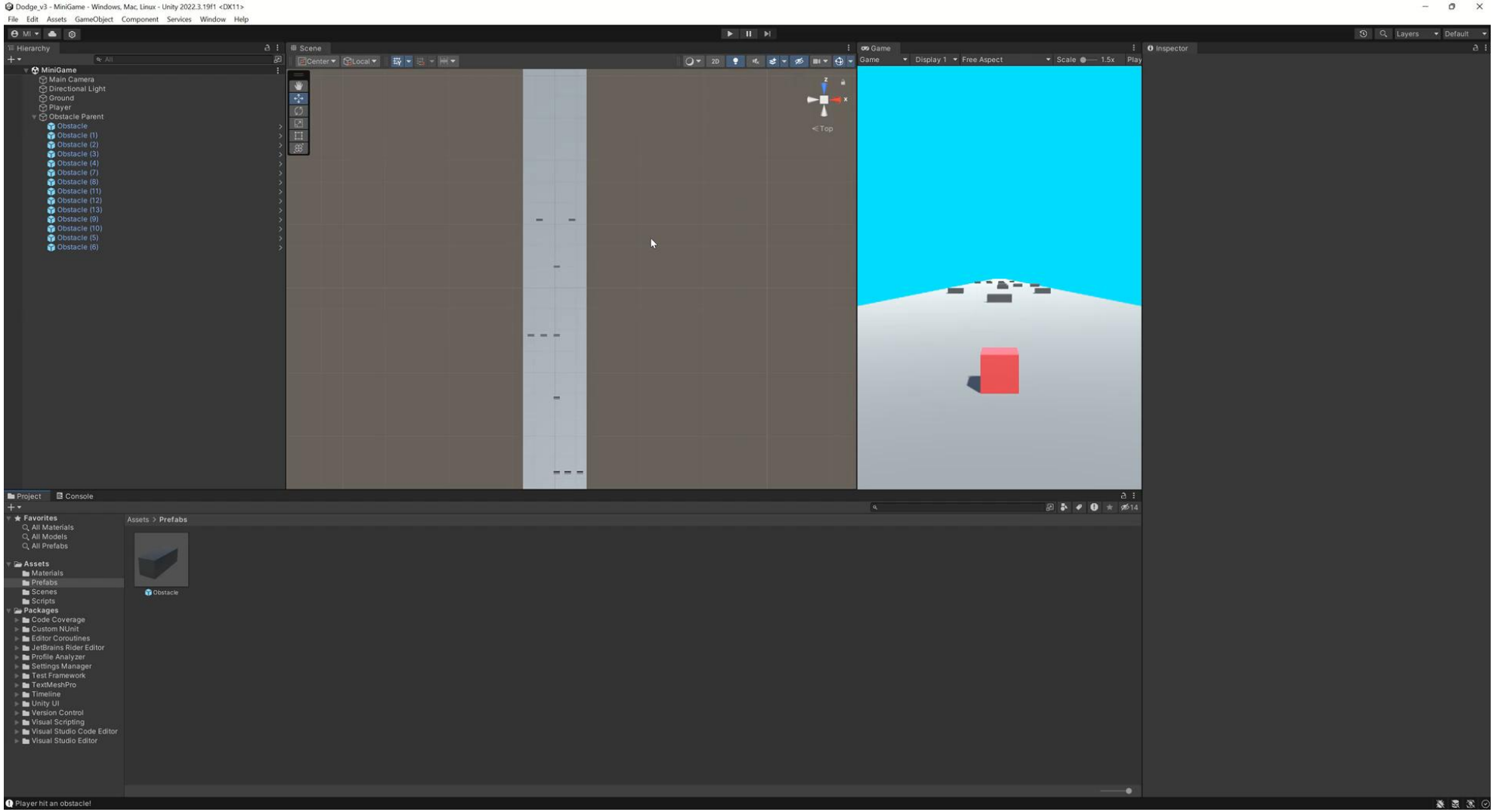
Improving Collision Detection

- You might encounter situations where the player seems to pass through obstacles. Let's explore ways to refine collision detection.

- Continuous Collision Detection:
  - Select the obstacle prefab in the Project panel.
  - In the Inspector window, navigate to the Rigidbody component.
  - Under Collision Detection, change the mode to "Continuous". This instructs Unity to perform collision checks more frequently, potentially reducing instances where the player clips through the obstacle.

- Player Rigidbody Settings:
  - Apply the same "Continuous" Collision Detection mode to the player object's Rigidbody component as well.

Improving Collision Detection

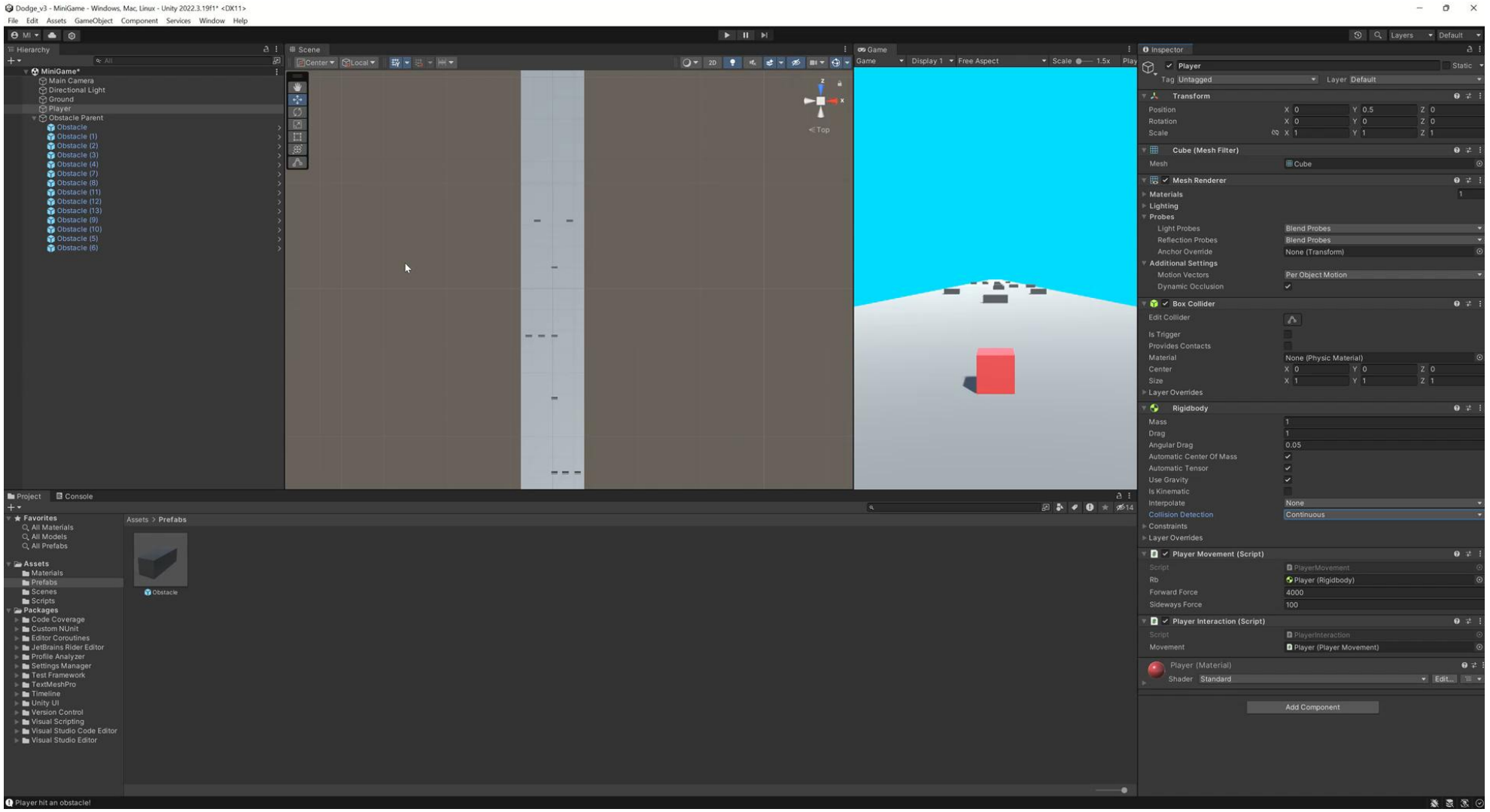


Improving Collision Detection

- The fixed timestep value in Unity's Time settings determines how often physics calculations are performed. A lower fixed timestep can lead to more precise physics simulations, potentially improving collision detection. However, it can also impact performance.

- Adjusting Fixed Timestep:
  - Go to Edit > Project Settings > Time.
  - Locate the Fixed Timestep property and consider setting it to a lower value (e.g., 0.01). (Experiment with this value to find a balance between collision accuracy and performance for your game.)

