Unity User Manual
2D Game Development

Written By: Jacob Drake

Table of Contents

Welcome to Unity

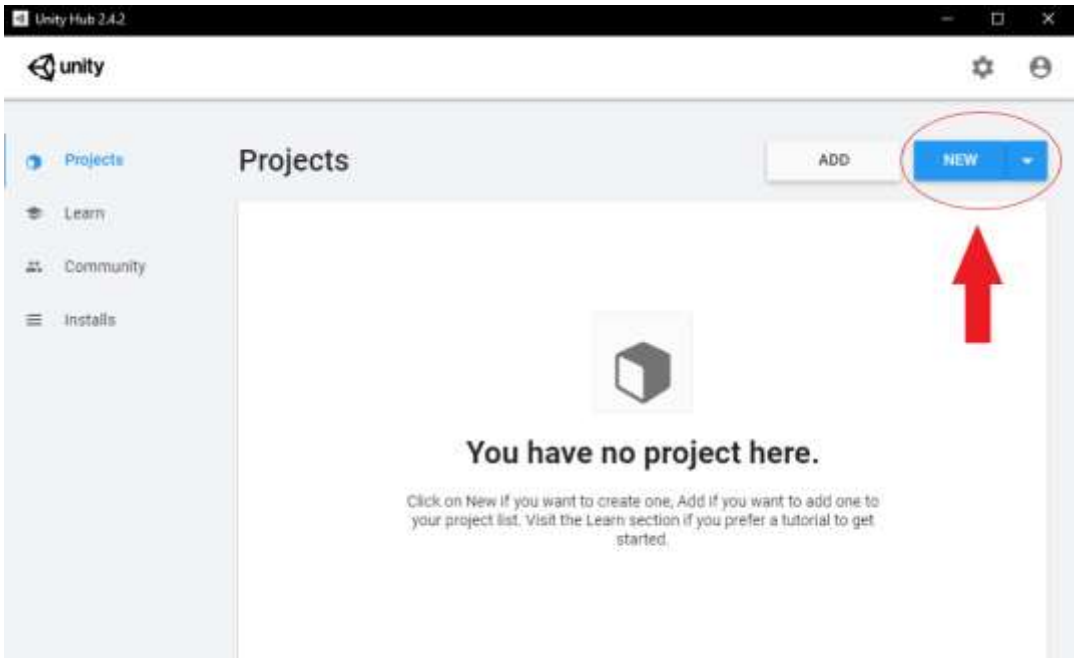
Unity or Unity Real-Time Development Platform is a cross-platform game engine made by Unity Technologies. This manual will serve as a beginner's guide to Unity and its different functionalities. While Unity's official user-manual is a great tool, it is not sufficient for learning the process of making a video game. Through this manual, you will learn how to create and manage new projects, navigate through Unity's editor, learn its various components, make game objects, write scripts, import assets, and other aspects for making a video game. You will gain experience in Unity by following this manual's step by step guide on how to make a top-down shooter with a player-controlled main character, a map, enemies, and basic combat system. Download Unity Hub from Unity's official website to begin using this manual: https://unity3d.com/get-unity/download.

Create a Project

1. Start Unity Hub and Make a New Unity Project

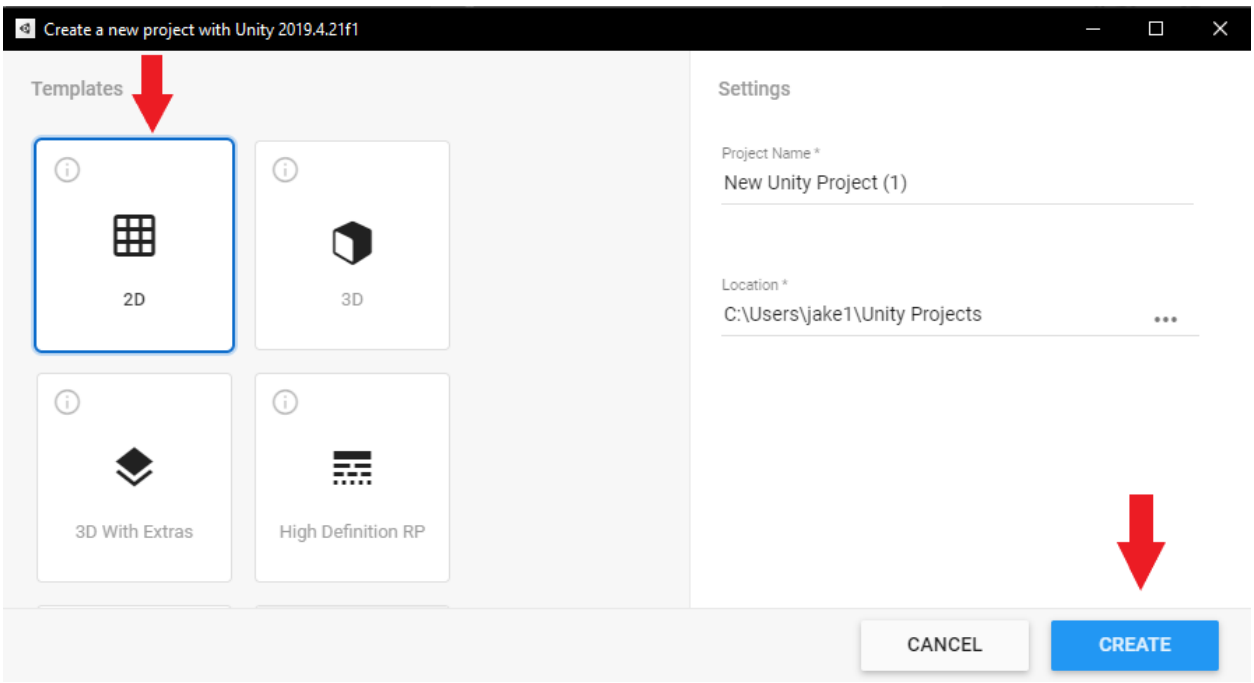
a. Double-click the Unity Hub icon on your desktop to start Unity Hub.

b. Create a new project in Unity by clicking “New” under the “Projects” tab in Unity Hub.

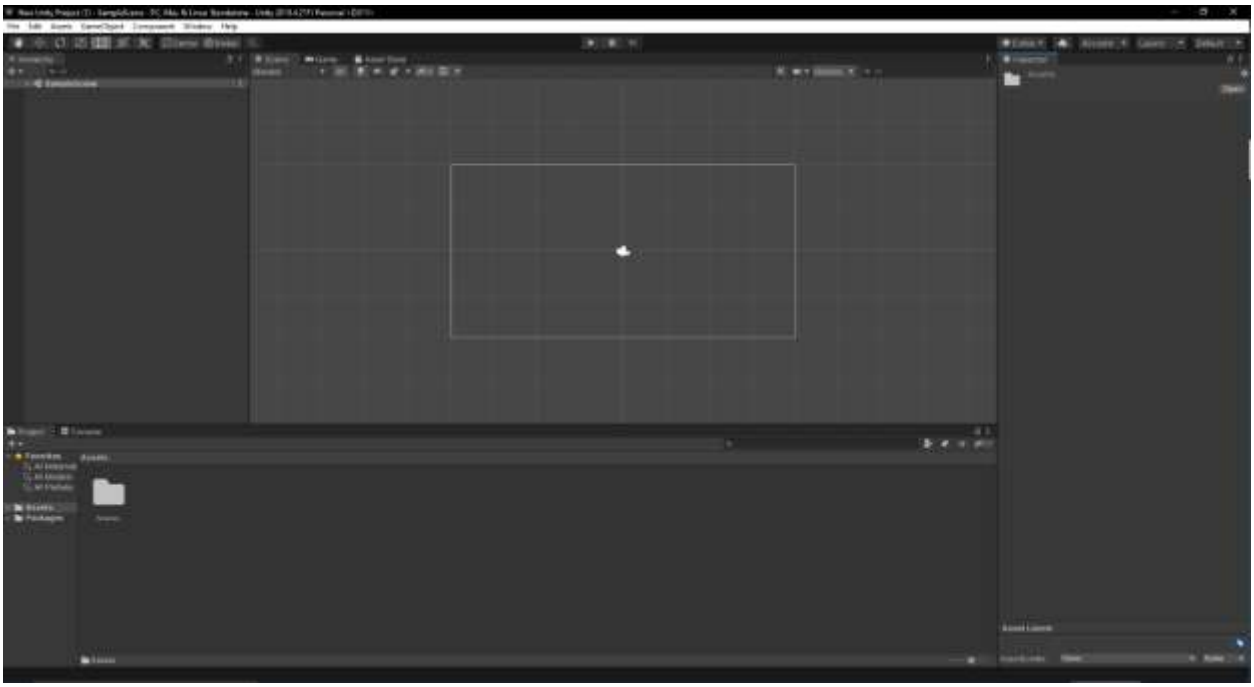


i. This will open a new window with different templates to choose from.

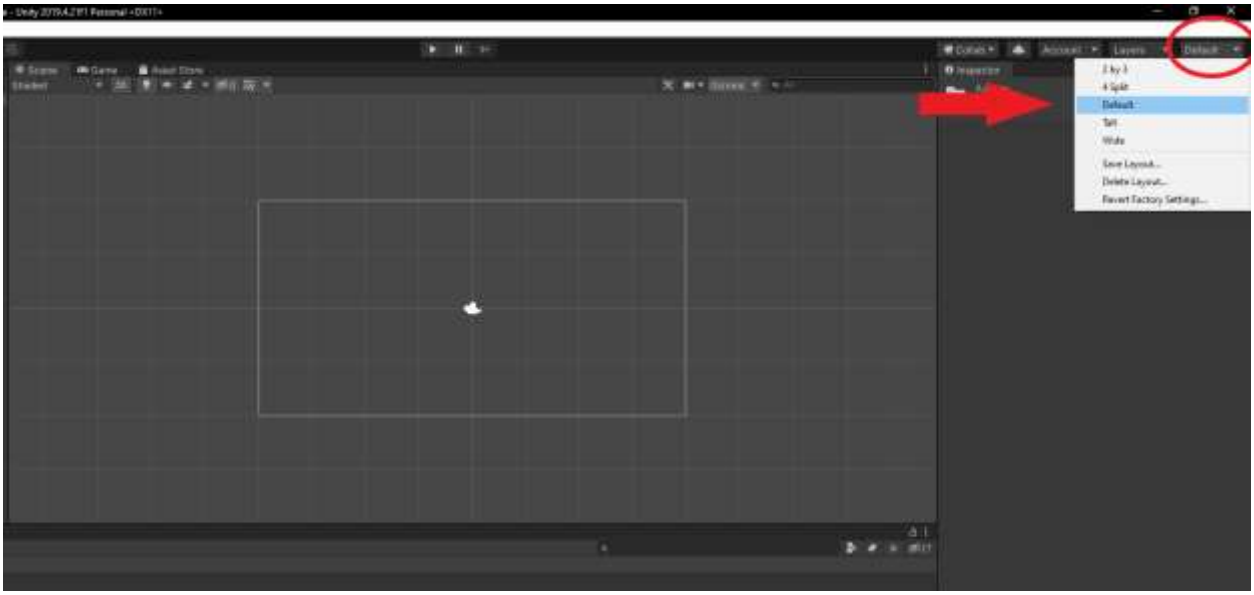
c. Select the 2D template and the click “Create” in the bottom-right corner.



