ECS657U: Multi-Platform Game Development

Game name:

Square Dasher Bob

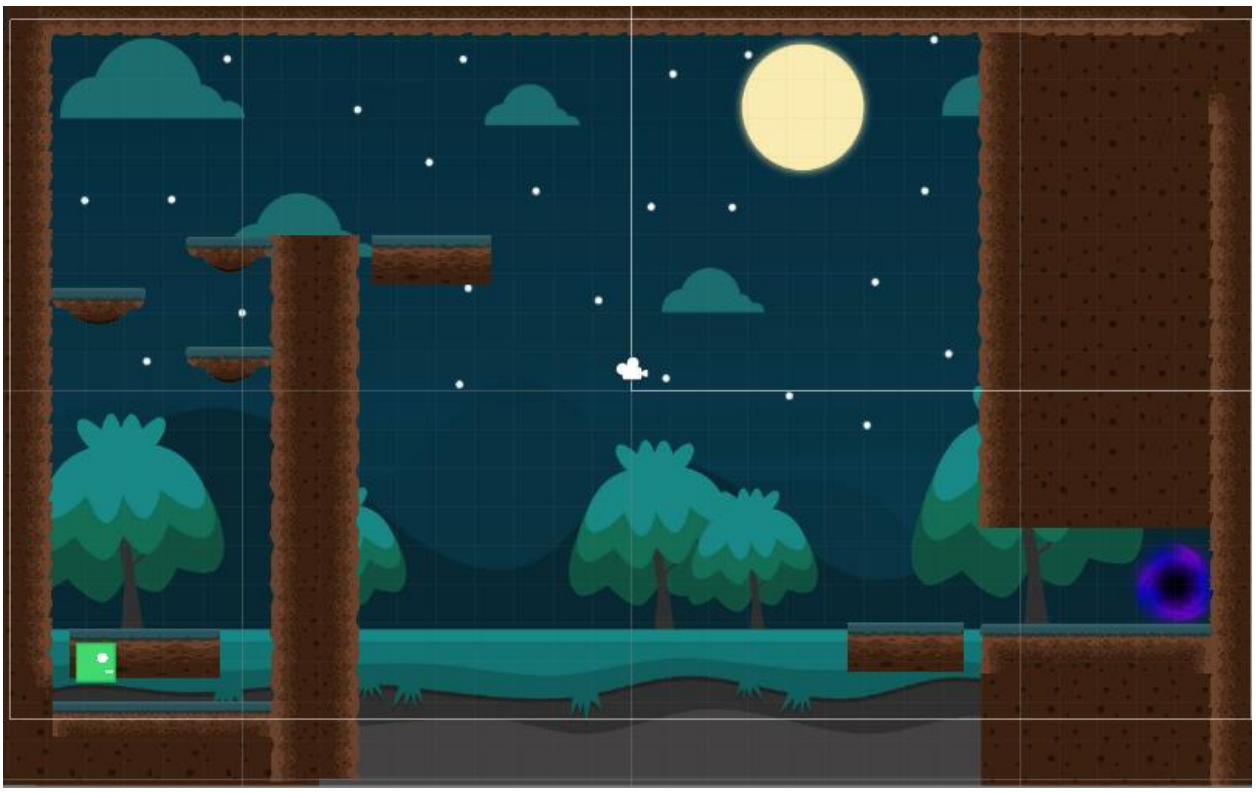
https://youtu.be/slWvATAq7fI

Part II: Final Report

Message of Acknowledgement

I would like to express my deepest appreciation for all the resources that others on the internet have shared for free to use helping me to complete this project.

SECTION 1: PROOF OF ENVIRONMENTS IMPLEMENTATION



[Acknowledgement – Free Graveyard Tileset from Game Art 2D, https://www.gameart2d.com/free-graveyard-platformer-tileset.html]

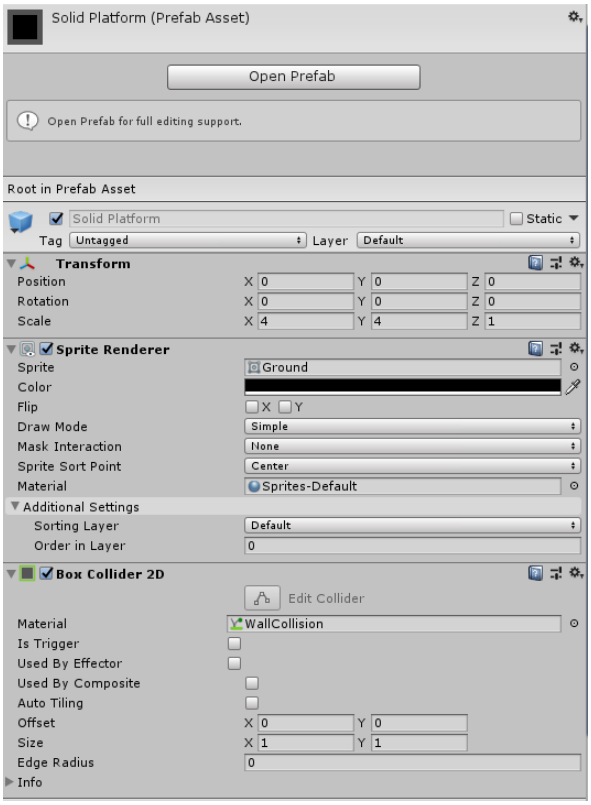
This is the terrain in level 1 of the game. It makes use of a group of Sprites I downloaded for free, which I modelled with three different types of platforms.

1. A solid platform
2. A 1-way platform (which I called action platform)
3. A moving platform (which also is a 1-way platform)

I put each of the 3 platforms as separate Prefa

1. The solid platform

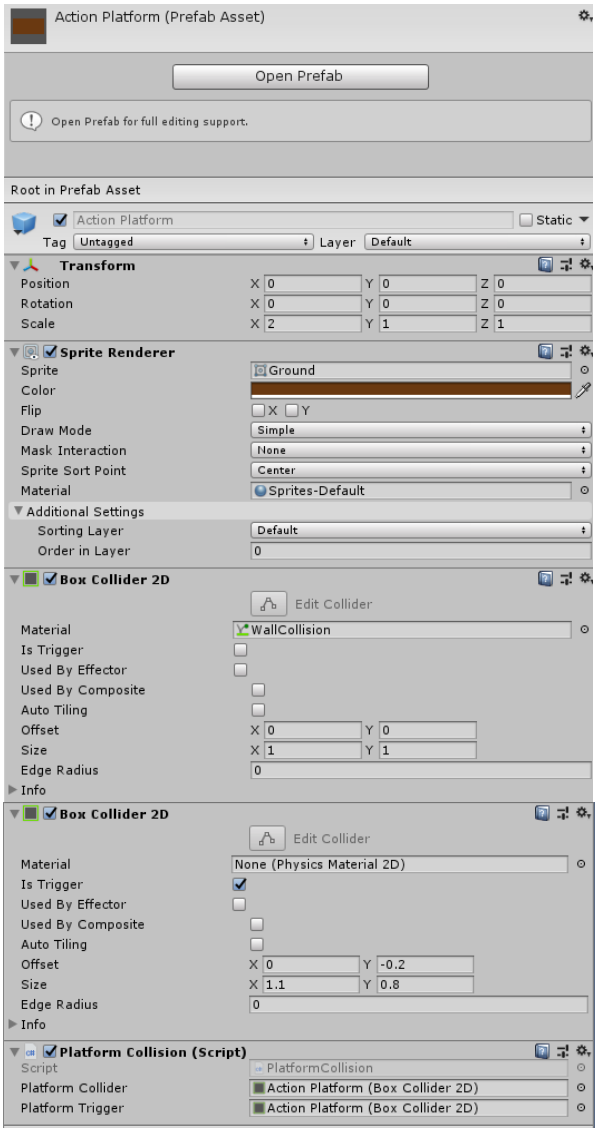
I used the solid platform for creating general terrain in the game. Since all that is required from general terrain is the BoxCollider2D component to prevent the Rigidbody2D game objects (i.e. player's avatar) from falling off, the borders as well as the floor is made of solid platforms.