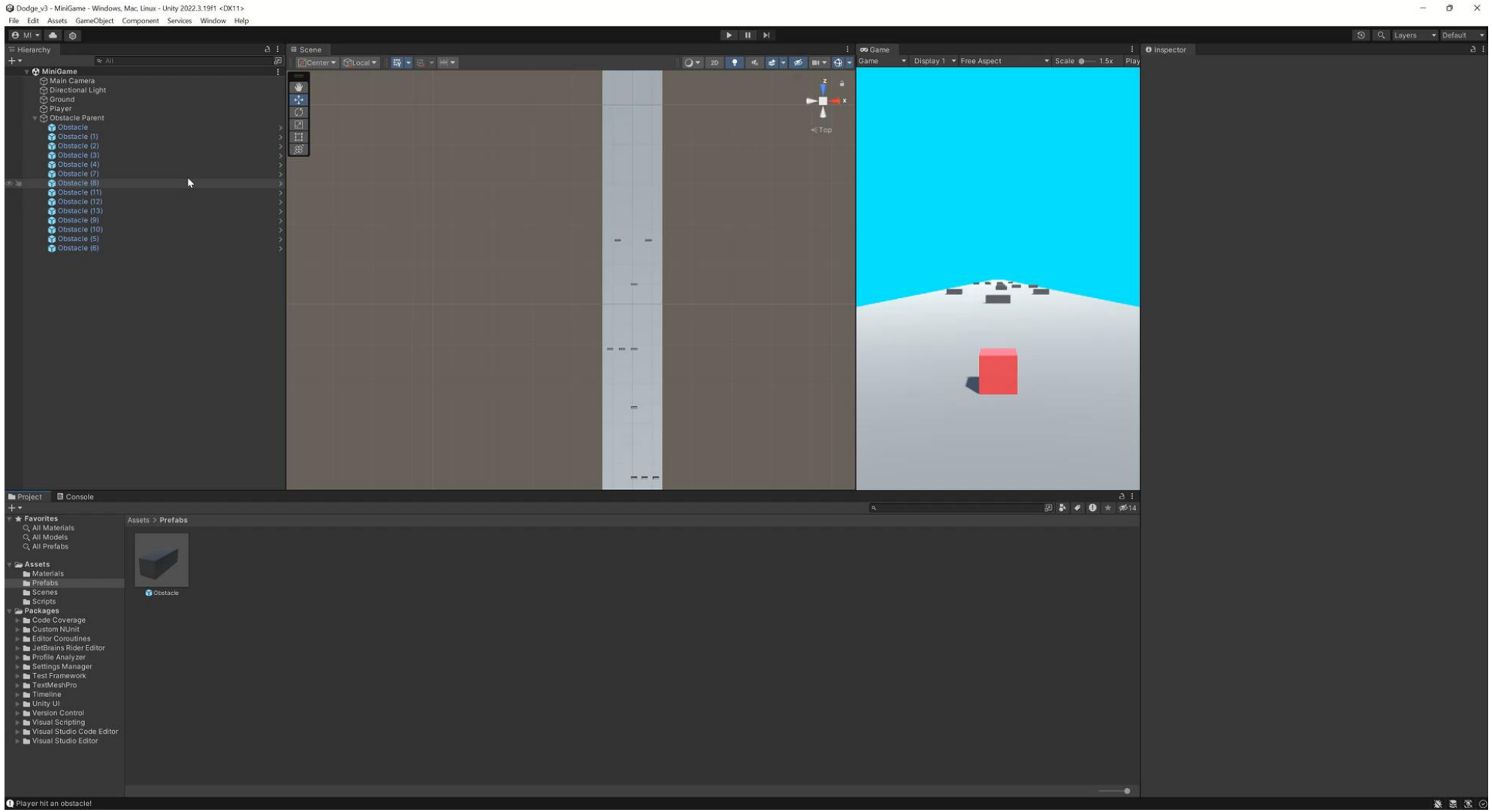
Improving Collision Detection



Improving Collision Detection

- To make the game more challenging, let's increase the forward force to 8000.
- Increasing the sideways force a bit (e.g., 120) may make the movement smoother.

Improving Collision Detection



Section

Displaying Text

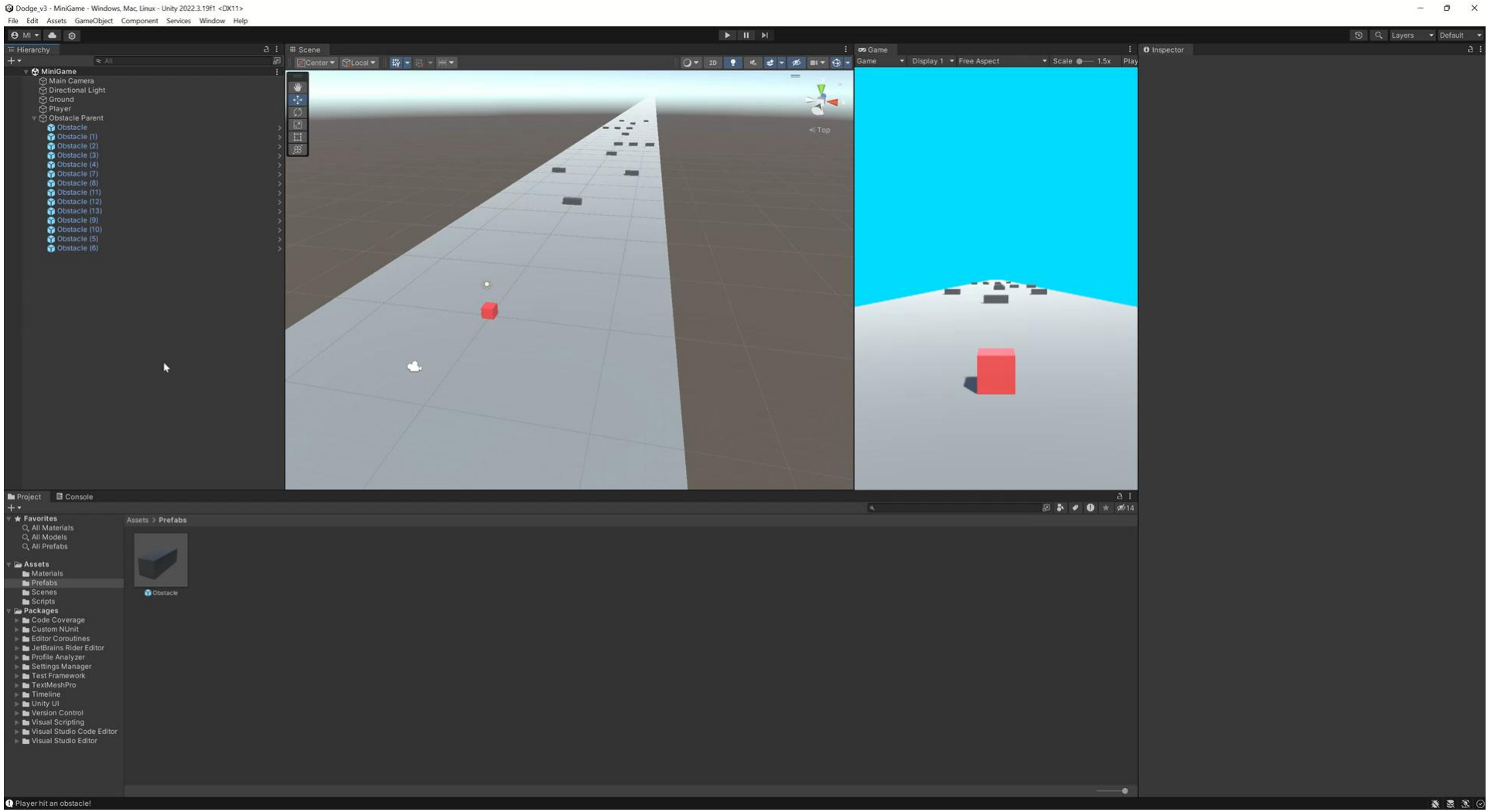
Displaying Text

- Now that we have a basic game structure, let's add a text element to display.
- Let’s say we want to display the text “**Dodge the obstacles**” at the top of the game when the game starts.
- When the player hits an obstacle Unity should display “**You lose!**” in place of “Dodge the obstacles”.

Creating a UI Text Object

- **Add a Text object named “DisplayText”.**
  - In the Hierarchy window, right-click and select UI > Text (Legacy). This creates a new text object in your scene.
  - Rename the **Text (Legacy)** GameObject “DisplayText”.