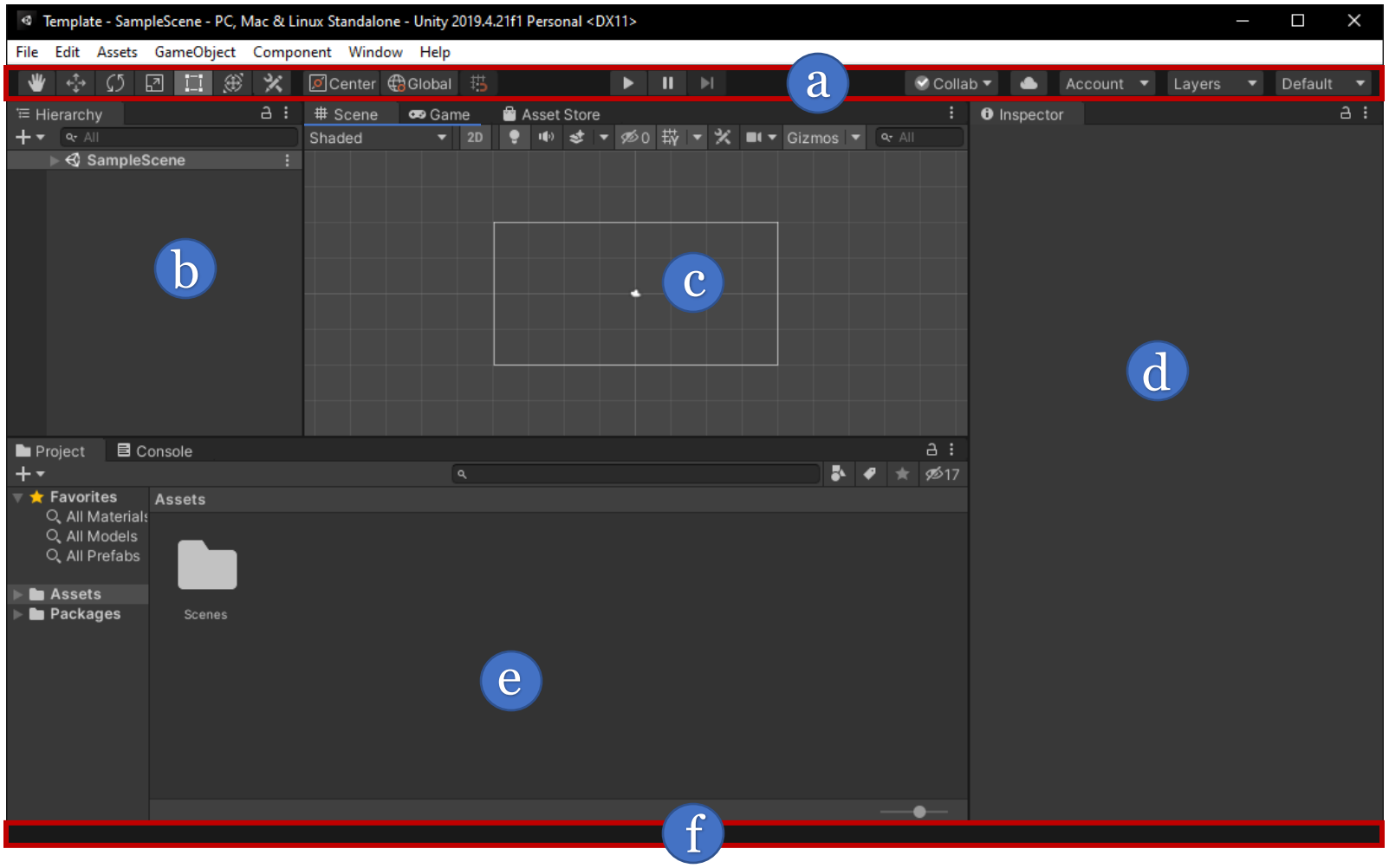
i. Clicking “Create” will open the main Unity Editor. This might take some time.
ii. Once opened, your Unity Window should look something like this:



d. If your Unity Editor does not look like this, change the Editor layout to “Default” in the top-right corner.



Learning Unity's Interface



1. Editor Windows

a. **Toolbar**- On the left side are tools for manipulating either the Scene view or selected GameObject. In the center are the play, pause, and step controls. To test your game, click the play button to start. On the right side, most notably, are the layer visibility menu and editor layout menu. These allow you to reorganize or reorder the windows in the editor or scene.

b. **Hierarchy Window**- This window represents every GameObject in the Scene. Anything loaded into the Scene will be displayed in the Hierarchy window. Here, GameObjects and their relationships can be viewed or selected.

c. **Scene/Game View**- Just below the Toolbar, the Scene or Game view can be selected. The Scene option allows you to edit and navigate your Scene; this is what you will be working in most of the time. The Game view displays what will actually be seen once the game is running. These windows can be simultaneously seen by dragging one of the views down into the scene or changing the Editor layout.

d. **Inspector Window-** This window displays all of the current properties of your selected GameObject. Here, you can add or edit components of your selected GameObject.

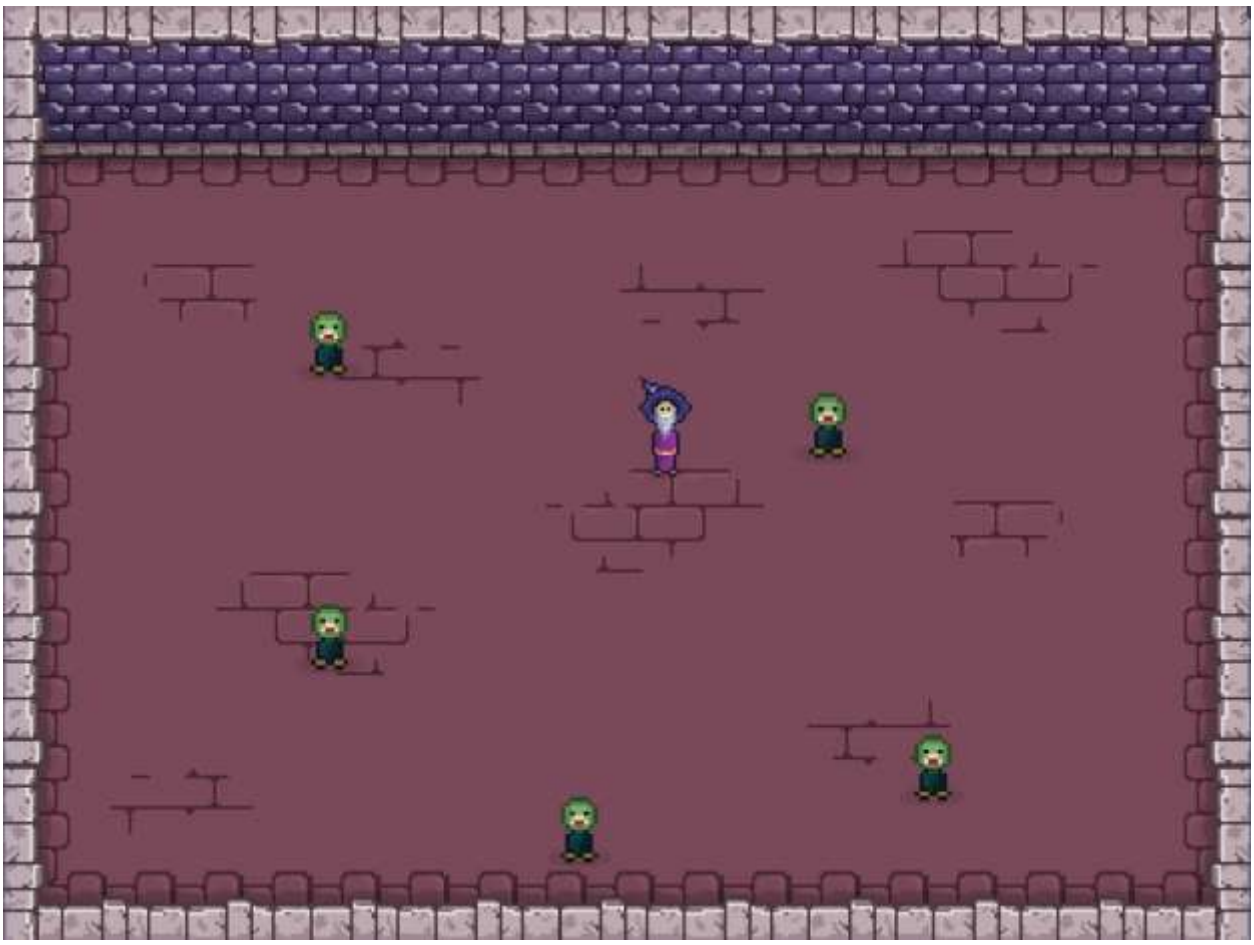
e. **Project Window-** All assets in your project's library can be found in the Project Window. This can include scripts, art, prefabs, GameObjects, sounds, music, etc.

f. **Status Bar-** This will display notifications about various Unity processes, most notably, errors.

These are the primary aspects of the Unity engine, and you will be constantly creating and editing your game through these. While there are many more features of these Editor windows in Unity, you do not need to know about them now. Through practicing with Unity and following this manual, you will become more familiar with Unity's more in-depth components. If you want to learn more about what each window has to offer, go to Unity's official user manual: https://docs.unity3d.com/Manual/UsingTheEditor.html.

Make a Video Game

The best way to learn Unity, is to *use* Unity. So, this manual will guide you through the process of making a video game with a detailed guide for making a basic top-down shooter. To do this, this manual will make use of free assets which can be imported from Unity's integrated Asset Store. If you would like to use the same assets shown in this manual, download and import "Brackeys 2D Pack." This is a free collection of characters, backgrounds, effects, sounds, UI elements, and other useful assets. With this asset pack, this manual will give you practice in making the roots of your game.

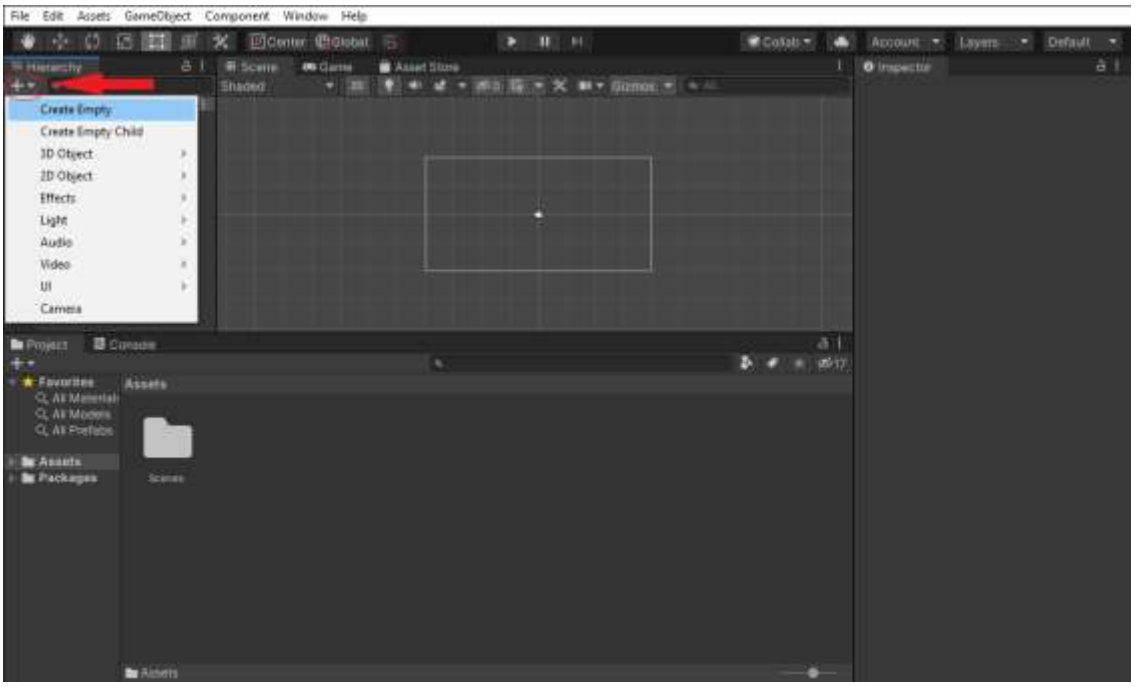


Make a Character with Basic Movement

1. Create an Empty Game Object

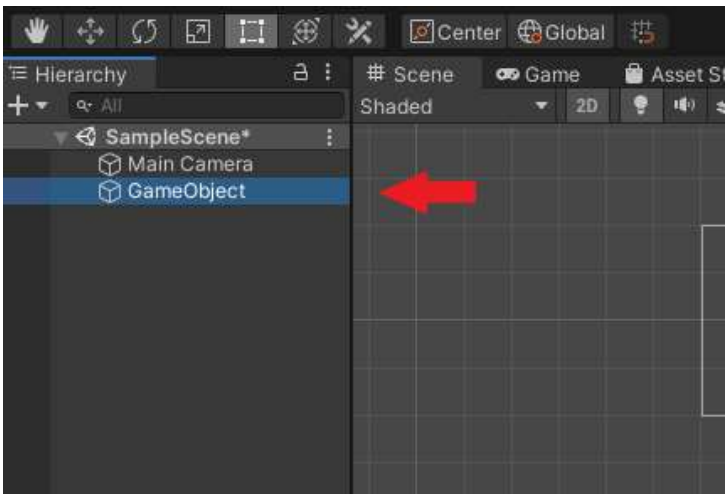
a. Click the + sign under the “Hierarchy” tab in the top-left corner and select “Create Empty”.