The Sprite Renderer sprite has been changed according to their location with the asset skins from the Graveyard Tileset.

2. A 1-way platform/action platform

I used the 1-way platform as a method of letting the player's avatar to be able to jump into platforms instead of bashing into them when the avatar's tries to jump. It is one-way because after he jumps, he stays on the platform. I created these platforms to utilise space and more importantly, to allow the player to freedom to move around also making it possible to open opportunities for close contact platforms and other game objects, especially when the player needed to move back the in other direction



I achieved this by having 2 BoxCollider2D objects.
1 BoxCollider2D is a trigger allowing the player to pass through while the other BoxCollider2D is the normal one that prevents the player from passing

Also, all platforms have a Material: wallCollision to prevent the avatar from getting stuck in the wall by setting the friction to 0

I did this by using the script Platform Collision

The script Platform Collision

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

[RequireComponent(typeof(SpriteRenderer))]
[RequireComponent(typeof(BoxCollider2D))]
[RequireComponent(typeof(BoxCollider2D))]

0 references
public class PlatformCollision : MonoBehaviour
{
    private BoxCollider2D playerCollider;
    [SerializeField] private BoxCollider2D platformCollider;
    [SerializeField] private BoxCollider2D platformTrigger;

    0 references
    void Start()
    {
        playerCollider = GameObject.FindGameObjectWithTag("Player").GetComponent<BoxColl
        Physics2D.IgnoreCollision(platformCollider, GetComponent<BoxCollider2D>(), true)
    }

    0 references
    void OnTriggerEnter2D(Collider2D collider)
    {
      if(collider.gameObject.tag == "Player")
        {
            Physics2D.IgnoreCollision(platformCollider, playerCollider, true);
        }
    }

    0 references
    private void OnTriggerExit2D(Collider2D collider)
    {
        if(collider.gameObject.tag == "Player")
        {
            Physics2D.IgnoreCollision(platformCollider, playerCollider, false);
        }
    }
}
```

In the script we assign the Player's (avatar)'s BoxCollider2D as playerCollider.

We also make sure that the two platform BoxCollider2D do not collide with each other.

When the BoxCollider2D which is the closest to the ground (which has the Is Trigger on) is triggered by the collision of the Player's avatar, it tells to ignore the BoxCollider2D of the platform above it while the player is still within the region.

However, when the player leaves the region, the normal effect of the BoxCollider2D resumes as the IgnoreCollision() function is set to false.

This is why the BoxCollider2D with the Is Trigger is slightly below the radius of the normal BoxCollider2D, because otherwise the player would trigger it off while on the platform making the player fall down

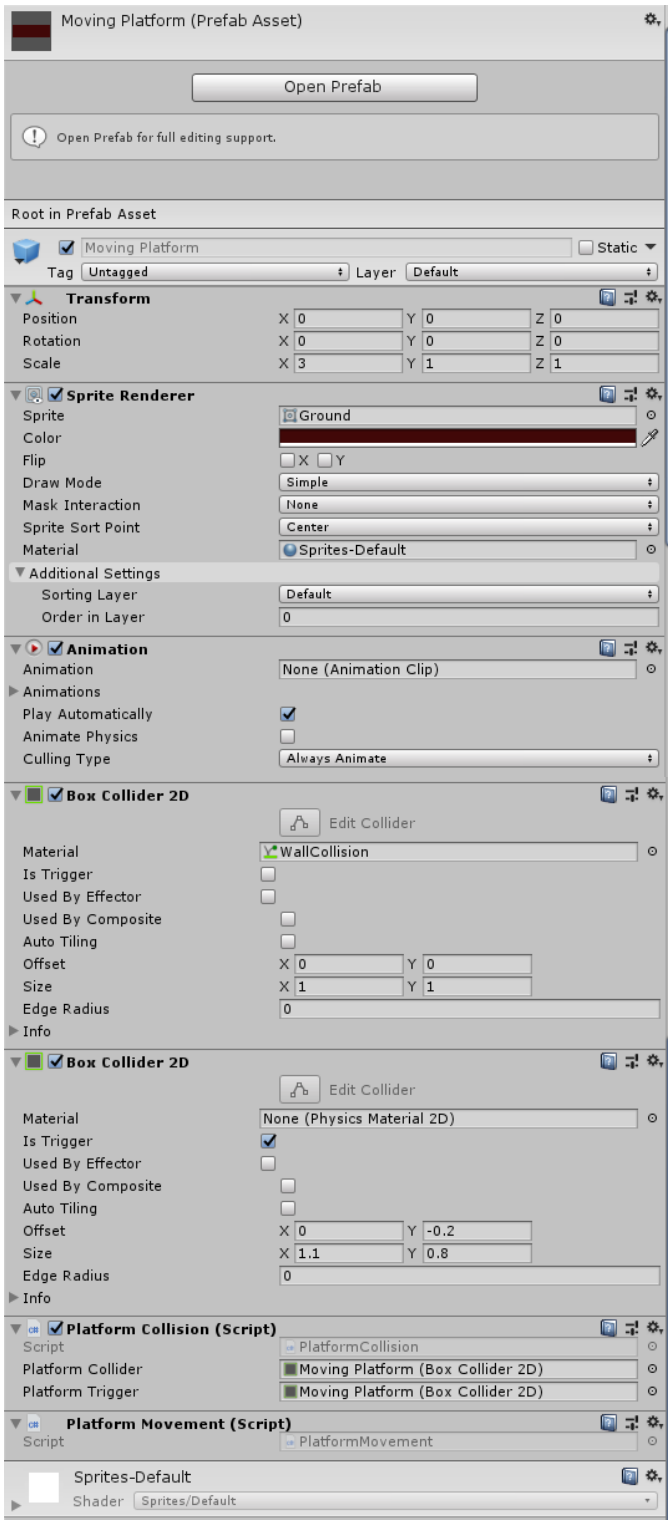
3. The moving platform

I used the moving platform as a convenient method to move a player from one location to another as well as increasing the difficulty by requiring the player to time their actions well to land on the moving platform safely.

I achieved this by adding an Animation on top of code that was in the action platform. There is also another script called Platform Movement which aims to keep the player on the platform instead of dropping off as it moves

Since this is a prefab, it doesn't have an animation clip on it but usually you either create/use an animation clip that allows you to move from one location to another in an adjustable amount of time

There was an alternative via script, however, the animation tool is very precise and can have multiple amount of points without needing to input more variables manually in a script



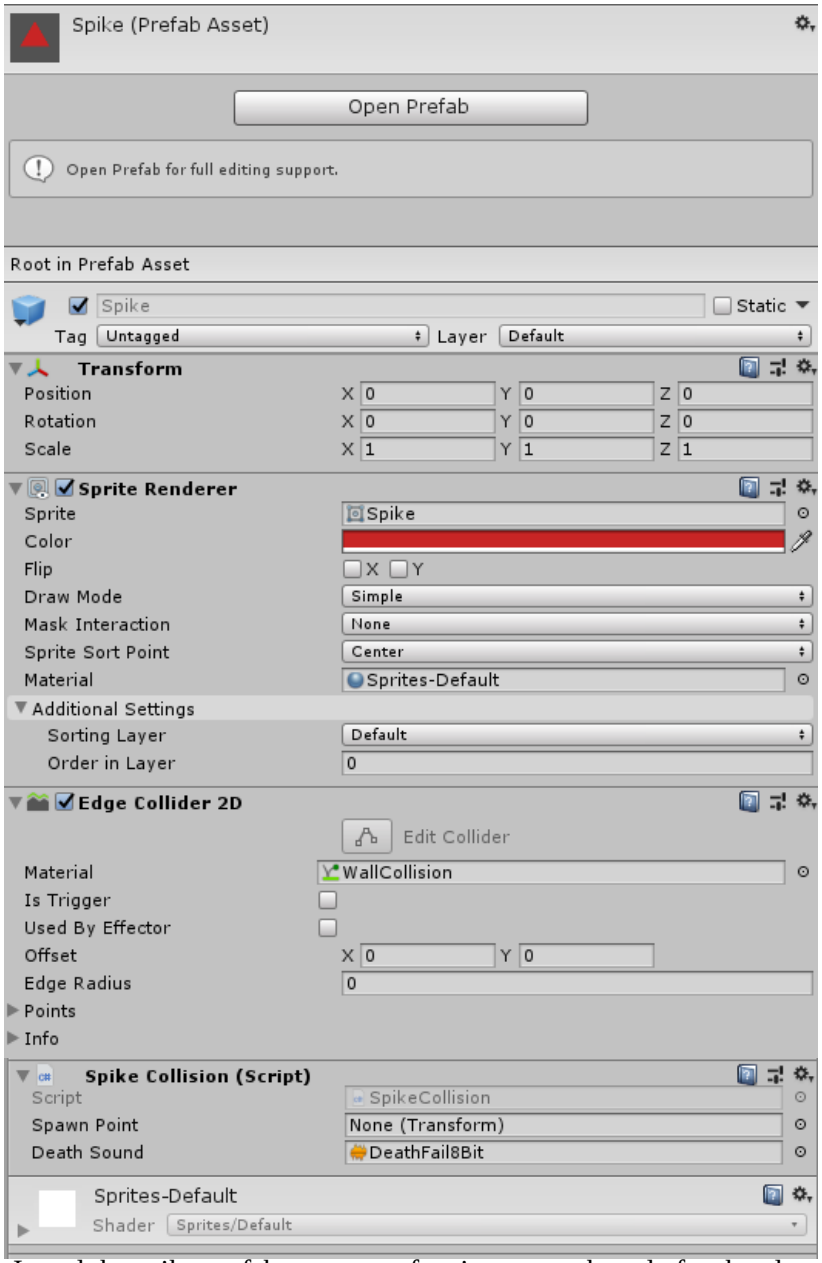
Script for Platform Movement