Creating a UI Text Object

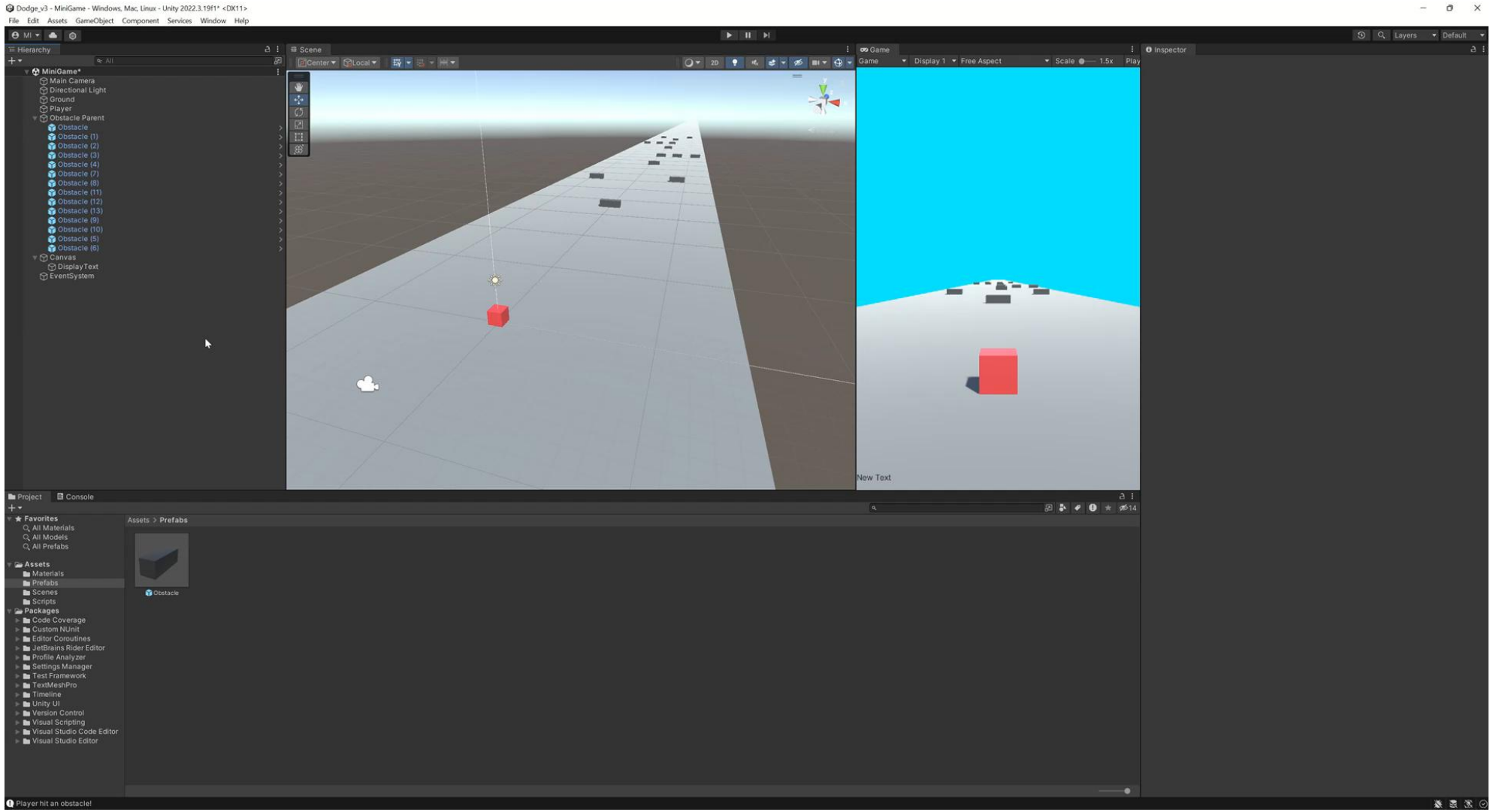


Creating a UI Text Object

- **Preview the full canvas in 2D view.**
  - Select the **Canvas** GameObject and press the **F** key to frame the entire GameObject in the **Scene** view.
  - Select the 2D toggle at the top of the **Scene** view to change to 2D view.

- **Edit the text.**
  - Select the **DisplayText** GameObject.
  - In the **Text** box, delete “New Text” and replace it with “Dodge the obstacles” as the default text (placeholder).

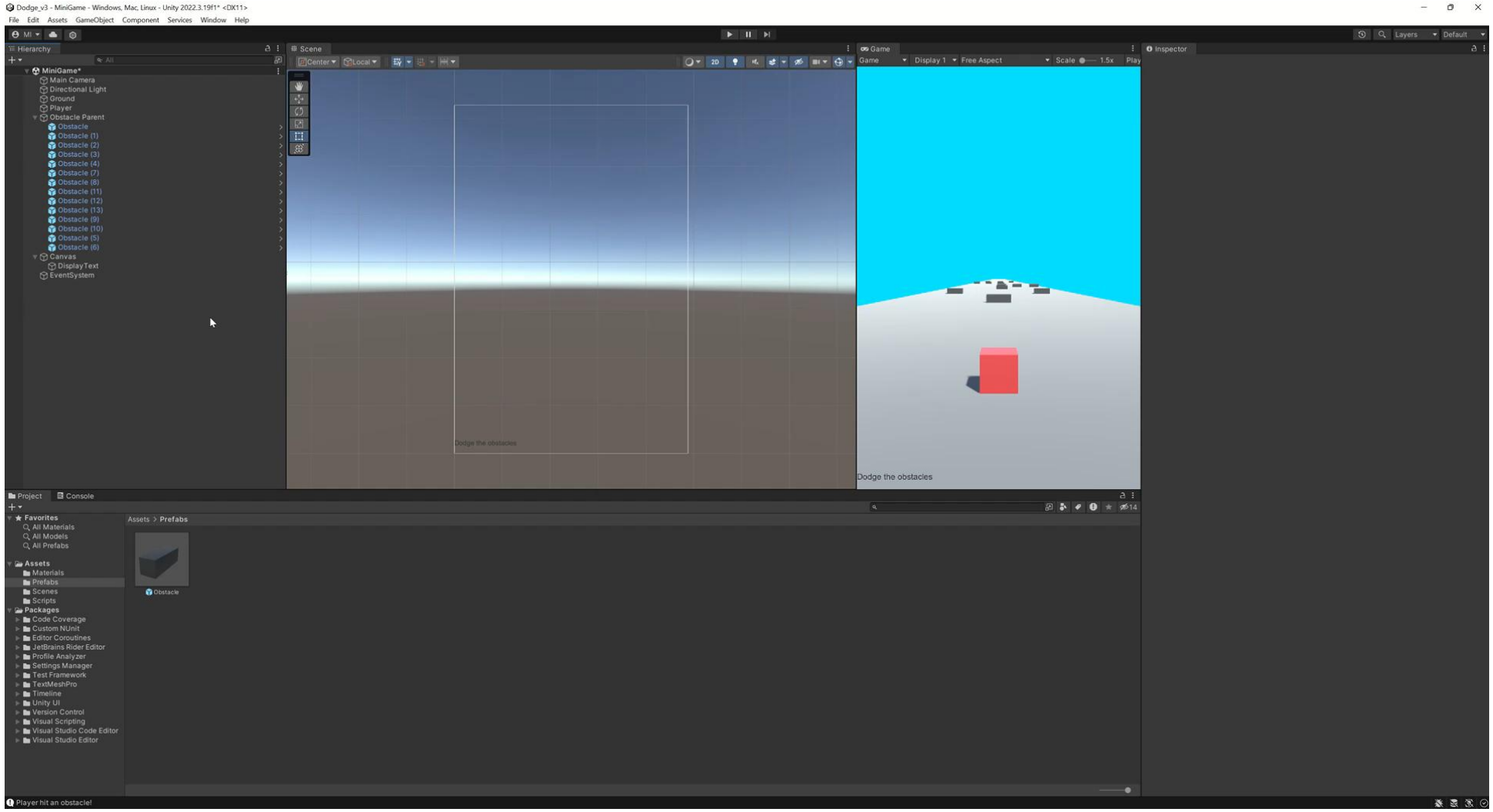
Creating a UI Text Object



Creating a UI Text Object

- **Modify the position and size of the text.**
  - Using the mouse, move the text in the Scene view to the upper center.
  - Change text alignment to Center on both the horizontal and vertical in the Inspector panel.
  - Set the font size to a larger value (such as 36)
  - Make sure you have enough space for the text by modifying the text boundary in the Scene view using the mouse (holding the Alt button on the keyboard).
  - Set Horizontal Overflow to Overflow
  - Select Canvas in Hierarchy -> (Inspector panel) Set UI Scale Mode: **Scale with screen size** and Match: **1**

Creating a UI Text Object



Changing Text on Collision

- We want the DisplayText to be set to “You lose!” when collision occurs (instead of printing “Player hit an obstacle” in the Console).