i. You can also do this by right clicking the space under the “Hierarchy” tab.



ii. This will create an Empty Game Object which can be used for anything such as terrain, buildings, enemies, projectiles, special effects, or anything else you might find in a video game. In our case, this Game Object will represent the main character that the player will control.

iii. You will see your created Game Object under the “Hierarchy” tab:

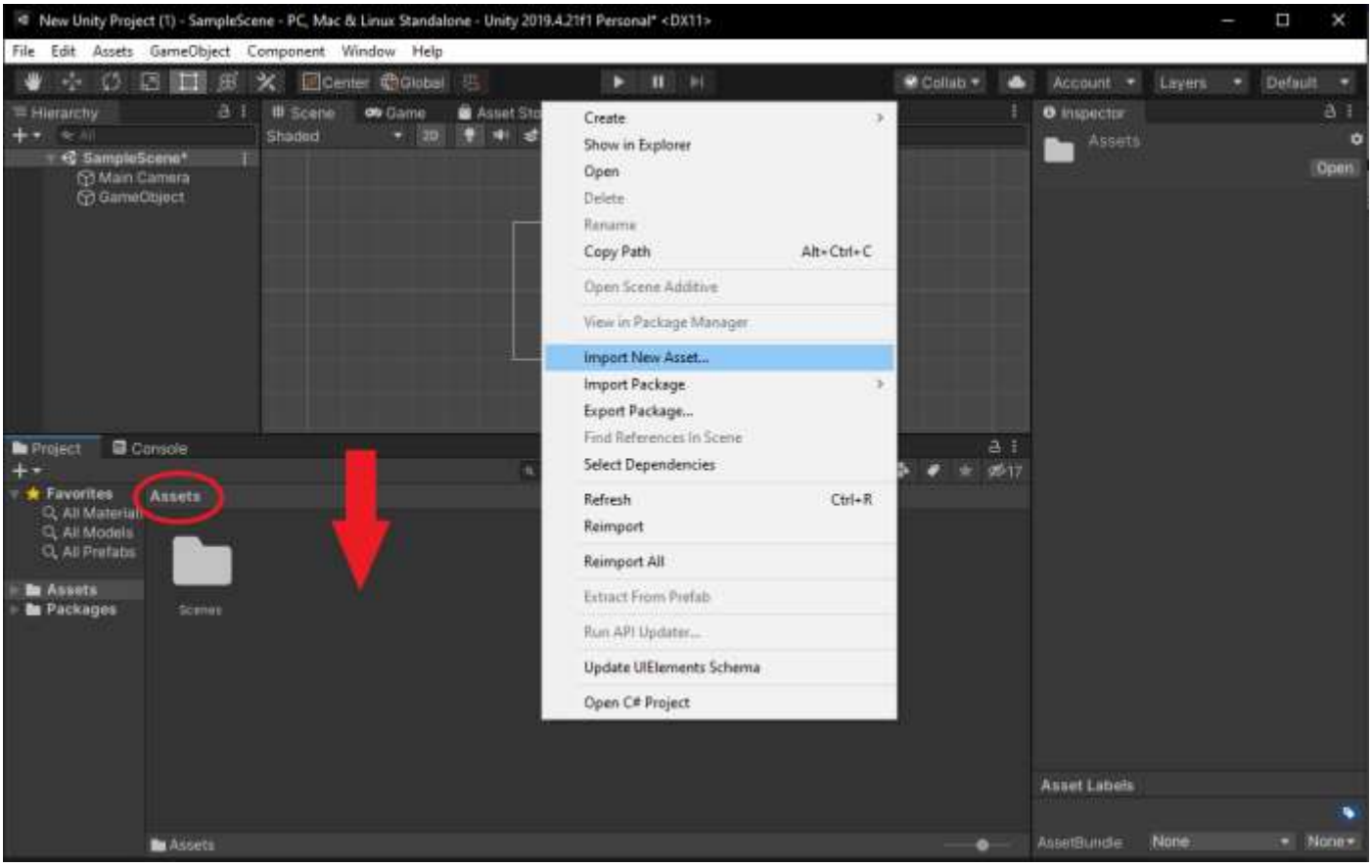


Important Note

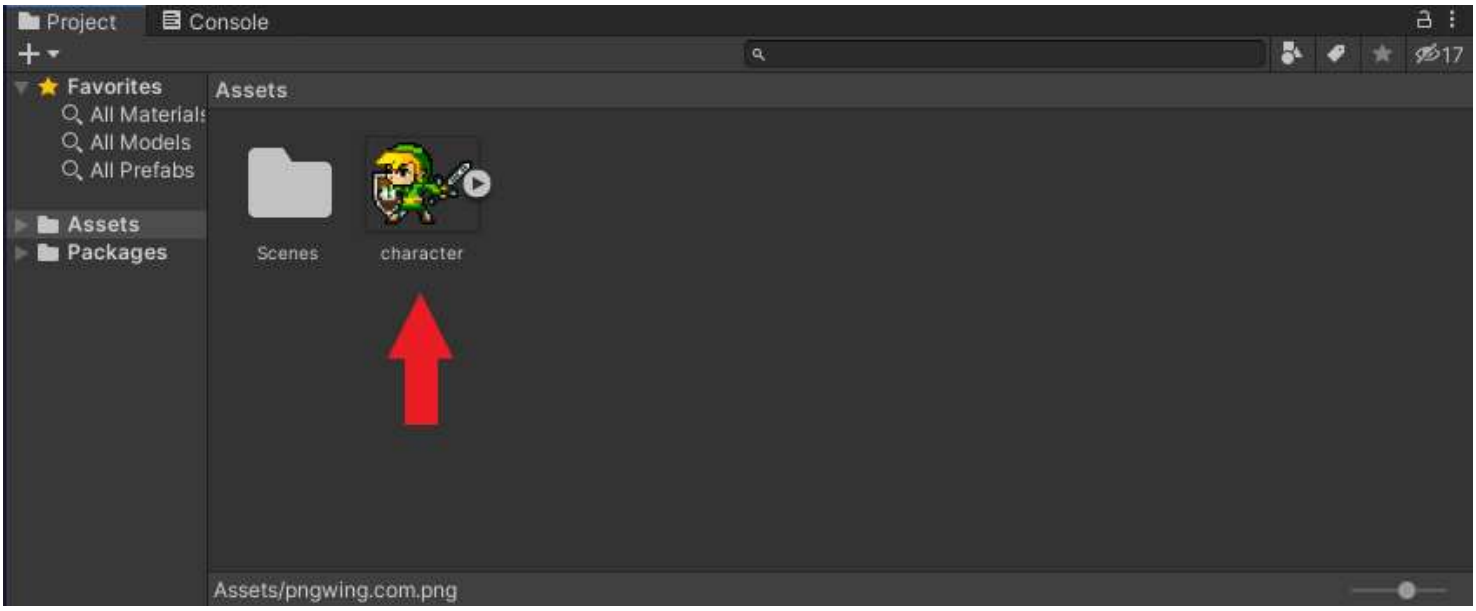
Whenever you modify or add components to your GameObject, make sure it is selected under the “Hierarchy” tab. GameObject is currently selected in the above picture.

2. Assign an Image to the Empty Game Object

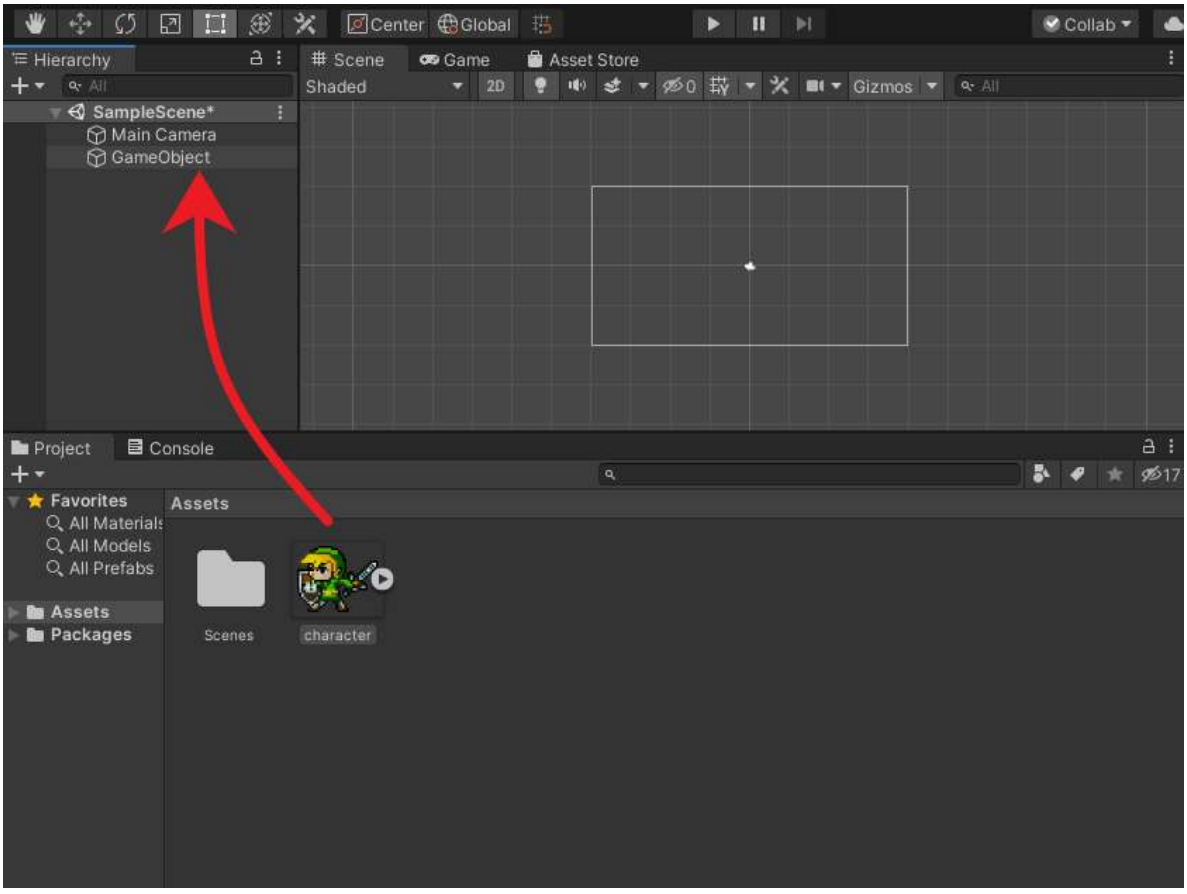
a. You must first, import a .png image you wish to use as the main character. Use a .png image with a transparent background, as images without transparent backgrounds will be displayed in Unity.
b. To import a .png image into Unity, right click the space under the “Assets” tab and select “Import New Asset” and select your desired image through Windows Explorer.
    i. You can also do this by dragging and dropping your desired image into the space under the “Assets” tab.
    ii. Your newly imported image will be shown under the “Assets” tab if correctly imported.



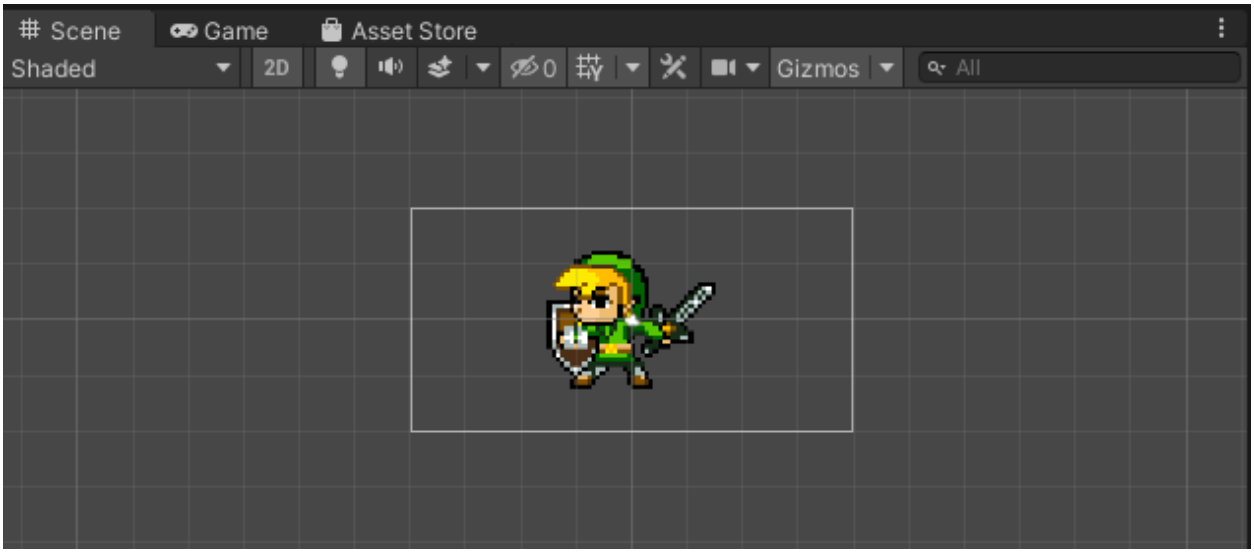
Below is a correctly imported .png image with a transparent background.



c. Assign your imported image to the GameObject you created by dragging the image from your "Assets" section onto your GameObject in the "Hierarchy" tab.