```
public class PlatformMovement : MonoBehaviour
{
    0 references
    void OnCollisionEnter2D(Collision2D other)
    {
        if(other.gameObject.tag == "Player")
        {
            other.transform.SetParent(this.transform);
            //make Player's transform a child of Platform's transform

        }
    }

    0 references
    void OnCollisionExit2D(Collision2D other)
    {
        if(other.gameObject.tag == "Player")
        {
            other.transform.SetParent(null);
        }
    }
}
```

In this script, I made it so that when the player collided with the moving platform, it would check if the game object really was the player (via the player tag) and in the case it was, the moving platform would act as a parent for the player's transform meaning that since it's the parent, any time it moves the child (player) would move proportionally the same amount. This effectively drags the player with the moving platform until the player moves off the platform in which the moving platform no longer is the parent transform of player

These 3 platforms are essentially the make up of the walkable terrain in the program.
There is also the spike that make up the terrain, however, instead of acting as a platform, they act as player killers, making the player reset the position back to its spawn point

Spike Prefab



I used the spike prefab as a way of acting as an obstacle for the player. Player would need to avoid the spike, otherwise it causes them to die and respawn back in their starting position of the level. Because of the shape of a spike (which is triangle in this case), instead of a BoxCollider2D, an EdgeCollider2D was used instead. It contains the script Spike Collision which is responsible for the respawning of the player.

Script Spike Collision

```
public class SpikeCollision : MonoBehaviour
{
    [SerializeField] private Transform spawnPoint;
    public AudioClip deathSound;
    0 references
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if(collision.gameObject.tag == "Player")
        {
            SoundManager.instance.PlaySingle(deathSound);
            collision.gameObject.transform.position = spawnPoint.position;
        }
    }
}
```

Ignoring the Sound, the method checks that when contact is made, it checks if it's the player who made contact and if so, sets the position (transform) of the player to the position of the spawnPoint which can be inputted (and therefore can also be used for checkpoints)

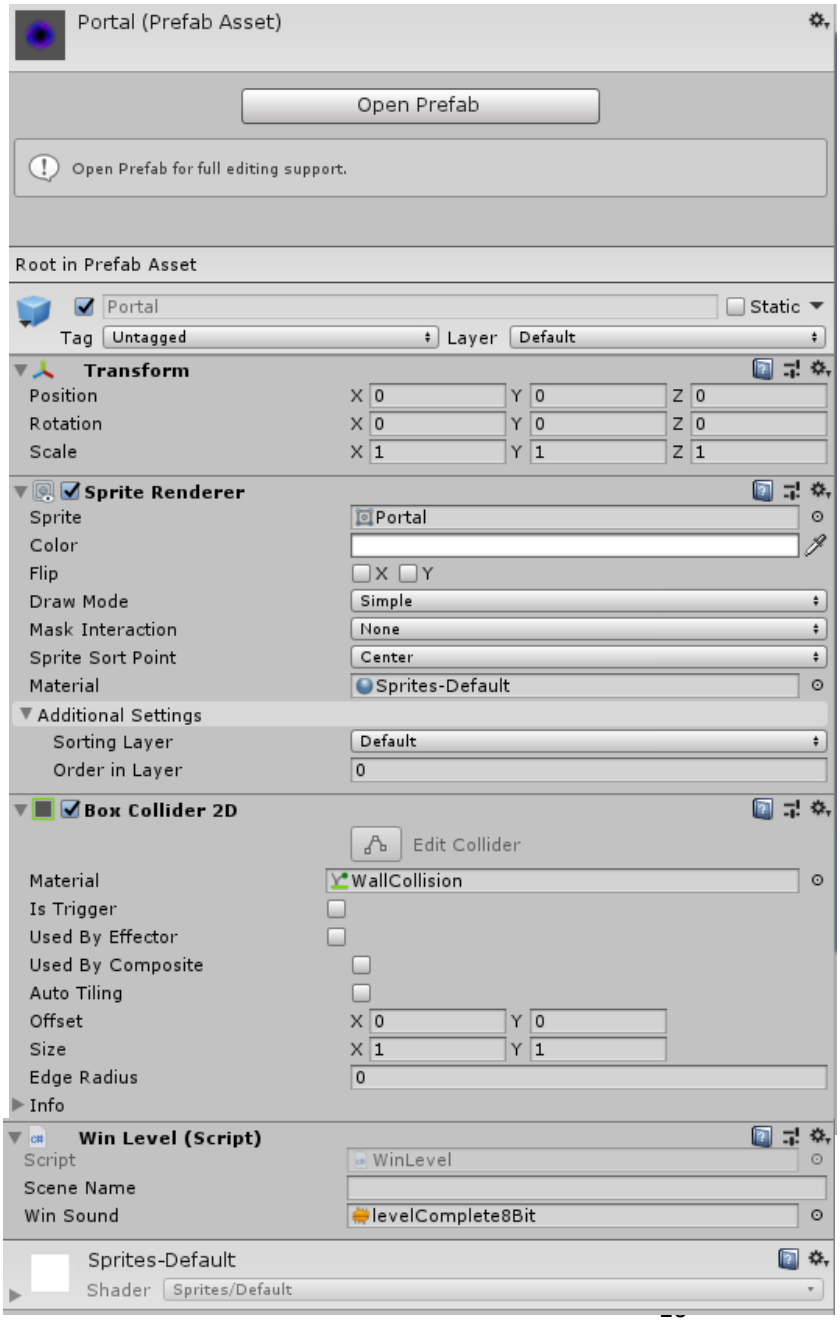
What I added

- Portals

Why I added it? (In reference to environments)

1. I added portals as a form of transitioning to different environments/levels
2. To provide a goal for the player to traverse the map in order to reach

How does it accomplish it?



- The portal has a BoxCollider2D allowing game objects to collide with it, this allows the player to come in contact with it
- It also contains a Script called Win Level which responsible for the code for transitioning of the scene