- **Declare a new text variable.**
  - Open the **PlayerInteraction** script in your script editor.
  - Add the following new line of code underneath the movement variable:

```
public Text displayText;
```

- **Update OnCollisionEnter**
  - In the OnCollisionEnter function replace **`Debug.Log("Player hit an obstacle!");`** with the following line:

```
displayText.text = "You Lose!";
```

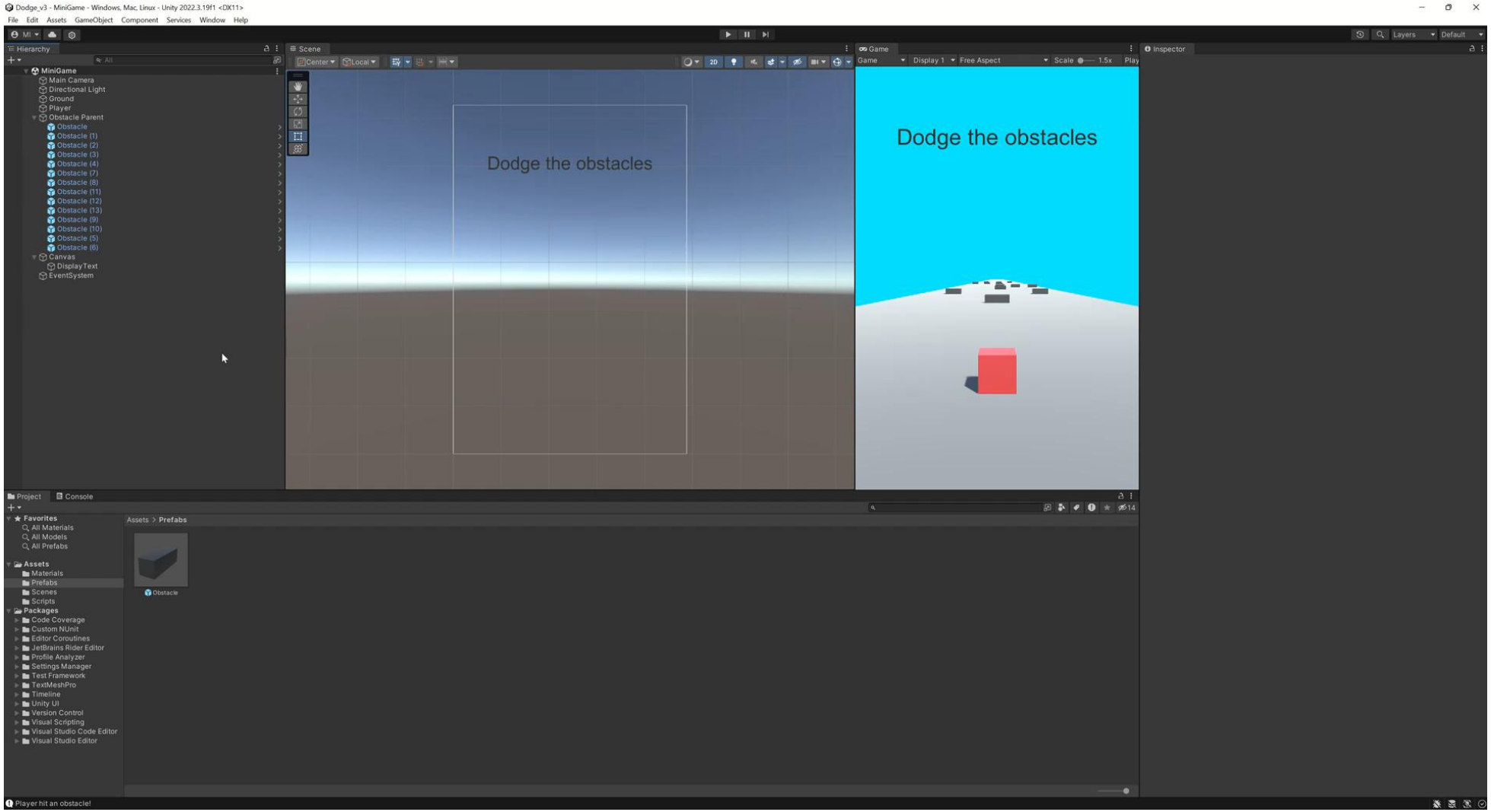
  - Save the script (Ctrl+S / Cmd+S).

Changing Text on Collision

- **Assign the DisplayText variable in the Inspector window.**
  - Select the Player GameObject in the Hierarchy window, then drag the **DisplayText** GameObject into the **Display Text** slot to reference the UI text element.
  - **(Important:** This step is very important and easy to miss. If you do not do this step, you will see a **NullReferenceException** error in the Console window and your game will not work.)

- **Test your game.**

Changing Text on Collision



Section

Implementing a Game Over State

Conditions for Win or Lose

- Now that we know how to add texts to our game, let's finish defining conditions for ending the game.

  - Player **loses** when
    - It collides with an obstacle (we have already taken care of it)
    - It falls off the ground

  - Player **wins** when
    - It reaches a certain finish point without any collision or falling off the ground

Detecting When the Player Falls off the Ground

- The y position indicates whether the player in on the ground or not.

- **Declare a new transform variable (to keep track of player's y position).**
  - Open the **PlayerInteraction** script in your script editor.
  - Add the following new line of code underneath the displayText variable:

```
public Transform player;
```

- **Add a condition in Update function**
  - Add the following lines inside the Update function:

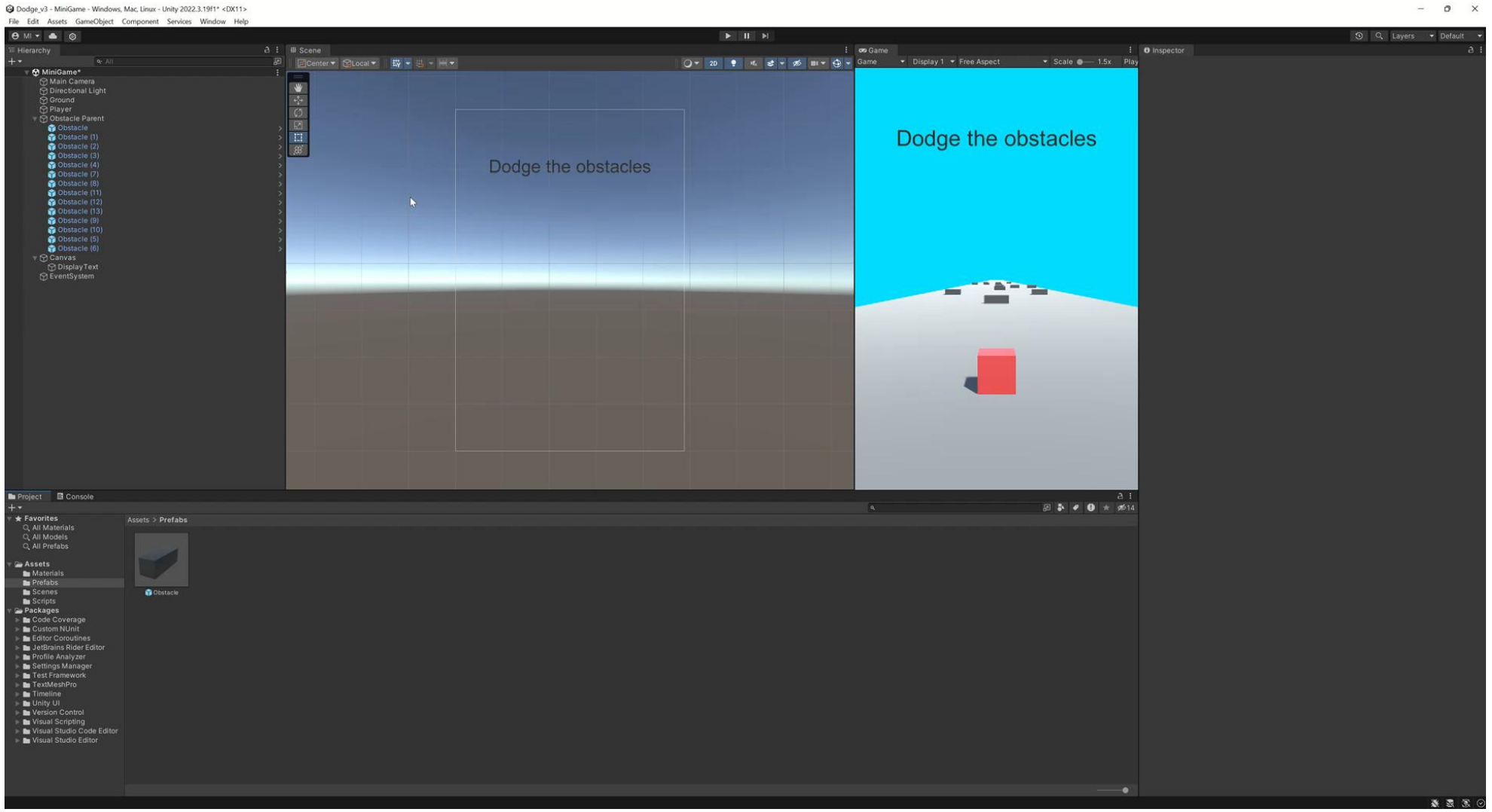
```
if (player.position.y < -1f)
{
    displayText.text = "You Lose!";
    movement.enabled = false;
}
```

Detecting When the Player Falls off the Ground

- **Assign the player variable in the Inspector window.**
  - Select the Player GameObject in the Hierarchy window, then drag the **Player** GameObject into the **Player** slot in the Player Interaction component (Inspector panel) to reference the Player's transform component (for checking its position in the script).