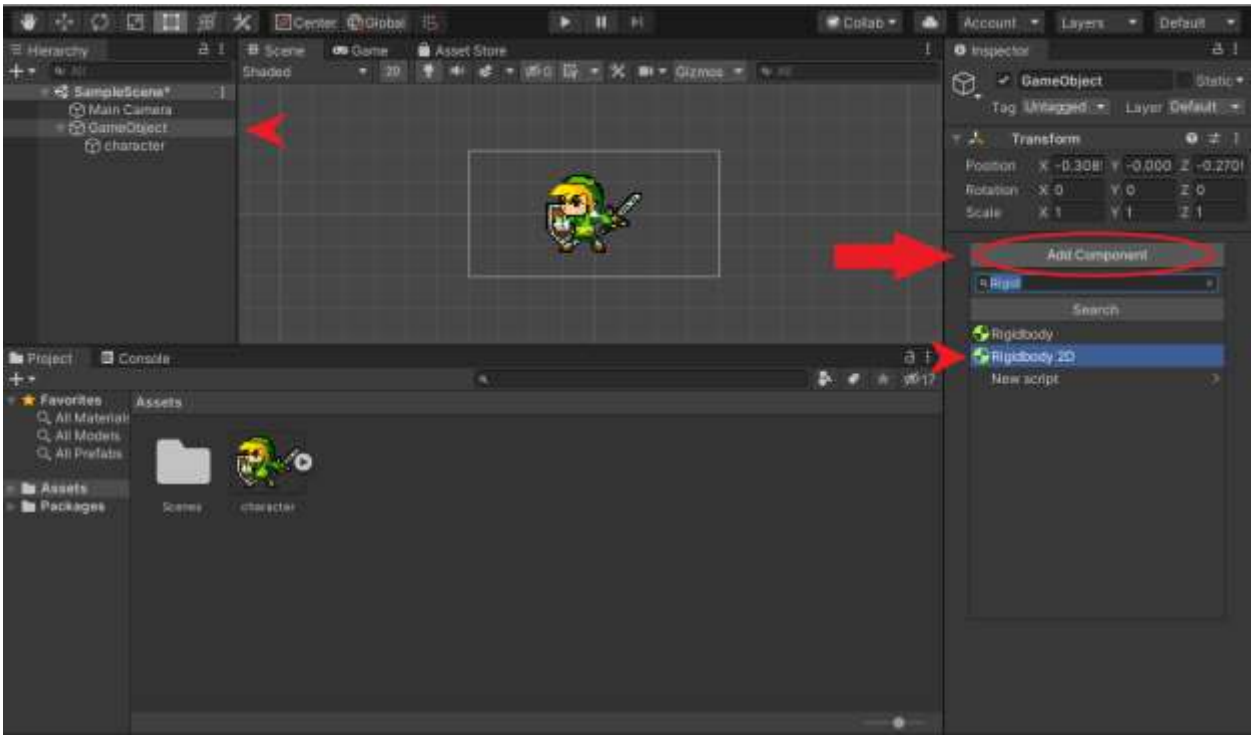


Once assigned, your imported .png will be displayed in the Unity Scene.

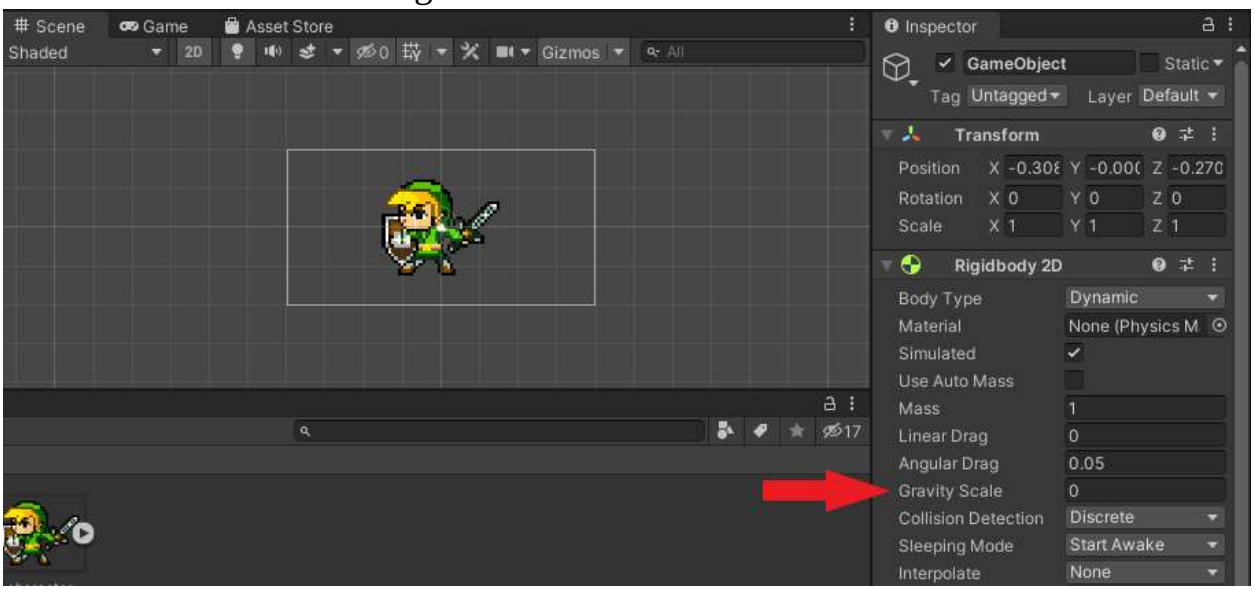


3. Add the Rigidbody 2D Component to your GameObject

a. First, click on your GameObject under the “Hierarchy” tab.
b. Next, click the “Add Component” button on the right side of the screen under the “Inspector” tab.
c. Search for and select the “Rigidbody 2D” component.
    i. This will later be used in the script to control your character.
    ii. This component can be used to control collisions, movement, gravity, and other aspects of your GameObject.



d. Once the “Rigidbody 2D” component has been added, change the gravity scale of your Object from 1 to 0.
    i. This will prevent your GameObject from falling out of the scene when testing.



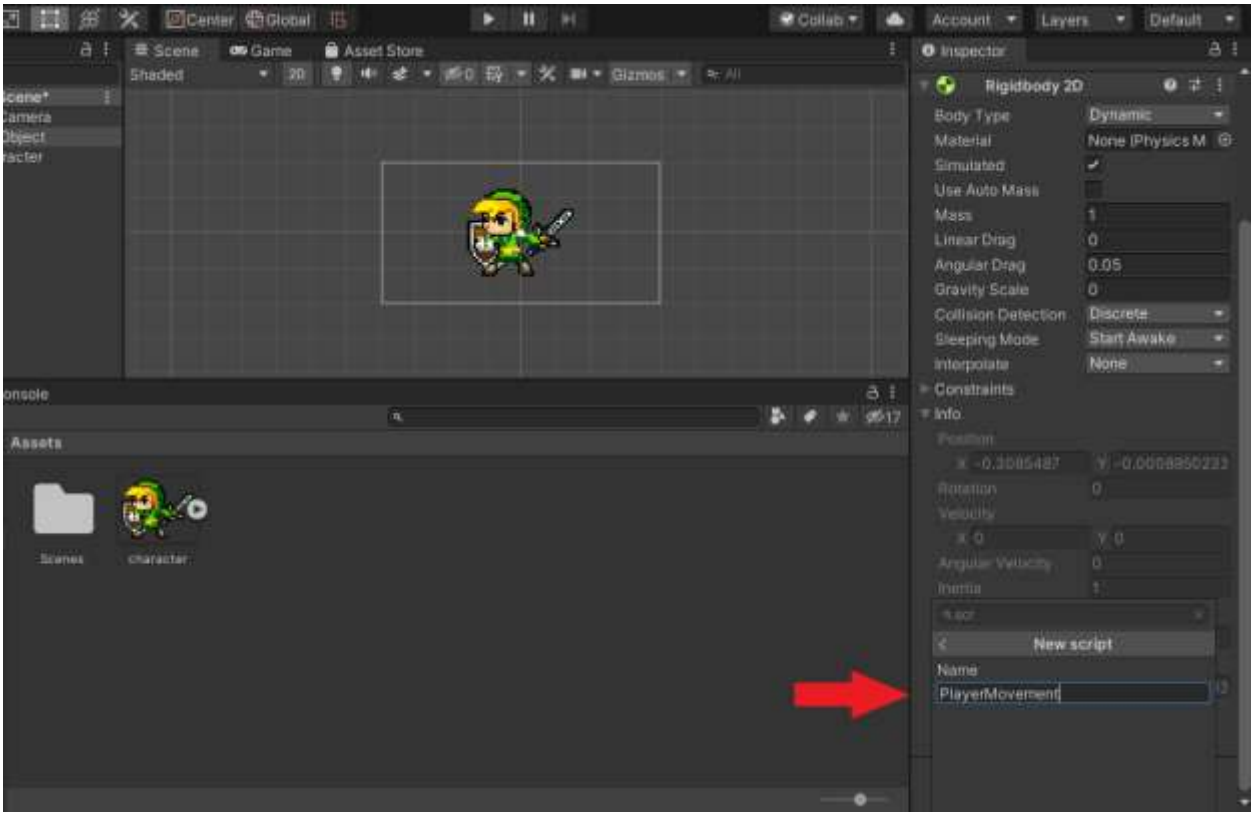
Important Note

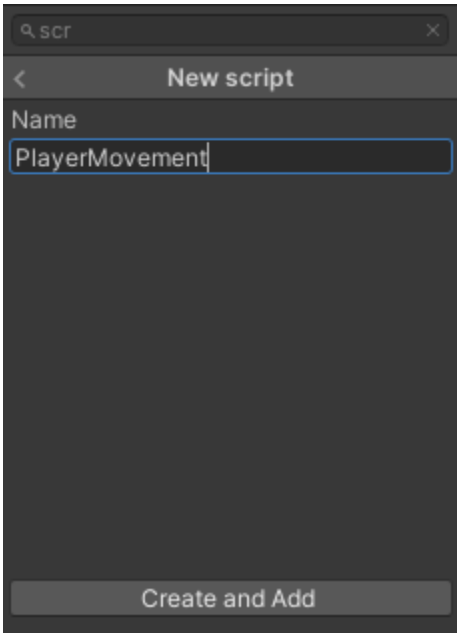
Make sure you are adding “Rigidbody 2D” and all future components to your GameObject and **NOT** to your newly assigned .png image. If all components are being added to your sprite and not the GameObject itself, you will be moving the image itself and not the actual GameObject.

4. Add the Script Component to your GameObject

a. The Script Component will be added just like the Rigidbody 2D Component; Click the "Add Component" button again and search for and select the "New Script" option.