```
using UnityEngine.SceneManagement;

0 references
public class WinLevel : MonoBehaviour
{
    [SerializeField] private string sceneName;
    public AudioClip winSound;
    0 references
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if(collision.gameObject.tag == "Player")
        {
            SoundManager.instance.PlaySingle(winSound);
            SceneManager.LoadScene(sceneName);

        }
    }
}
```

- In the code it makes use of the library SceneManagement which allows you to use the SceneManager to transition to each a different scene
- We created a [SerializeField] for choose the sceneName as it more convenient to see the text at face value rather than opening the code everytime
- The method triggers on Collision and checks if the game object is player, and if it is, loads the scene name inputted in the [SerializeField]

SECTION 2: PROOF OF GAMEPLAY FEATURES

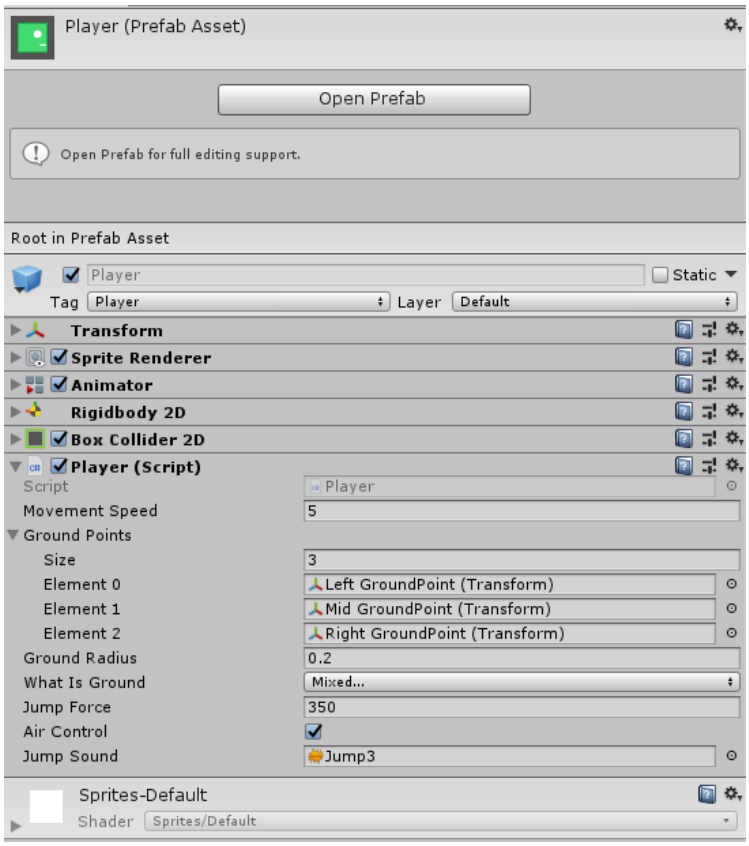
What I added:

- Horizontal movement

Why I added it? (In terms of Gameplay features)

- A part of the player controls to realise gameplay

How does it accomplish it?



The Player prefab contains a Rigidbody2D which is required for gravitational pull. It contains BoxCollider2D to prevent it from falling off a platform and lastly a “Player” script which is responsible for making it possible to move horizontally

```
void FixedUpdate()
{
    isGrounded = IsGrounded();
    float horizontal = Input.GetAxis("Horizontal");
    //Debug.Log(horizontal);

    UserInput();
    Movement(horizontal);
```

```
private void Movement(float horizontal)
{
    if(isGrounded || airControl)
    {
        rb2D.velocity = new Vector2(horizontal * movementSpeed, rb2D.velocity.y);

        if (isGrounded && jump)
        {
            isGrounded = false;
            SoundManager.instance.PlaySingle(jumpSound);
            rb2D.AddForce(new Vector2(0, jumpForce));

        }
    }
}
```

- The FixedUpdate() function allows this code to be repetitively
- Allows the Input.GetAxis("Horizontal"); method to call which will return either -1 or 1 depending on left or right and put the value into the float horizontal
- The value is sent to the Movement(value) method
- Ignoring the jump elements of it, it creates a new vector2(which the x value will be the value multipled by the movementSpeed) while the y value remains the same which applies a force to the Rigidbody2D to move ultimately according to the key inputted

What I added:

- Jumping mechanism

Why I added it? (In reference to Gameplay features)

- A part of player control to realsie gameplay

How does it accomplish it?

- It uses the same components as walking horizontally (no point of prefab)

```
[SerializeField] private Transform[] groundPoints;
[SerializeField] private float groundRadius; //coll
[SerializeField] private LayerMask whatIsGround; //
private bool isGrounded;
private bool jump;
[SerializeField] private float jumpForce;
```

These are the variables that are required for the jump operation

```
void FixedUpdate()
{
    isGrounded = IsGrounded();
    float horizontal = Input.GetAxis("Horizontal");
    //Debug.Log(horizontal);

    UserInput();
    Movement(horizontal);
    Flip(horizontal);
    ResetValues();

}
```

1. First, it checks if the player is on the ground via isGrounded = IsGrounded();

```
private bool IsGrounded()
{
    if(rb2D.velocity.y <= 0) //if velocity.y <= 0 means you're falling
    {
        foreach(Transform point in groundPoints) //checks if groundPoint is colliding
        {
            Collider2D[] colliders = Physics2D.OverlapCircleAll(point.position, ground

            for(int i = 0; i < colliders.Length; i++)
            {
                if(colliders[i].gameObject != gameObject)
                {
                    return true;
                }
            }
        }

    }
    return false;
}
```

2. It checks the Rigidbody2D to see if velocity.y <= 0 meaning that you're falling or on the ground
3. It uses 3 groundPoints as insurance that you're on the ground providing they collide
4. After it analysing that you are indeed on the ground it returns a boolean value
5. It gets the UserInput()

```
private void UserInput()
{
    if(Input.GetKeyDown(KeyCode.UpArrow))
    {
        jump = true;
    }
}
```

6. Checks to see if the player has pressed the key associated with jump (in this case it's the up arrow)
7. If they have, it sets the jump to true (from false)

```
private void Movement(float horizontal)
{
    if(isGrounded || airControl)
    {
        rb2D.velocity = new Vector2(horizontal * movementSpeed, rb2D.velocity.y);

        if (isGrounded && jump)
        {
            isGrounded = false;
            SoundManager.instance.PlaySingle(jumpSound);
            rb2D.AddForce(new Vector2(0, jumpForce));