- **Test your game.**

Detecting When the Player Falls off the Ground



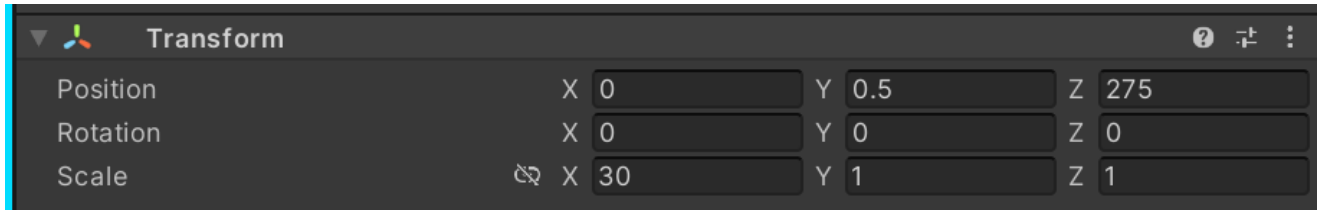
Setting Up Win Condition

- Let's say the player wins when it reaches a finish line without any collision or falling off the ground.

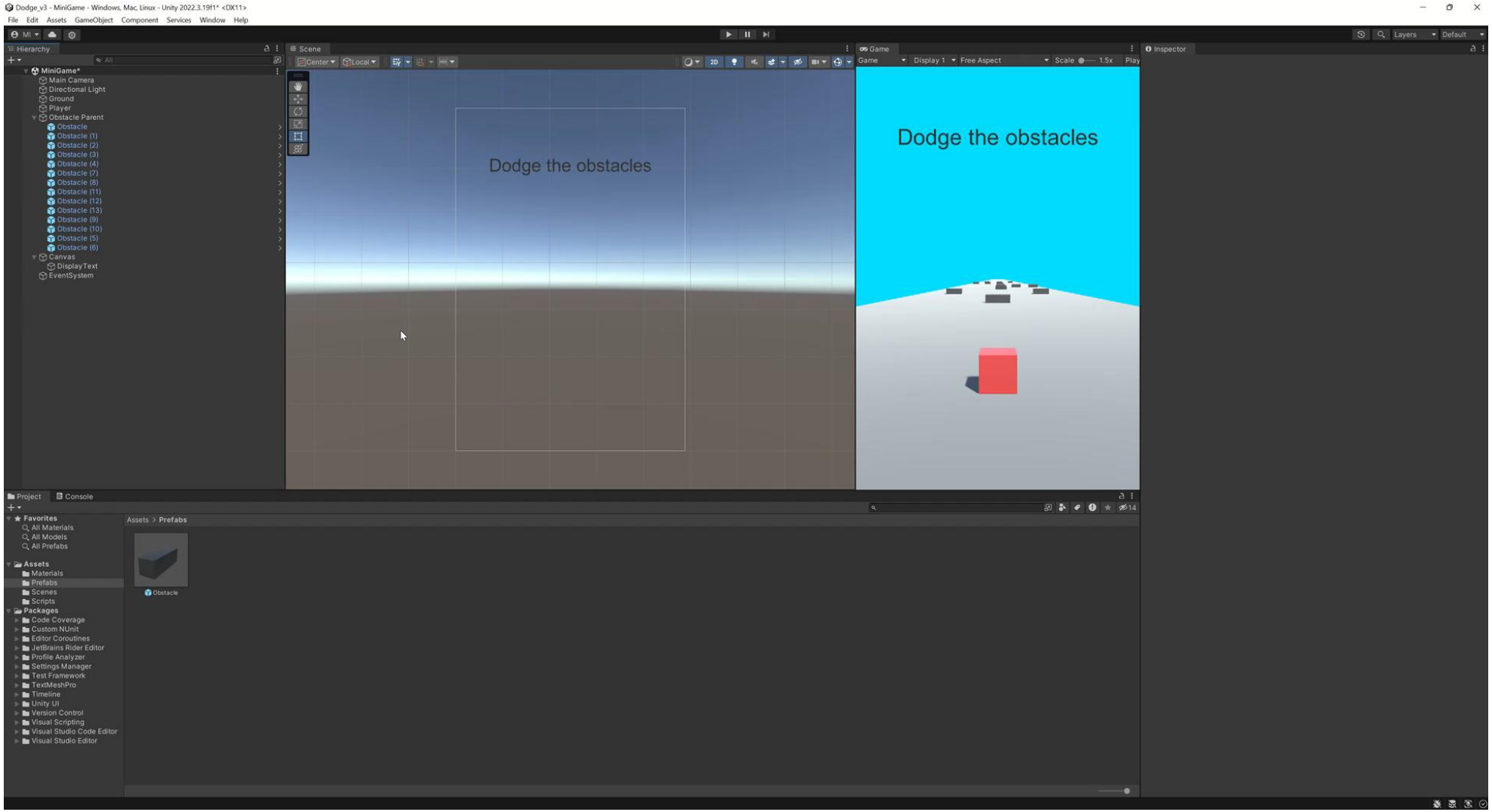
- We can create and horizontal bar shaped object at the end of the obstacles which will act as the finish line. When the Player touches the bar, we will make the bar disappear the display that the Player has won.

Setting Up Win Condition

- **Create a finish line bar.**
  - In the **Hierarchy** window, select the **Add** menu (**+**) > **3D Object** > **Cube**.
  - At the top of the **Inspector** window, rename the newly created **Cube** GameObject “FinishLine".
  - Reset the position of the cube.
  - Then the transform values as shown below (**Note:** you may need to set the Z position value to a smaller or larger number based on where you places the obstacles):