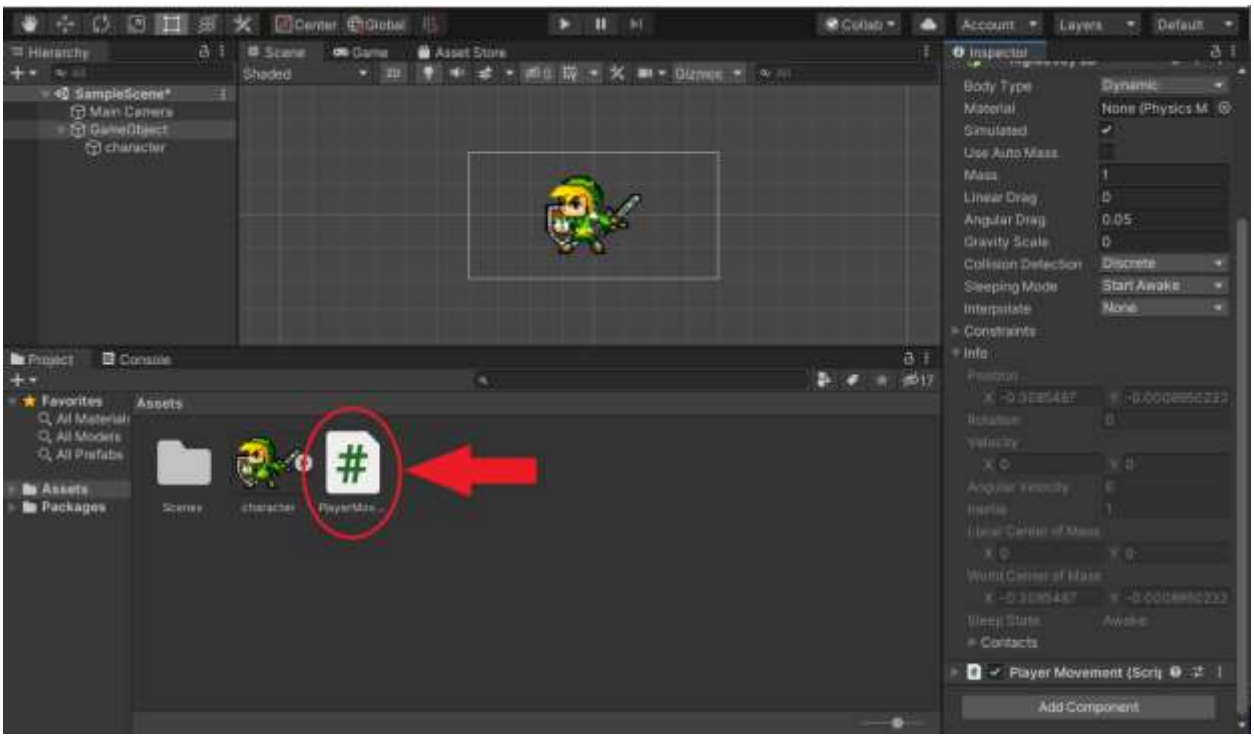
   i. You will be prompted with a window to name the script. The script name can be anything you like; but in our case, we'll name it: "PlayerMovement". Click Create and Add once you have named the script.



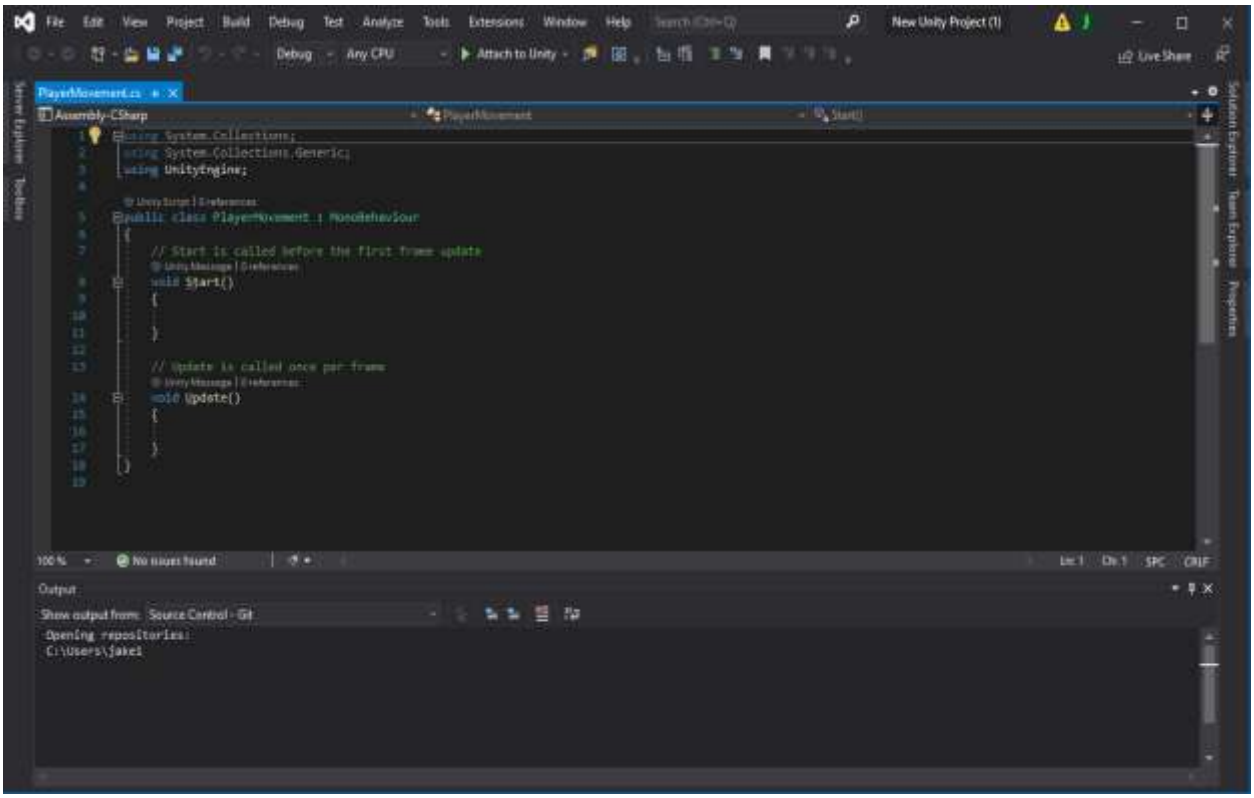


5. Edit the Script to Add Player Movement

a. Your newly added “PlayerMovement” script should now appear under the “Assets” tab. Double click the script to edit it with Visual Studio or any other compiler.



Once opened, your script should look something like this in Visual Studio:



b. Under the “public class PlayerMovement : MonoBehavior” add the following lines of code:

```
Unity Script | 0 references
public class PlayerMovement : MonoBehaviour
{
    Vector2 movement;
    public float movementSpeed = 5f;
    public Rigidbody2D rb;
```

c. Under “void Update()” add the following lines of code:
   i. Note: Any lines of code starting with // are comments and are unnecessary.

```
void Update()
{
    //WASD Movement
    movement.x = Input.GetAxisRaw("Horizontal");
    movement.y = Input.GetAxisRaw("Vertical");
}
```

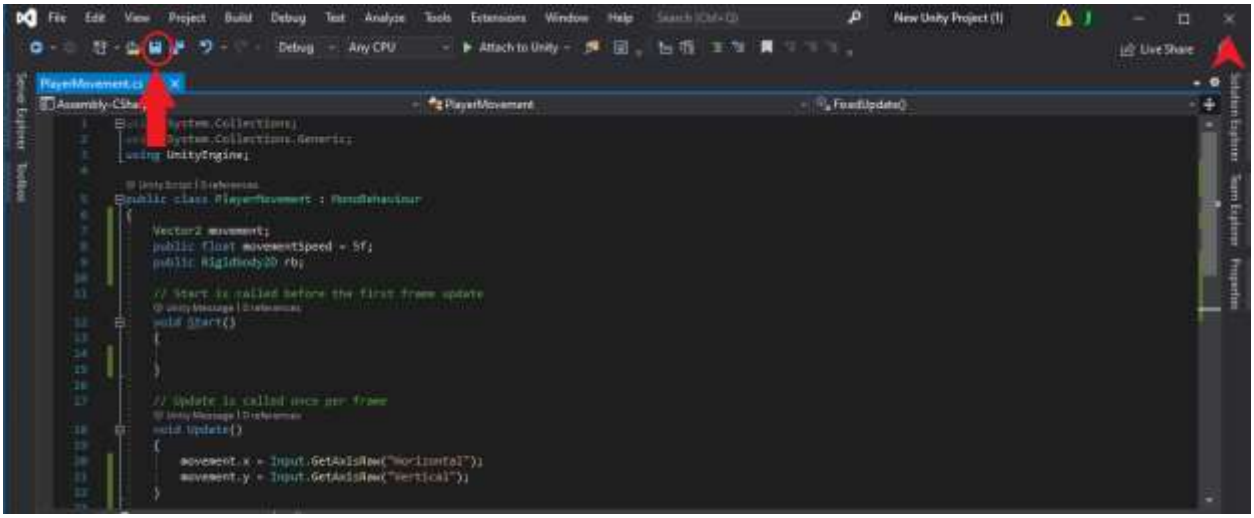
d. Add the following lines of code after “void Update()”:

```
private void FixedUpdate()
{
    //ClampMagnitude function is used to prevent increased diagonal movement speed
    movement = Vector2.ClampMagnitude(movement, 1);
    rb.MovePosition(rb.position + (movement * movementSpeed * Time.fixedDeltaTime));
}
```

Your entire script should look like this:

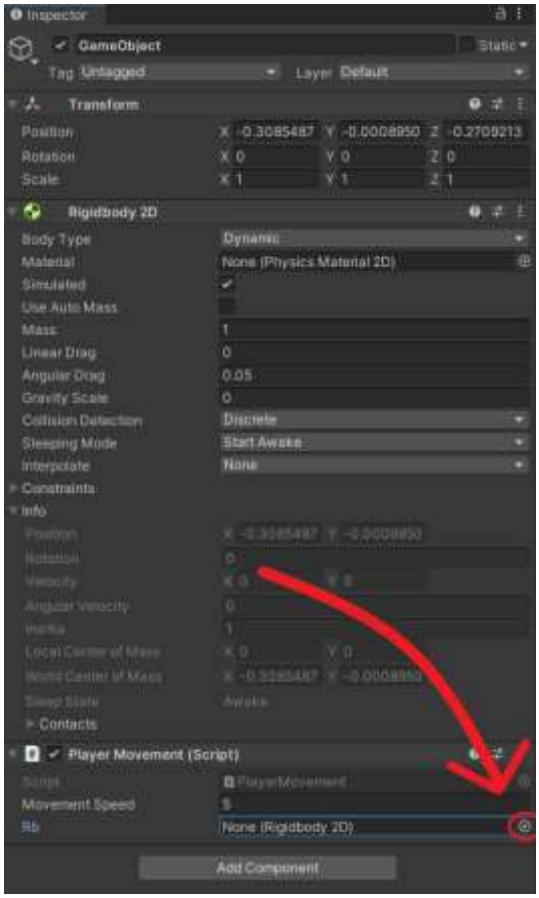
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity Script | 0 references
public class PlayerMovement : MonoBehaviour
{
    Vector2 movement;
    public float movementSpeed = 5f;
    public Rigidbody2D rb;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {

    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        //WASD Movement
        movement.x = Input.GetAxisRaw("Horizontal");
        movement.y = Input.GetAxisRaw("Vertical");
    }