        }
    }
}
```

8. It checks that the player is indeed on the ground (for different purpose)
9. Then it checks that both the player is on the ground and the jump button has been pressed
10. If that is the case, then it adds a jump force to the position.y of the Rigidbody2D causing it to jump up

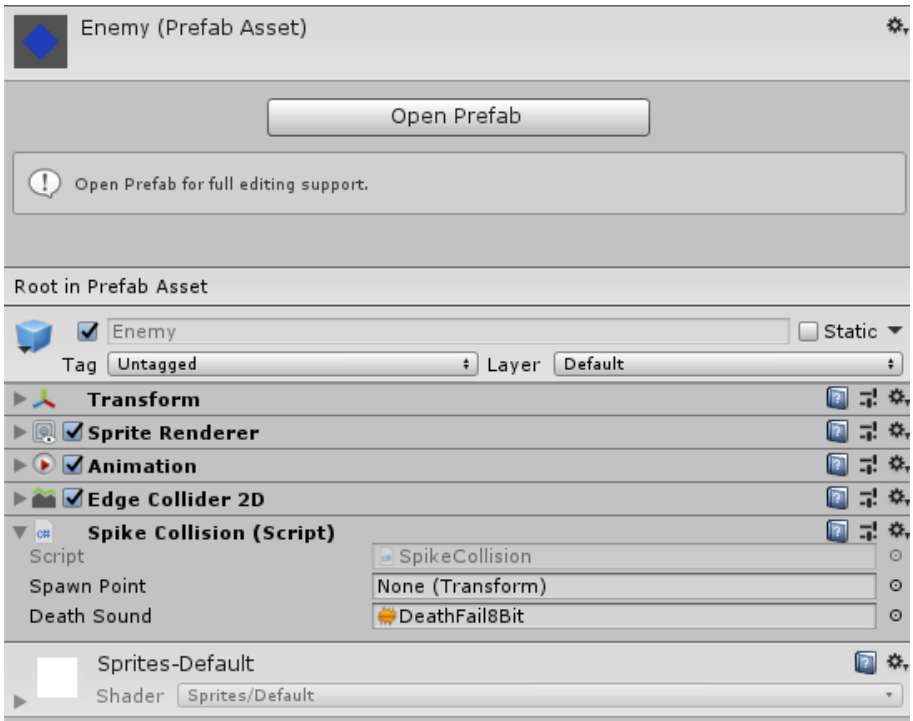
What I added:

- Enemy minions

Why I added it? (In terms of Gameplay features)

- Interaction between the player and entities

How does it accomplish it?



- The enemy uses an animation to move about
- The enemy uses a EdgeCollider2D (since it's a diamond)
- The enemy has a script Spike Collision (which was a script explained before that it checks that the game object is player then sets the position of the player to the spawnPoint)

Essentially, the different between the spike and the Enemy is that the Enemy has an animation meaning that it can move around in a specified location whereas the spike is static. Having a moving enemy does make the game more engaging since not only is it more difficult to predict when to avoid it but also because the speed can vary between each enemy.

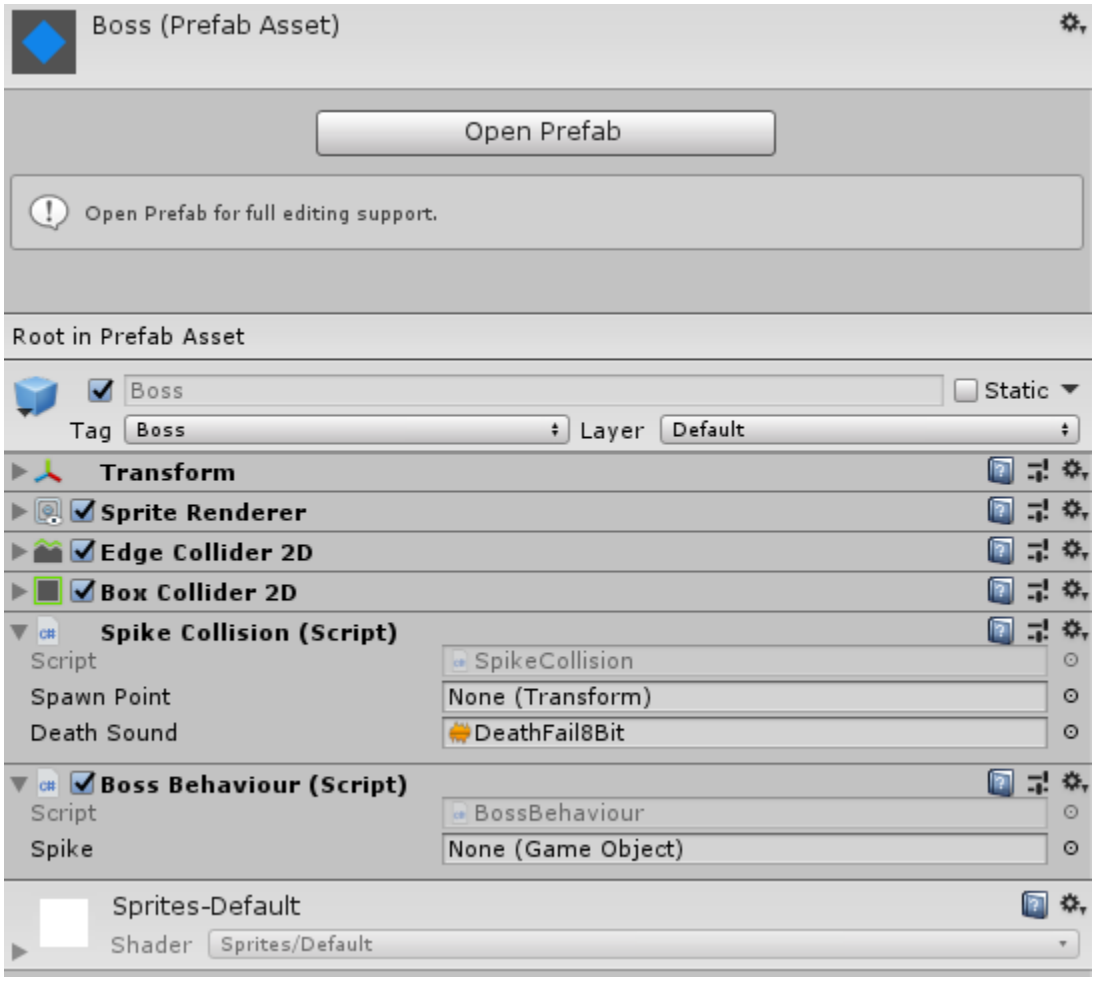
What I added:

- Boss

Why I added it? (In terms of Gameplay features)

- Interaction between the player and entities

How does it accomplish it?



The boss is special since it's the start of type of AI

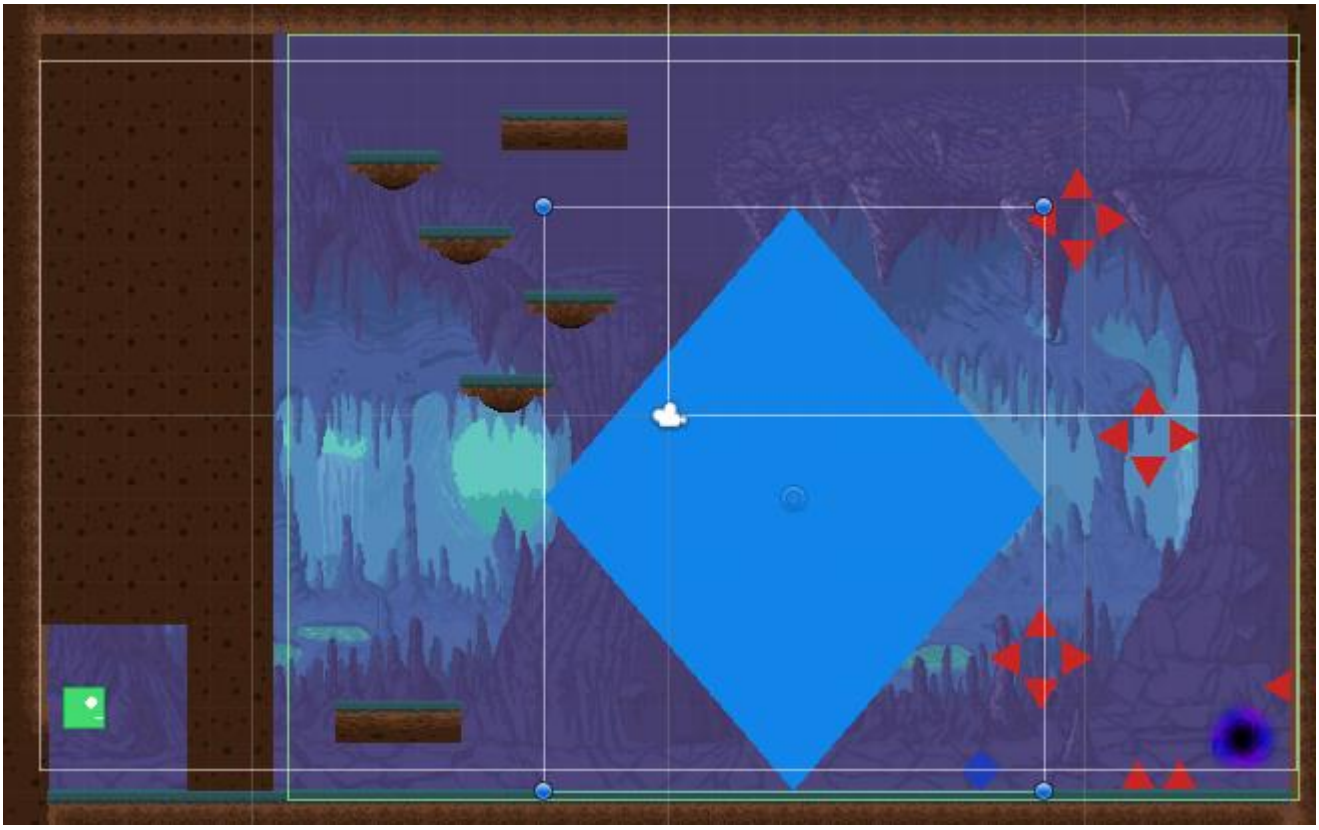
- It has the plain characteristics of an enemy
- However, it has it's own set of behaviours

The boss essentially triggers 3 things:

1. Acts as a massive enemy
2. Shoots out projectiles according to the player's location
3. Acts as a trigger for spawning the door behind the player (to seal him in)

1 has already been covered, just imagine the scale factor larger

Focus on 2:



The green line is the BoxCollider2D radius which acts as a trigger to these other two events