Setting Up Win Condition

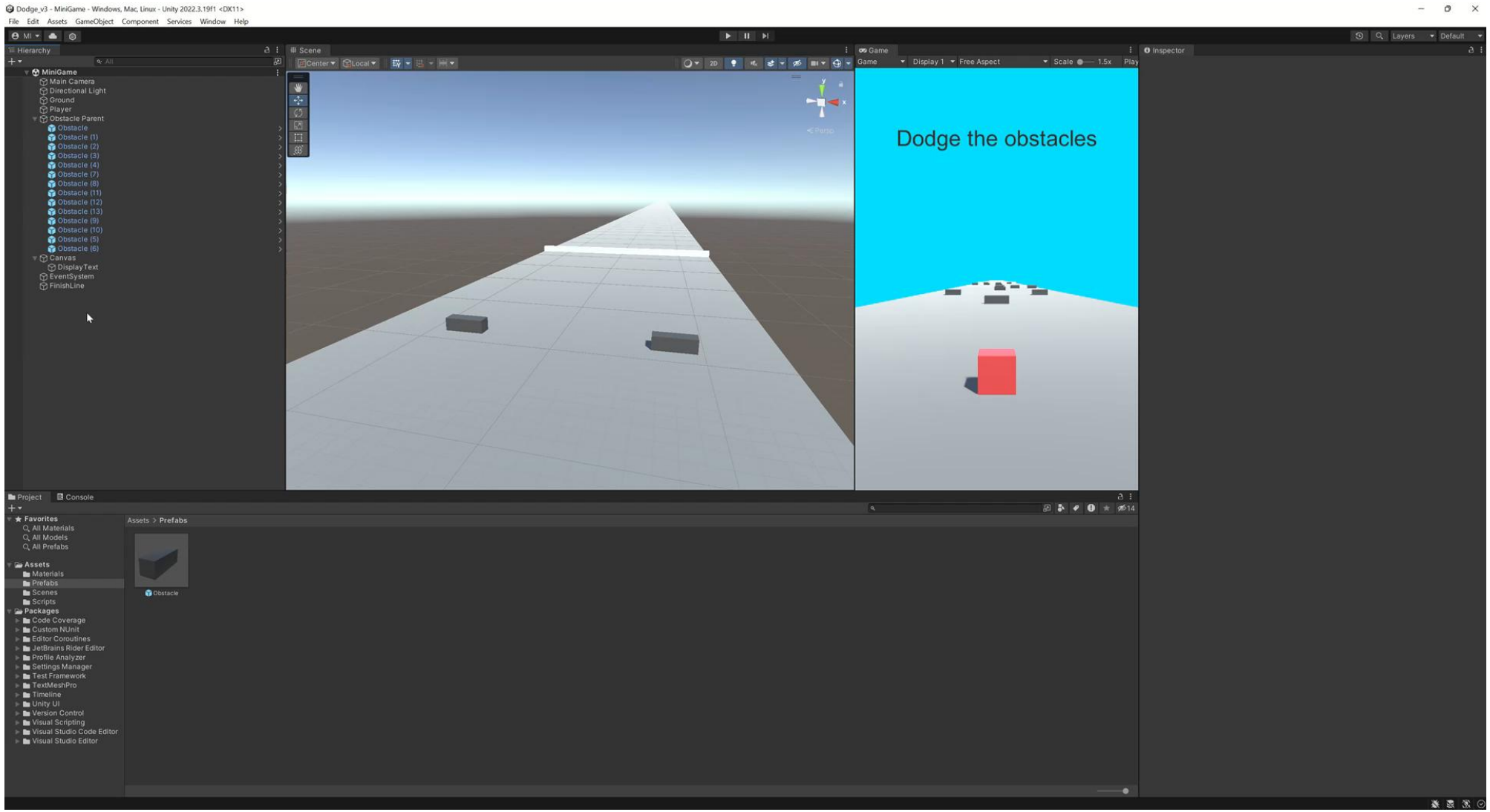


Setting Up Win Condition

- **Create a tag (called “finish”) for the finish line object like we did for the obstacle object.**

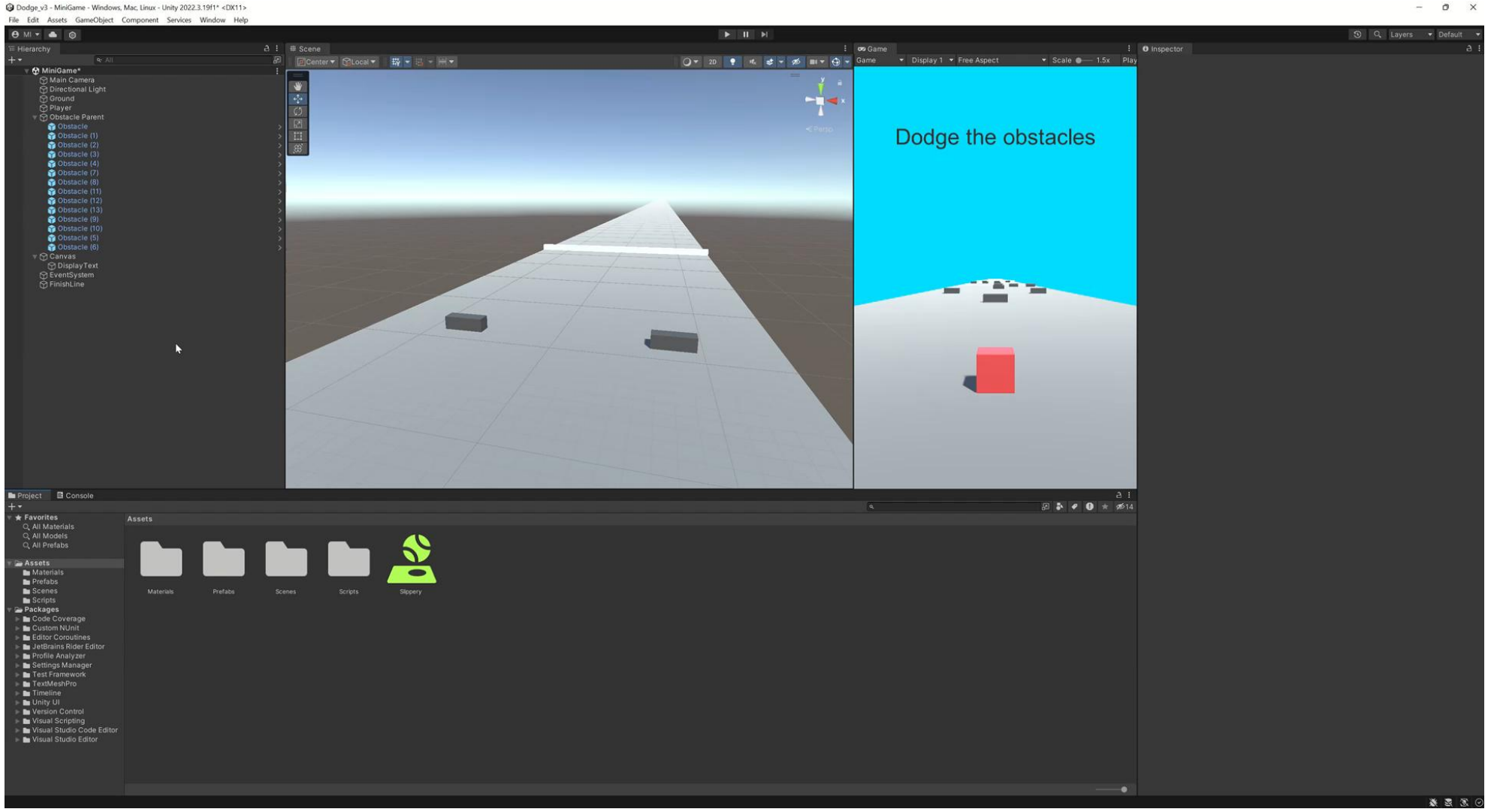
- **Enable Is Trigger for FinishLine object.**
  - Select the FinishLine object in the Hierarchy window.
  - Check the Is Trigger box underneath the Box Collider component in the Inspector window.

Setting Up Win Condition



Setting Up Win Condition

After creating the “finish” tag, make sure to assign the tag to FinishLine object.