    Unity Message | 0 references
    private void FixedUpdate()
    {
        //ClampMagnitude function is used to prevent increased diagonal movement speed
        movement = Vector2.ClampMagnitude(movement, 1);
        rb.MovePosition(rb.position + (movement * movementSpeed * Time.fixedDeltaTime));
    }
}
```

e. Save by clicking on the floppy disc symbol in the top-left and then exit Visual Studio by clicking the x in the top-right.

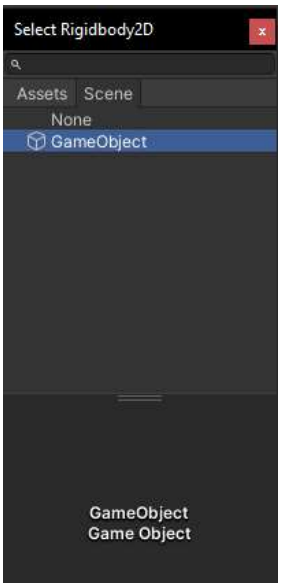


6. Assign GameObject to the Rb field under the PlayerMovement Script

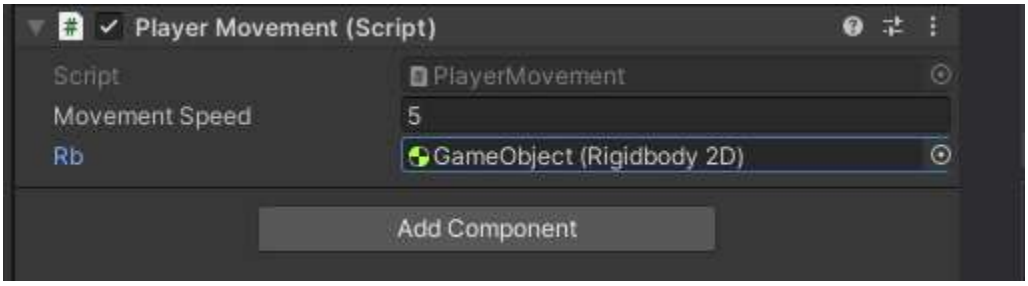
a. Under the Unity Inspector, assign the empty Rb field to your GameObject by clicking the dotted-circle symbol next to “Rb” under the Script component and selecting “GameObject”.
    i. You can also drag the GameObject from the “Hierarchy” tab onto the empty “Rb” field.



ii. Select “GameObject” from the pop-up window:



Once assigned, the “Rb” field should look like this:



7. Test Player Movement

a. Test your script by clicking the “Play” button at the top-center of Unity. Try controlling your character using WASD or the arrow keys.

Make the Character Change Direction

1. Import Different Images for Each Direction

a. Instead of having a static image float across the screen, import 4 different images for each direction the player can move: up, down, left, and right.

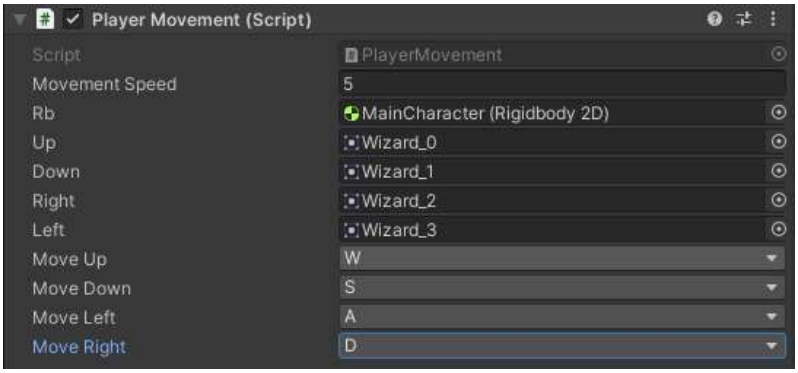
b. Remove your static, attached image from your MainCharacter (GameObject) and edit your PlayerMovement Script again.

c. Add the following lines of code to your PlayerMovement Script just below your already added movement parameters:

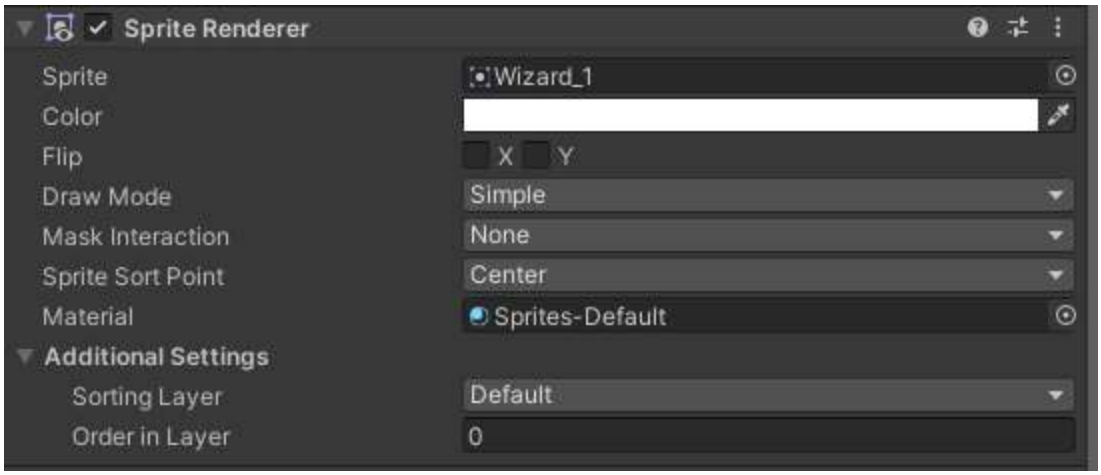
```
//Movement
Vector2 movement;
public float movementSpeed = 5f;
public Rigidbody2D rb;

//Sprites
public Sprite up;
public Sprite down;
public Sprite right;
public Sprite left;

//Controls
public KeyCode moveUp;
public KeyCode moveDown;
public KeyCode moveLeft;
public KeyCode moveRight;
```

i. The Sprite variables will allow you to assign each appropriate Sprite with their corresponding direction in the Unity Editor.

ii. The newly added controls will allow you to dynamically change the key-bindings which will be responsible for changing the direction of your character.



d. Add the Sprite Renderer Component to your MainCharacter and assign of the directional images as your starting, default sprite.



2. Add Logic for Sprite Change

a. Add the following lines of code to the void Update() section of your PlayerMovement Script:

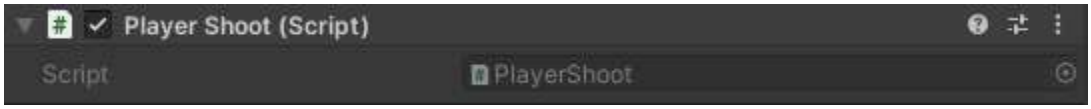
```
void Update()
{
    //WASD Movement
    movement.x = Input.GetAxisRaw("Horizontal");
    movement.y = Input.GetAxisRaw("Vertical");

    //Sprite Control
    if (Input.GetKeyDown(moveUp))
    {
        GetComponent<SpriteRenderer>().sprite = up;
    }
    else if (Input.GetKeyDown(moveDown))
    {
        GetComponent<SpriteRenderer>().sprite = down;
    }
    else if (Input.GetKeyDown(moveLeft))
    {
        GetComponent<SpriteRenderer>().sprite = left;
    }
    else if (Input.GetKeyDown(moveRight))
    {
        GetComponent<SpriteRenderer>().sprite = right;
    }
}
```

i. Based on your assigned keycodes in the Unity Inspector, the script will change the sprite for each direction.

Make the Character Fire Projectiles

1. Create New Script and Shoot Origin with Projectile

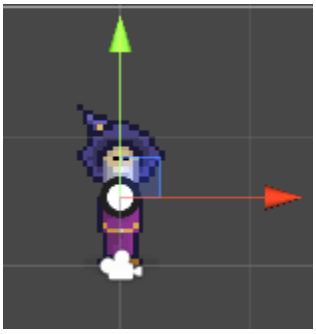
a. Now that your character can look and move around, we will start to work on what will be the basic combat of your game. Add another script to your MainCharacter called “PlayerShoot”:



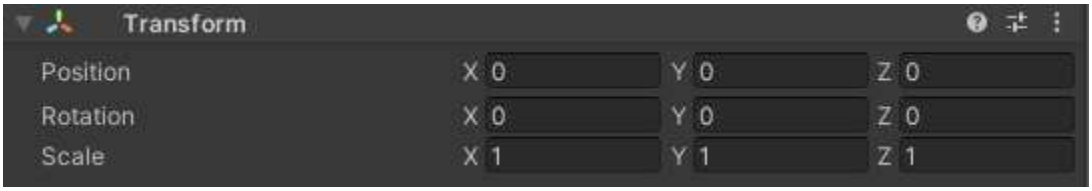
b. Before opening and editing your new Script, create a new empty Game Object in the Hierarchy tab called “Shoot Origin” and attach it to Main Character as its child.

i. This will later be referenced in the Player Shoot Script where the MainCharacter will be firing projectiles from. As the MainCharacter’s child, Shoot Origin will inherit the Main Character’s movement.

c. Move Shoot Origin in the scene to where you wish to shoot projectiles from on your character. You can attach a temporary sprite or shape to Shoot Origin to help do this.

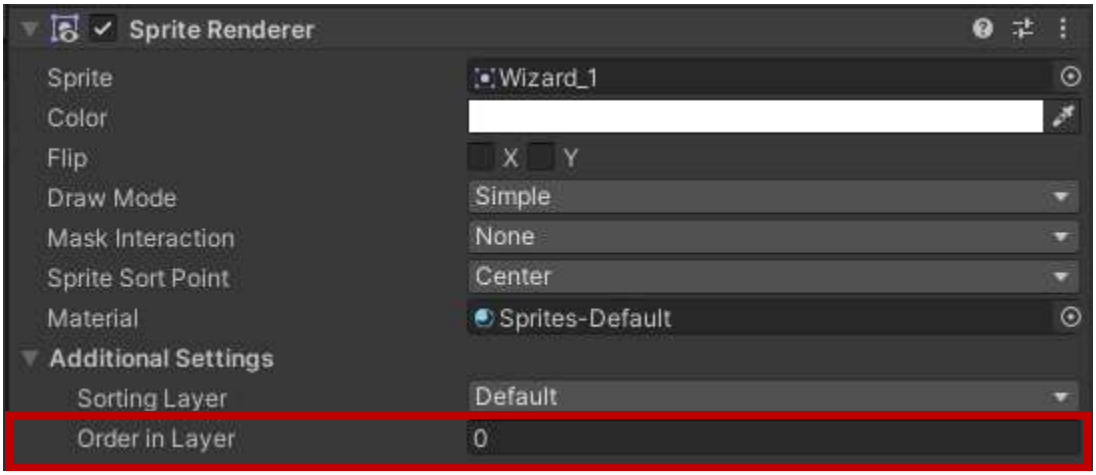


i. You can move an object via the green vertical axis and red horizontal axis or click and drag the blue box to move the object around freely. You can also manually set an object’s coordinates in the Transform window of the inspector.



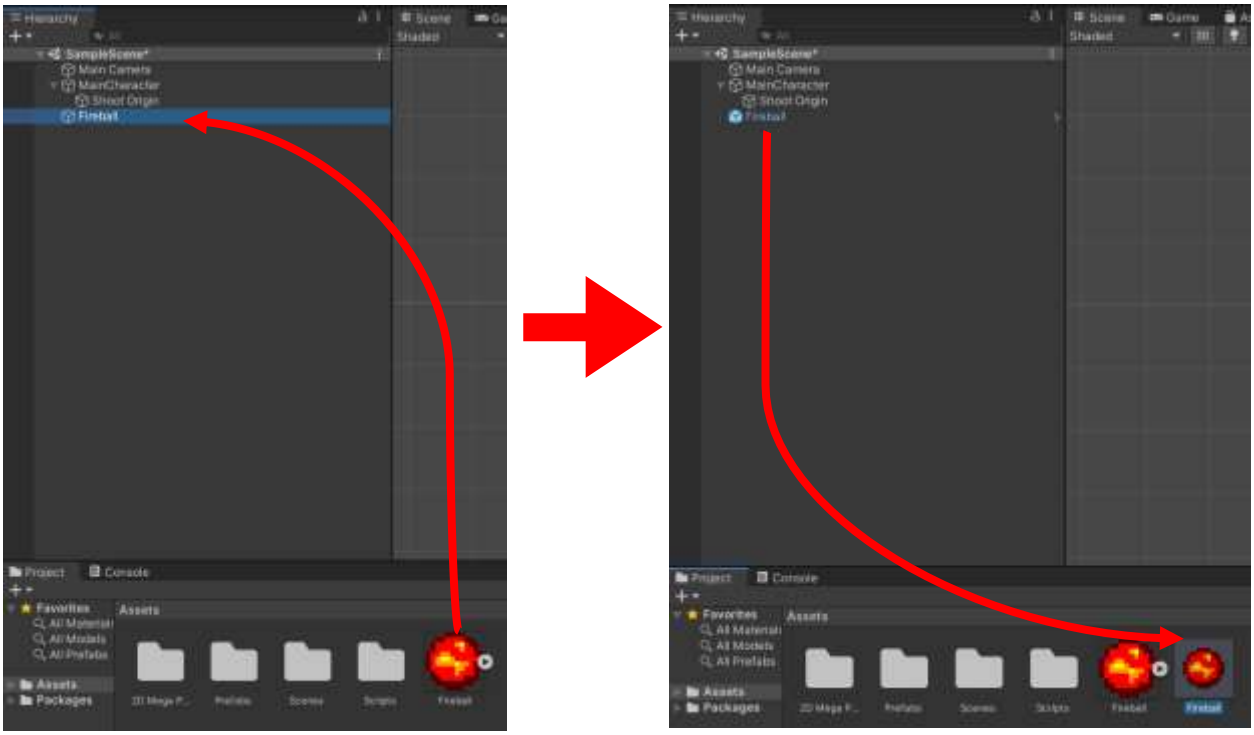
Important Note

Whenever you add new Sprites, you can determine which sprites will appear over others using the Sprite Render Component's "Order in Layer" feature. The numerical value of a Sprite Renderer's Order in Layer will determine its priority in the rendering queue. The higher the number, the more priority that Sprite will receive. If you want an object to appear in front of another object, set the object that you wish to be in front to a higher number than the object behind it.



For the purposes of this manual, make sure that each new object that is added to the scene has the same Z axis in its transform window. Otherwise, object collisions will not work properly.

d. Next, import the picture you wish to use as a projectile as a prefab.
    i. From Unity's official manual, "**Prefabs** are a special type of component that allows fully configured GameObjects to be saved in the Project for reuse. These assets can then be shared between scenes, or even other projects without having to be configured again."
e. To make your picture of a projectile a prefab, drag the picture from your Assets window onto the Hierarchy Tab and then drag that same, new GameObject back into your assets. Add the RigidBody2D component to the Prefab in the Assets section, not under the hierarchy tab. Examine the pictures below for more clarity:



f. Then, delete the projectile in the Hierarchy tab only.
   i. This will leave you with just the prefab for the projectile which will be used in the “Player Shoot” Script.
   ii. Below is the prefab for the projectile:

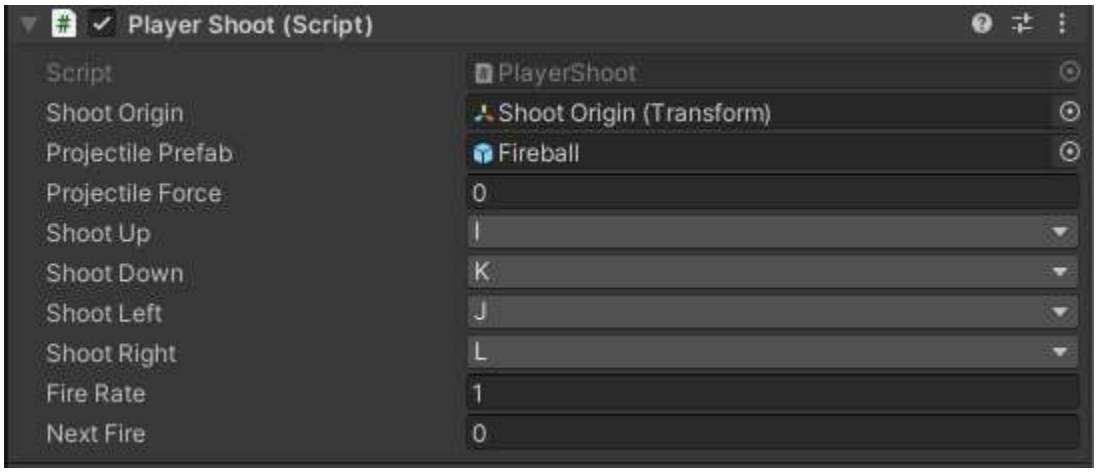
   iii. This will allow us to “instantiate” or make many instances of our projectile for use in the game.

2. Edit PlayerShoot Script to Add Projectile Logic

a. Add the following lines of code to your Player Shoot Script:

```
public class PlayerShoot : MonoBehaviour
{
    //Projectile References
    public Transform shootOrigin;
    public GameObject projectilePrefab;
    public float projectileForce;
    Rigidbody2D rb;

    //Shoot Controls
    public KeyCode shootUp;
    public KeyCode shootDown;
    public KeyCode shootLeft;
    public KeyCode shootRight;

    //Cooldown on Fire Rate
    public float fireRate;
    public float nextFire;
```

b. Assign the appropriate members to the newly added fields in the PlayerShoot Script in the Inspector. You can make the Shoot controls whatever you like, as long as they are not the same controls being used for movement. Make Fire Rate 1.



i. The Fire Rate and Next Fire variables will be used to limit the amount of times the MainCharacter can shoot in a certain time interval. With Fire Rate at 1, the main character will only be able to fire once every second. Make the Projectile Force however fast you wish your projectile to travel.

c. Add the following lines of code to your PlayerShoot Script:

```
void Update()
{
    if (Input.GetKeyDown(shootUp) && Time.time > nextFire)
    {
        nextFire = Time.time + fireRate; //resets nextFire time to a new time
        Shoot();
        rb.AddForce(shootOrigin.up * projectileForce, ForceMode2D.Impulse);
    }
    else if (Input.GetKeyDown(shootDown) && Time.time > nextFire)
    {
        nextFire = Time.time + fireRate;
        Shoot();
        rb.AddForce(-shootOrigin.up * projectileForce, ForceMode2D.Impulse);
    }
    else if (Input.GetKeyDown(shootLeft) && Time.time > nextFire)
    {
        nextFire = Time.time + fireRate;
        Shoot();
        rb.AddForce(-shootOrigin.right * projectileForce, ForceMode2D.Impulse);
    }
    else if (Input.GetKeyDown(shootRight) && Time.time > nextFire)
    {
        nextFire = Time.time + fireRate;
        Shoot();
        rb.AddForce(shootOrigin.right * projectileForce, ForceMode2D.Impulse);
    }
}
4 references
void Shoot()
{
    GameObject projectile = Instantiate(projectilePrefab, shootOrigin.position, Quaternion.identity);
    rb = projectile.GetComponent<Rigidbody2D>();
}
}
```

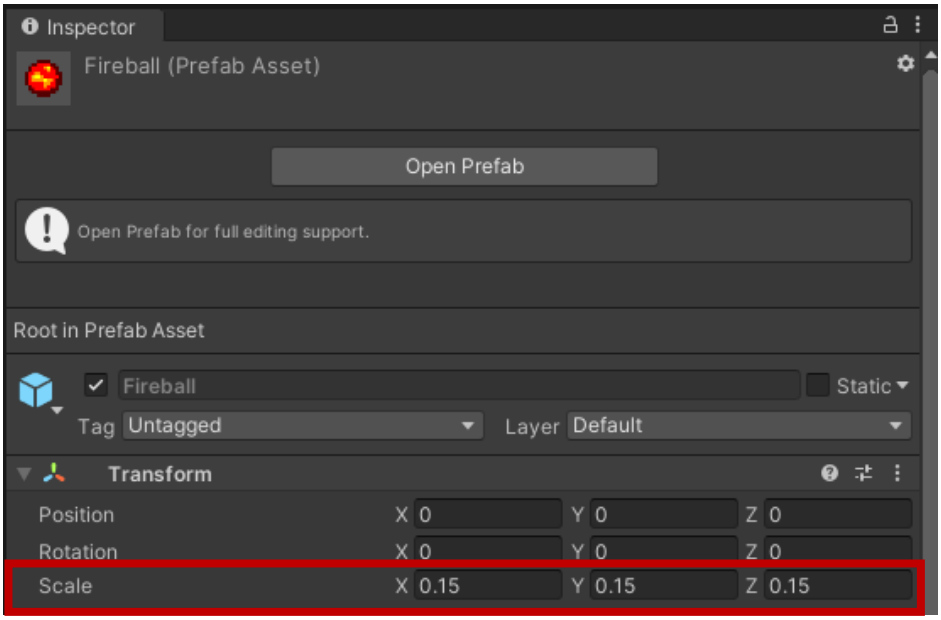
i. Let's dissect this code. First, it checks to see if any of the four Keycodes that you set in the Unity Inspector (key-bindings) are pressed, then it checks to see if the current time is greater than the designated next time to fire. This prevents the MainCharacter from firing as fast as possible. As soon as both conditions are met, the next designated time to fire is increased by the fireRate which is 1 in the example. If the player were to try to fire again before the next designated time, the condition would fail, because the current time would be less than the designated time to fire.

ii. As for how the projectile is made, the newly implemented Shoot() function is called, which instantiates a copy of the projectile prefab at the shootOrigin's position. Then the variable, rb, is assigned to the projectile prefab's Rigidbody2D component.

3. Test PlayerShoot Script

a. Check your work by clicking the Play button on the Toolbar at the top-center of the screen. Check that you can fire up, down, left and right with your assigned keys.

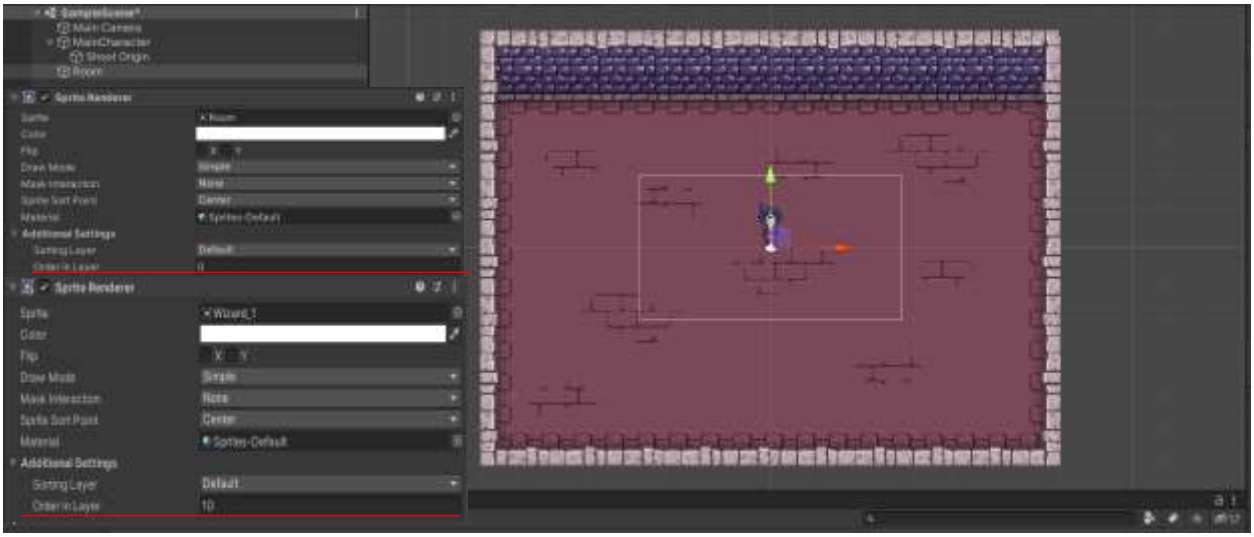
b. If your projectile is too large or small, you can edit its scale by selecting the projectile prefab in the Assets section and adjusting the x, y, and z numbers in the Transform section in the Unity Inspector.



Make the Environment

1. Import and Place Assets

a. Start by importing any asset which you think your character would look suitable standing on.
b. Drag your imported asset onto the hierarchy tab to be placed in the scene.
c. Make your imported asset's Order in Layer less than your Main Character's so that your Main Character will be in front of the newly imported Asset.

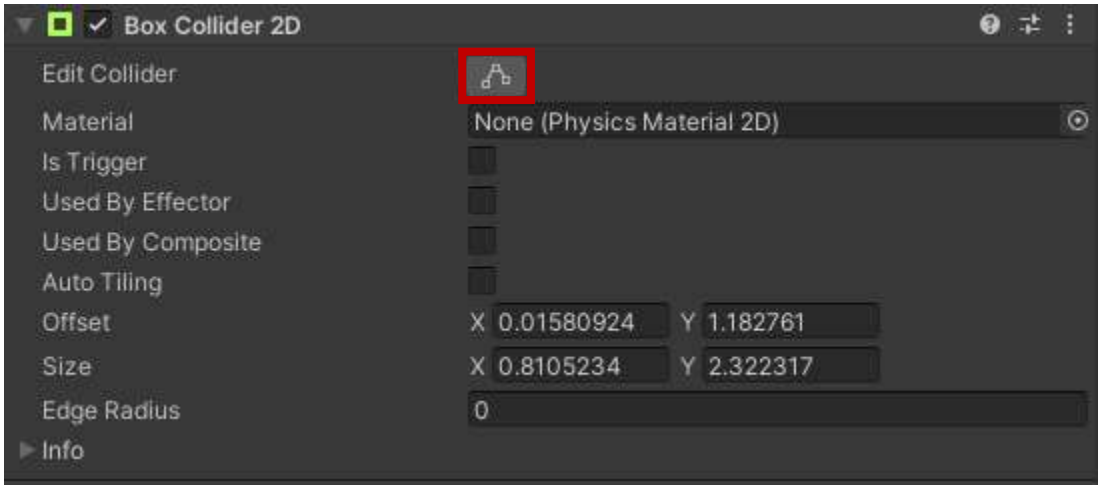


    i. Feel free to add more rooms, terrain, objects, or anything else to fill your game world.

2. Add Collision to Player and Objects

a. Right now, your MainCharacter will just walk on top of and over walls and objects. To stop this, we must add collisions to both the player and environment. Add the Box Collider 2D component to your MainCharacter.
    i. This will not only stop the MainCharacter from walking through objects, but it will also set up our foundation for combat.

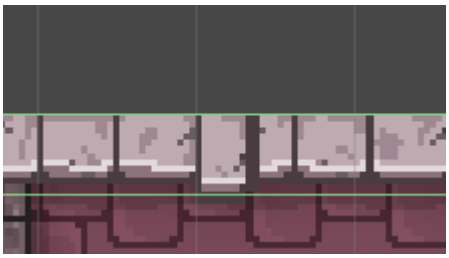
b. Click the “Edit Collider” button at the top of the Box Collider 2D component to edit the hitbox of your MainCharacter.



i. Generally, it is good practice to make the hitboxes of the character lenient for the player.

c. To add collisions to objects, create Empty Game Objects under the Hierarchy tab and add the Box Collider 2D component to them. Then, you can drag those objects and edit their box colliders to align with whatever objects you wish to give collision. In our example, objects with collisions will be assigned to each of the four walls.

i. You can organize your objects by making them children of another empty Game Object and minimizing them in the hierarchy tab.



Make the Enemies

1. Import your Enemy and Make a Prefab

a. This will be done exactly like the projectile. Drag your imported image of the enemy you wish to use onto the hierarchy tab.
b. Drag that same image you dragged onto the hierarchy tab back into your Assets.
c. Delete the enemy in your hierarchy.

2. Add Necessary Components to Enemy Prefab

a. Add the Rigidbody 2D component.
    i. Do not forget to set its gravity scale to 0.
b. Add the Box Collider 2D component.
    i. Edit its hitbox with the Edit Collider button at the top of the window.
c. Add a new script to the Enemy prefab and name it “EnemyBehavior”.

3. Open and Edit EnemyBehavior Script

a. Add the following variables to your script:

```
public class EnemyBehavior : MonoBehaviour
{
    //Movement
    public float movementSpeed = 5f;
    public Rigidbody2D rb;

    //Aggro
    public float aggroDistance;
    public bool aggro;

    //Parameters
    public Transform target;
    public Transform self;

    //Health
    public int health = 3;
```

b. Assign the Enemy's RigidBody 2D component to the rb field in the Inspector.
c. Assign the Transform component of your Enemy prefab to the self variable field in the inspector.
d. The health variable will be used for combat and should also be added to the PlayerMovement script.
e. Change the aggroDistance variable to whatever distance you wish the enemy to begin attacking the MainCharacter.
f. The target variable cannot be assigned until you drag the prefab back into the hierarchy and load it into the scene. When loaded, assign the MainCharacter's transform component as target in the Unity Inspector.
g. Add the following line of code to both this script and the PlayerMovement Script to prevent Rotation of your objects now that they have collision:

```
void Start()
{
    rb.freezeRotation = true;
}
```

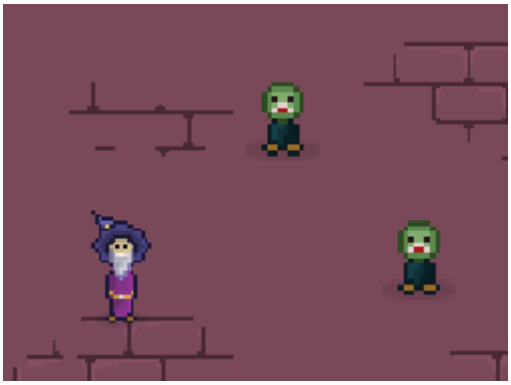
h. Add the following lines of code to the Enemy Behavior Script:

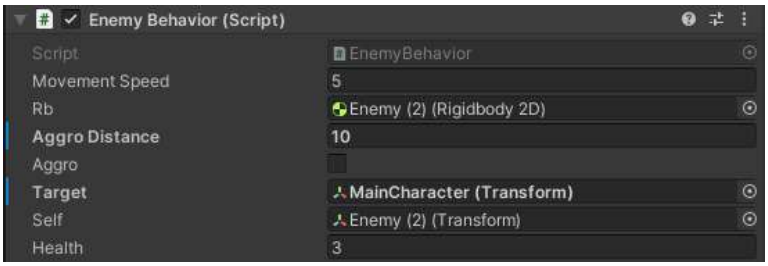
```
private void FixedUpdate()
{
    if (aggro == true)
    {
        //move towards player
        transform.position = Vector2.MoveTowards(transform.position, target.position, movementSpeed * Time.deltaTime);

    }
    AIUpdate();
}

1 reference
private void AIUpdate()
{
    //Sense Player
    var playerPosition = target.position;
    var selfPosition = self.position;

    //Think
    var distanceToPlayer = Vector3.Distance(selfPosition, playerPosition);
    var playerIsOnLeft = (playerPosition.x < selfPosition.x);

    if (distanceToPlayer <= aggroDistance)
    {
        aggro = true;
    }

}
```

i. Let's see what the code is doing.
    i. It first checks to see if aggro is true, which is false by default.
    ii. Then it calls the function we made called AIUpdate(). This function looks at the position of the MainCharacter and itself, and then it checks to see if the distance between the MainCharacter and Enemy is less than the aggroDistance. If it is, aggro becomes true. Once aggro is true, the Enemy will move towards the MainCharacter.

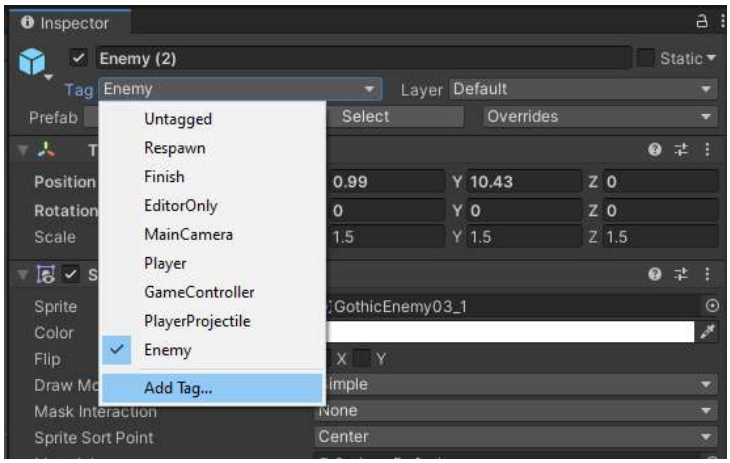
4. Make the Enemy Prefab Hurt

a. For now, close the EnemyBehavior script and test if everything is working properly by adding a couple of enemies into the scene by dragging the prefab onto the hierarchy tab and assigning their target fields to the MainCharacter's transform component.





b. Add and Assign the "Enemy" tag to your Enemy prefab by clicking the "Add Tag" option under the Tag button at the top of the Unity Inspector when the Enemy prefab is selected.



c. Open and edit your PlayerMovement Script again and add the following lines of code:

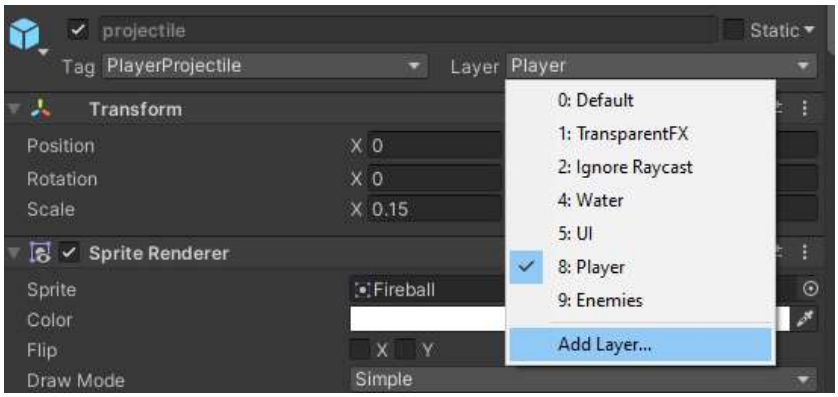
```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Enemy")
    {
        health -= 1;
        takeDamageSound.Play();
        if (health <= 0)
        {
            //Die Result
        }
    }
}
```

i. This makes it so that when the Enemy prefab comes into contact with the player, the player will take damage. Once all health is lost, you can write whatever happens when you die. You can reload the level, quit the application, or load another scene. Anything can be done here, so it's your choice.

Make the Character Fight Back

1. Fix Collision

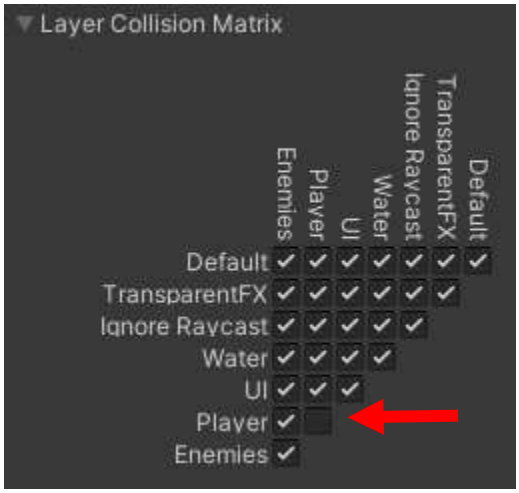
a. If you have not already noticed by testing, the MainCharacter's projectiles collide with itself, which is not good. To fix this, we'll use Unity's built in Layer Collision Matrix. First, select your projectile prefab and add and assign a custom layer called, "Player".



b. Repeat the above step exactly the same for the MainCharacter.

c. Add and assign another custom layer to the Enemy prefab called, "Enemies".

d. Go to the top-left of the Unity Editor and select: Edit -> Project Settings -> Physics 2D. Scroll to the bottom and open the Layer Collision Matrix.
e. Untick the checkmark between Player and Player in the Layer Collision Matrix.



i. This will ignore all collisions among objects of Layer type "Player".

2. Add and Edit projectile Script

a. Select your projectile prefab and add a new script called, "ProjectileBehavior".
b. Open the script and add the following lines of code:

```
public class ProjectileBehavior : MonoBehaviour
{
    public GameObject enemy;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {

    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {

    }

    Unity Message | 0 references
    void OnCollisionEnter2D(Collision2D collision)
    {

        Destroy(gameObject);
    }
}
```

i. This script will destroy the projectile once it collides with any object.

c. Add and assign a new tag to the projectile prefab called, “PlayerProjectile”.

3. Edit the EnemyBehavior Script

a. The MainCharacter can move, shoot projectiles, take damage, and die. Now it’s time for the Enemy prefab to be able to take damage from the MainCharacter. Add the following lines of code to the EnemyBehavior Script:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "PlayerProjectile")
    {
        aggro = true;
        health -= 1;
        if (health <= 0)
        {
            movementSpeed = 0f;
            Destroy(gameObject);
        }
    }
}
```

i. This code will, on collision, check if the colliding object has the tag we assigned to our projectile prefab, “PlayerProjectile”. If true, the Enemy prefab will take damage and destroy itself if its health is equal to or less than 0.

4. Test Your Game

a. Test your game by clicking the Play button at the top-center of the screen on the Toolbar.

b. Check the collisions of your objects, damage states, projectile logic, or anything else. For debugging, you can insert print(“Say Something Here”) into your different Scripts. Ensure that all objects in the scene are on the same Z axis and that their Order in Layer values make sense.

Make Your Game

This concludes the User Manual for 2D Game Development with Unity. Now that you've finished making the manual's game, it's time to make your own game. You can either continue to expand upon this game or make something entirely new. There's still plenty to learn and implement. Shouldn't there be a User-Interface for the player's health? Where is the Main Menu to load into the game? Why aren't there sound effects and music? Where are the animations? How do I progress? How do I win? These are all questions that must be answered when making a video game, and the first step to solving them is to learn and practice. Take to the internet to learn and expand upon the tools learned here and make something you can be proud of.

Bibliography

Free 2D Mega pack: 2d: Unity asset store. (n.d.). Retrieved April 23, 2021, from https://assetstore.unity.com/packages/2d/free-2d-mega-pack-177430.

Technologies, U. (n.d.). Unity user manual 2020.3 (lts). Retrieved March 24, 2021, from https://docs.unity3d.com/Manual/index.html.