```
private void Start()
{
    door = GameObject.FindGameObjectWithTag("Door").GetComponent<BoxCollider2D>();
    door.gameObject.SetActive(false);
    trigger = false;
}
```

1. To begin with, trigger is set to false

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if(collision.gameObject.tag == "Player")
    {
        trigger = true;
        door.gameObject.SetActive(true);
    }
}
```

2. When the player enters the radius of the ColliderBox2D, it will detect whether it is the player and make trigger to true

```
private void OnTriggerExit2D(Collider2D collision)
{
    if(collision.gameObject.tag == "Player")
    {
        trigger = false;
        door.gameObject.SetActive(false);
    }
}
```

3. The opposite appllies if the player leaves the radius but that's only possible if he dies and that's due to reason 3

```
void FixedUpdate()
{
    if (trigger)
    {
        delay -= Time.deltaTime;

        if(delay < 0)
        {
            target = GameObject.FindGameObjectWithTag("Player").transform;
            direction = new Vector2(target.position.x, target.position.y);

            GameObject temp = (GameObject)Instantiate(spike, transform.position, Qua
            temp.GetComponent<ThrowSpike>().Initialize(direction);
            delay = 2;
            Destroy(temp, 2);
        }
    }
}
```

So FixedUpdate() regularly loops this method checking if trigger == true

If it is it will start it's attack on the player

- Delay is just what it means, a time delay for the if loop
- There is a default of 2 seconds for delay but after it's finished we can enter the loop

4 It gets the position of player and stores it in target

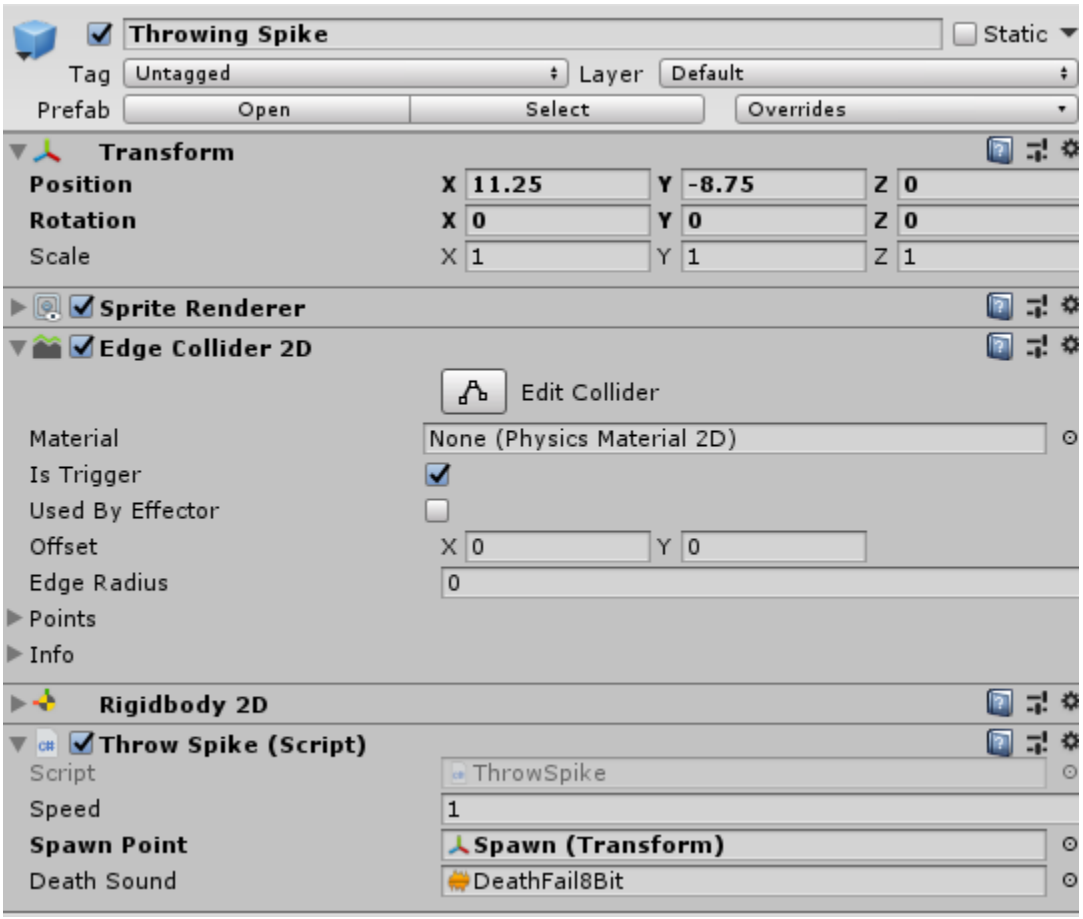
5 Using the transform, it creates a new Vector2 to find which direction its projectile should go and stores it in the variable direction