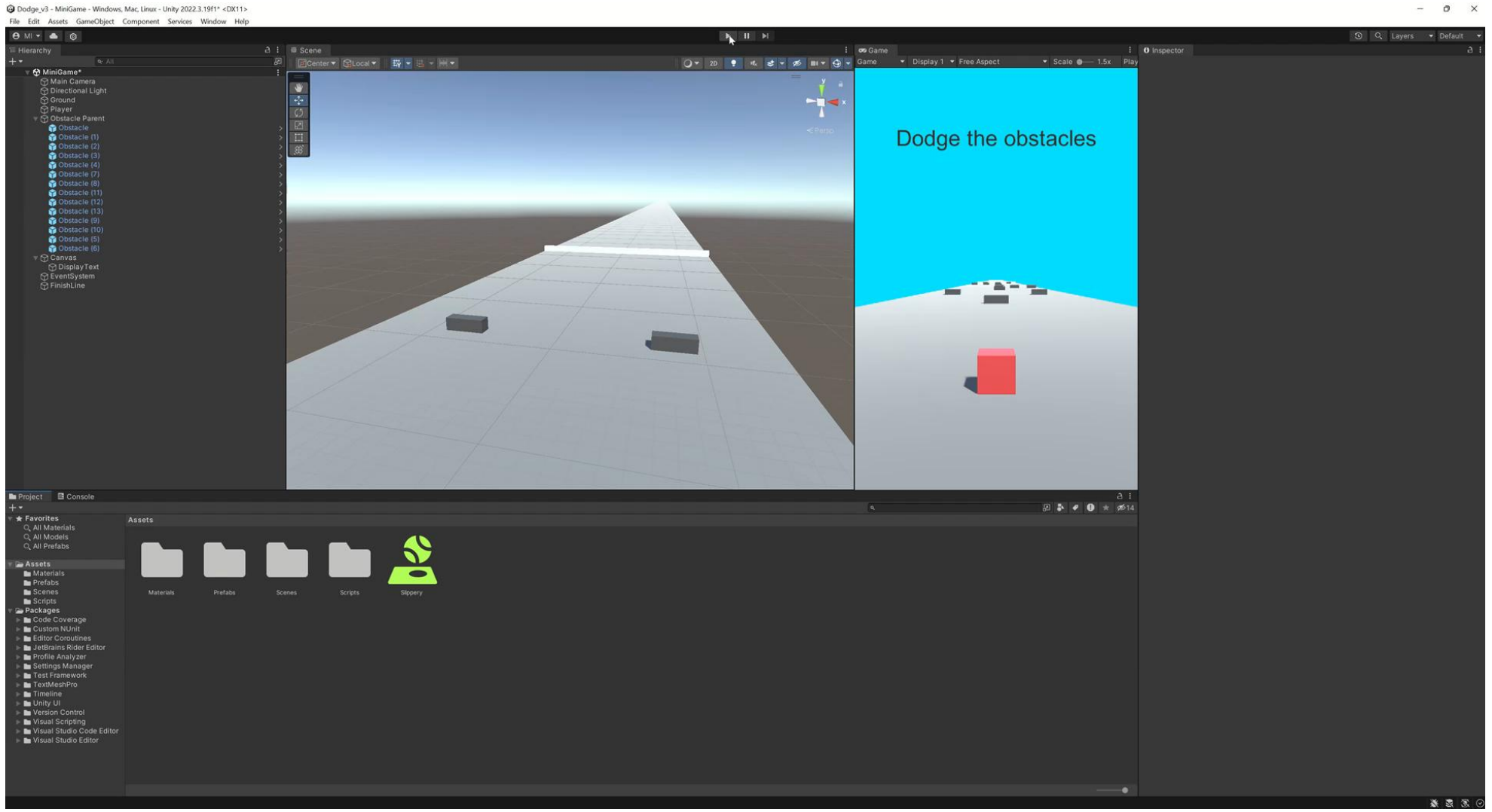
Setting Up Win Condition

- **Add OnTriggerEnter function.**
  - Open the PlayerInteraction script.
  - Within the first curly braces, below the Update function, add the following lines:

```
private void OnTriggerEnter(Collider other) {
    if (other.gameObject.CompareTag("finish"))
    {
        displayText.text = "You Win!";
        movement.enabled = false;
        other.gameObject.SetActive(false);
    }
}
```

- **Save the script.**
- **Test the game.**

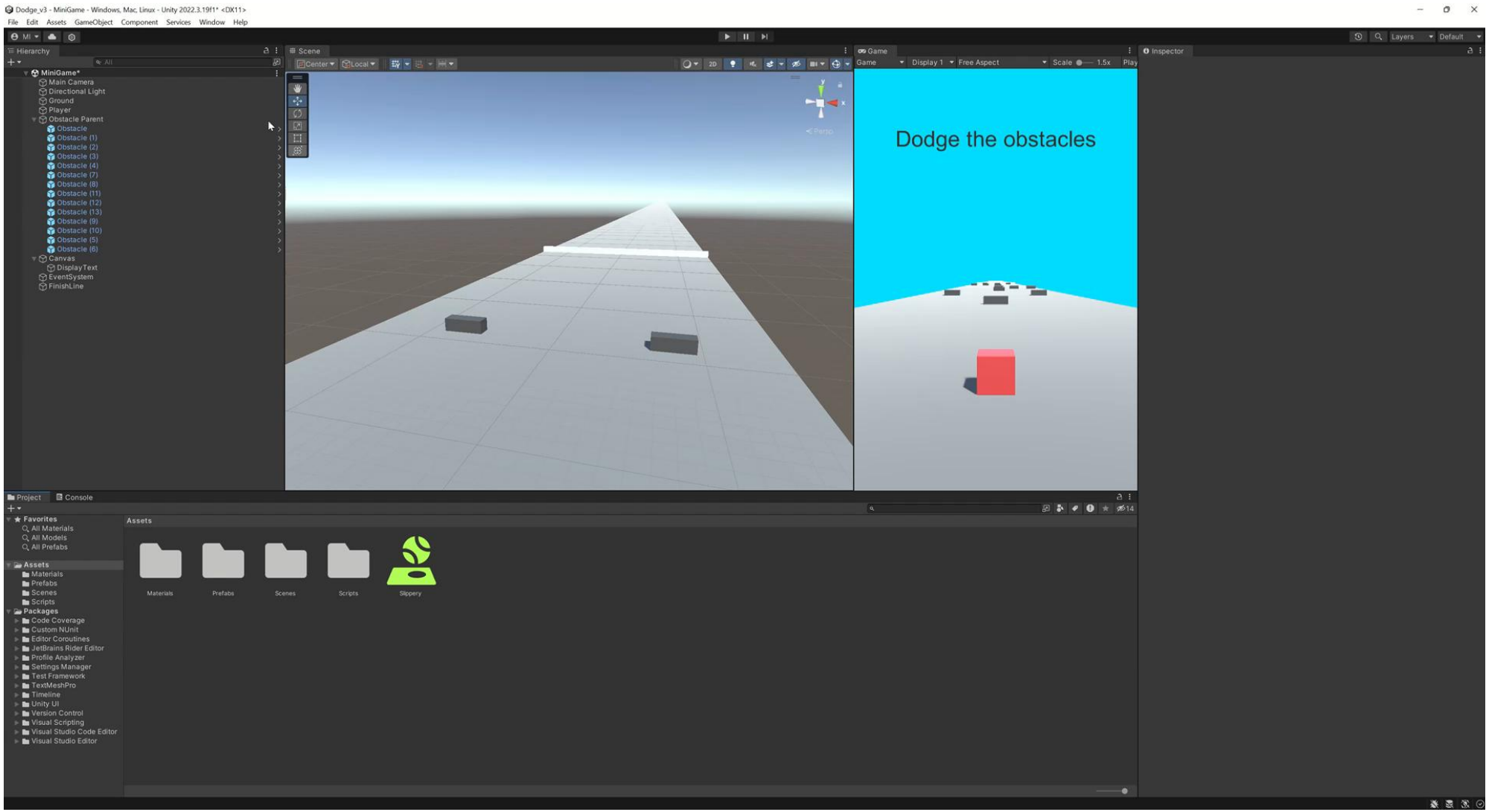
Setting Up Win Condition



Adding Fog

- **Adding Fog for Atmosphere (Optional)**
  - To enhance the visual appeal of the scene, we can add fog so that the Player will not be able to see all the obstacles at once.
  - Window → Rendering → Lighting
  - Select the Environment tab.
  - Check the Fog box.
  - Select the color to be the light blue color we picked for the sky.
  - Also set the Fog's density to 0.02.

Adding Fog


Dodge_v3 - MiniGame - Windows, Mac, Linux - Unity 2022.3.19f1* <DX11>
File Edit Assets GameObject Component Services Window Help
Hierarchy
MiniGame*
Main Camera
Directional Light
Ground
Player
Obstacle Parent
Obstacle
Obstacle (1)
Obstacle (2)
Obstacle (3)
Obstacle (4)
Obstacle (7)
Obstacle (8)
Obstacle (11)
Obstacle (12)
Obstacle (13)
Obstacle (9)
Obstacle (10)
Obstacle (5)
Obstacle (6)
Canvas
DisplayText
EventSystem
FinishLine
Scene
Game
Display 1
Free Aspect
Scale
1.5x
Inspector
Layers
Default
Dodge the obstacles
Project
Console
Favorites
All Materials
All Models
All Prefabs
Assets
Materials
Prefabs
Scenes
Scripts
Packages
Code Coverage
Custom NUnit
Editor Coroutines
JetBrains Rider Editor
Profile Analyzer
Settings Manager
Test Framework
TextMeshPro
Timeline
Unity UI
Version Control
Visual Scripting
Visual Studio Code Editor
Visual Studio Editor
Assets
Materials
Prefabs
Scenes
Scripts
Slippery

Section

Building the Game

Building the Game

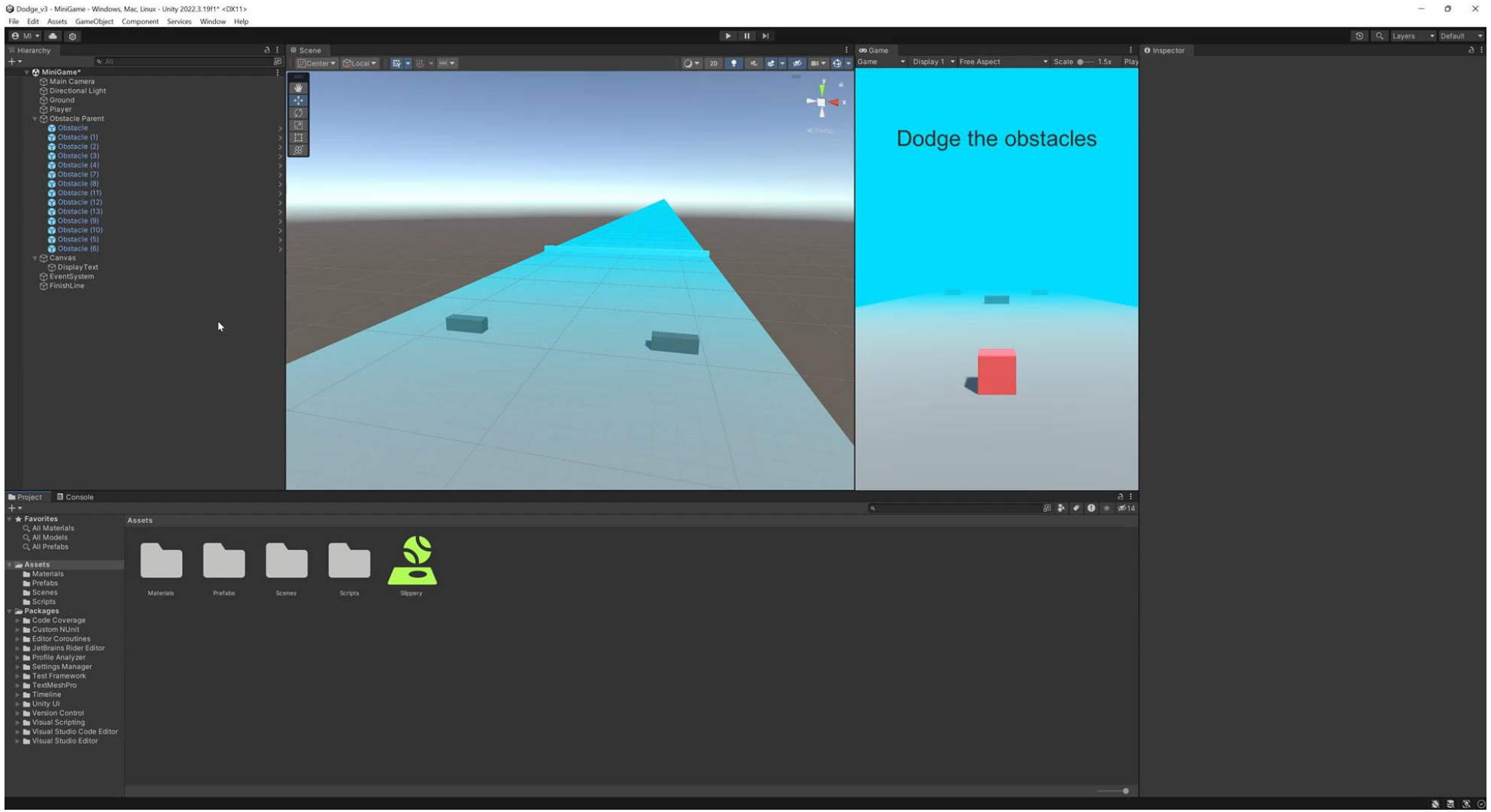
- We’ve completed making our game. Although it is a avery basic game, hopefully you’ve learnt fundamental concepts for game development in Unity.

- Now let’s build/export our game.

Building the Game

- **Configure your build settings.**
  - Save your scene.
  - To open the **Build Settings** window, from the main menu go to **File** > **Build Settings**. Alternatively, you can press **Shift**+**Control**+**B** (macOS: **Shit**+**Cmd**+**B**).
  - Select **Windows, Mac, Linux** in the **Target Platform** box,
  - In **Scenes in Build**, select **Add Open Scenes** to add your Roll-a-ball game to the build. You can also drag scenes from the **Project** window to this list.
  - If **SampleScene** is showing in the list, disable it to exclude it from your build.

Building the Game



Building the Game

- **Configure your player settings.**
  - Select **Player Settings** to open a range of different configuration options for the **Game** view.
  - If you want, change the **Company Name**, **Product Name**, and **Version**.
  - Select **Resolution and Presentation** and change the **Fullscreen Mode** box to **Windowed**.
  - By default, this will have a specific resolution but set the default width and height to something smaller if your screen uses a lower resolution.
  - Close the **Project Settings** window and return to the **Build Settings** window.

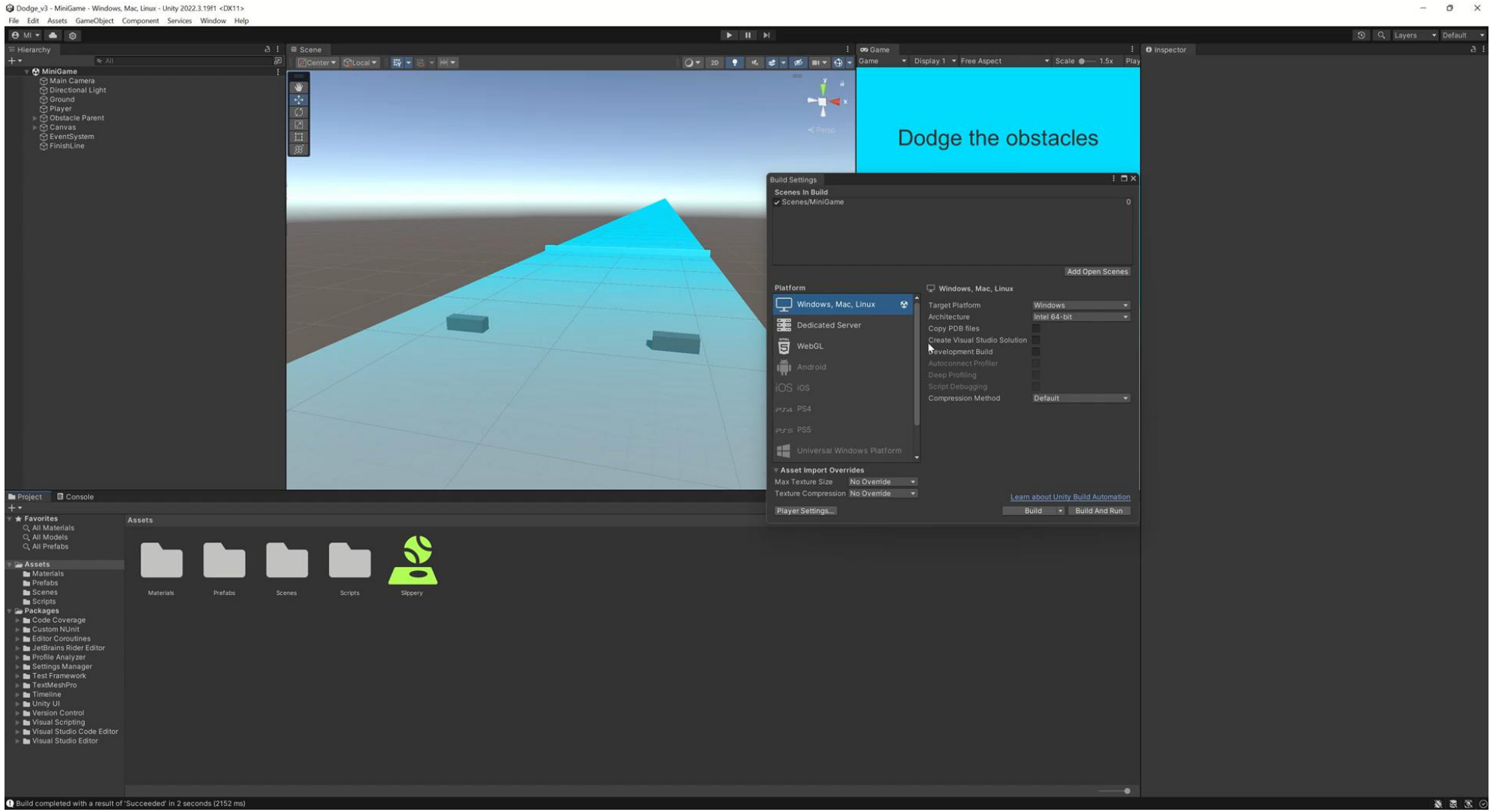
Building the Game

- **Build your game.**
  - Select **Build**. This will bring up a dialog that asks you to choose a build location.
  - To keep things tidy, create a new folder inside your project called “Builds” at the root of the project, alongside the **Assets** and **Library** folders.
  - If you’re using macOS, you can also choose a name for your build.
  - Confirm that you want to **Select Folder** (Windows) or **Save** (macOS). Unity will now build the application and save it to the **Builds** folder.

Building the Game

- **Play your standalone game!**
  - Navigate to the **Builds** folder and then run the executable application.

Building the Game



Acknowledgement

- The contents of these slides have been adopted primarily from the following two sources:
  - https://www.youtube.com/playlist?list=PLPV2KyIb3jR53Jce9hP7G5xC4O9AgnOuL
  - https://learn.unity.com/project/roll-a-ball