6 It creates an ThrowSpike game object from it's body of origin

7 It directs the ThrowSpike object to the direction of its enemy (the player)

8 It sets up another delay for 2 seconds so there is not a constant stream of thrownspikes thrown and the thrownspike is destroyed 2 seconds so it doesn't lurk in the background

- A ThrowSpike is slightly different to a usual spike because it uses a Rigidbody2D for movement as well as uses an OnTrigger BoxCollider2D as opposed to the OnCollision BoxCollider2D methods



```
public class ThrowSpike : MonoBehaviour
{
    [SerializeField] private float speed;
    private Rigidbody2D rb2D;
    private Vector2 direction;
    [SerializeField] Transform spawnPoint;
    public AudioClip deathSound;

    0 references
    void Start()
    {
        rb2D = GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    0 references
    void FixedUpdate()
    {
        rb2D.velocity = direction * speed;
    }

    1 reference
    public void Initialize(Vector2 direction)
    {
        this.direction = direction;
    }

    0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            SoundManager.instance.PlaySingle(deathSound);
            collision.gameObject.transform.position = spawnPoint.position;
        }
    }
}
```

- The code of Throwing Spike using Rigidbody2D and a public Initialize() function to allow the Boss game object to create objects of it

Focus on 3

- The last thing it does it is to activate a solid platform acting as the wall/door when the player enters the collision radius using door.gameObject.setActive(false) and (true)
- This prevents the player from leaving the fight after entering the fight and keeps it so that the Boss does not continuously attack the player when the player respawns

SECTION 3: PROOF OF NON-PLAYABLE FEATURES

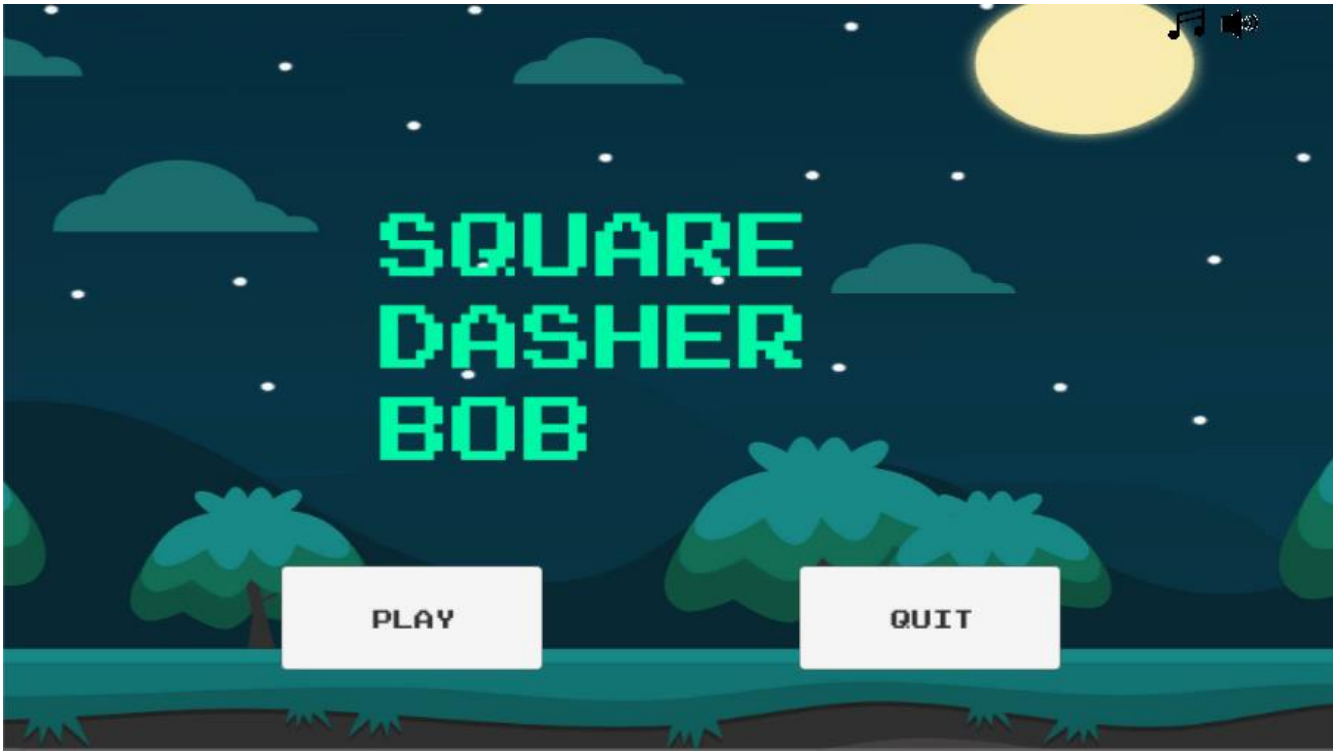
What I added:

- Main Menu

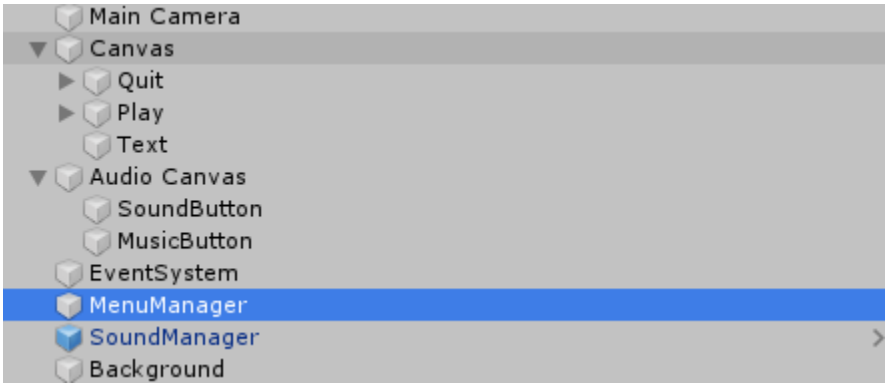
Why I added it

- Introduction to the game, also used to return back

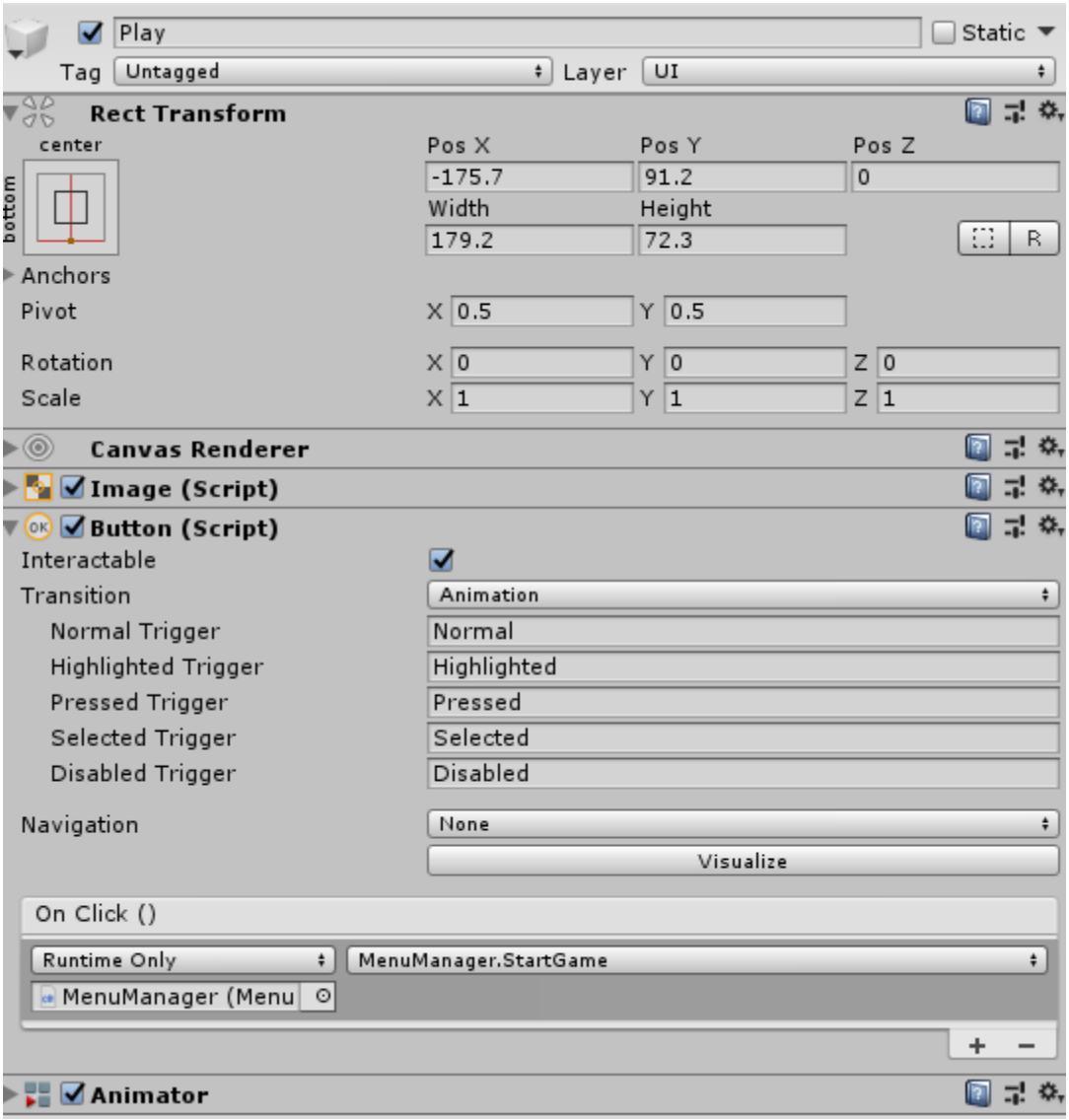
How does it accomplish it?

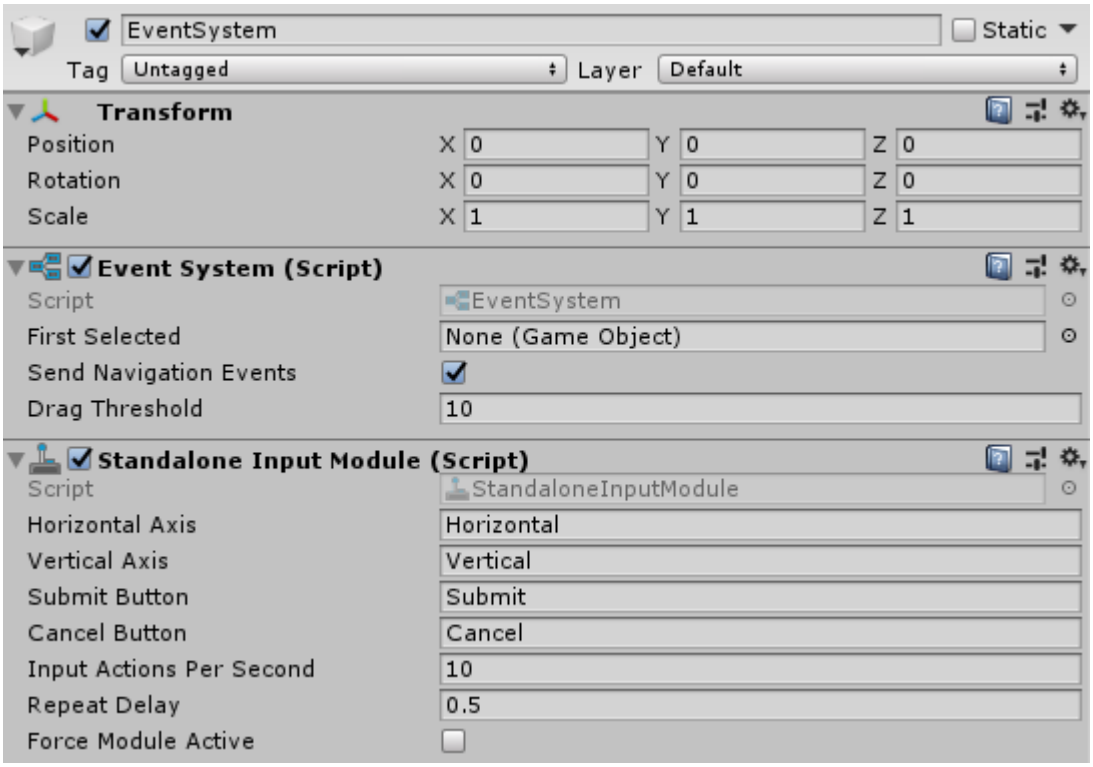


Introduces the players to the game, has two buttons for navigation “play” and “quit” which play takes you to level 1 scene while quit exits the application. There are also buttons on the top for toggleability.



There are a lot of components in this Main Menu but the main required for moving to the next scene will be the Canvas with the buttons and the EventSystem object that will manage the actions when they are clicked and the code from the MenuManager






```
using UnityEngine.SceneManagement;

0 references
public class MenuManager : MonoBehaviour
{


    0 references
    public void StartGame()
    {
        SceneManager.LoadScene("Level 1");
    }

    0 references
    public void QuitGame()
    {
        Application.Quit();
    }

}
```

The methods can be invoked when you pass the MenuManager to Button script allowing you to go to the next scene, in this case level 1. You need the include the EventSystem or else it won't work.

What I added:

- End screen

Why I added it

- Shows user that the game has ended

How does it accomplish it?

The message implies that the game has reached it's end and the indication of the Main Menu button. The Navigation to the main menu will be similar to the how the Start Menu went to Level 1.

SECTION 4: PROOF OF AUDIOVISUALS

What I added:

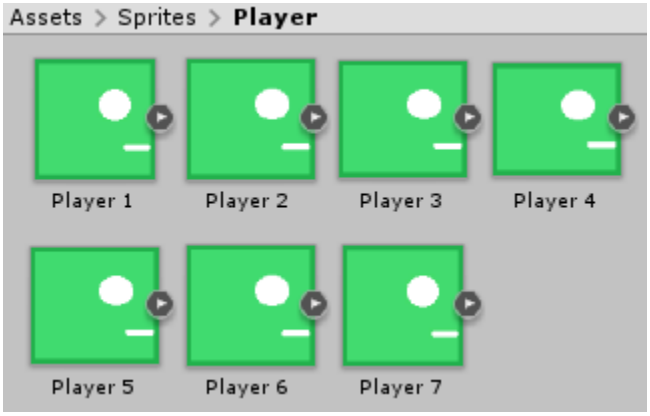
- Player idle animation

Why I added it:

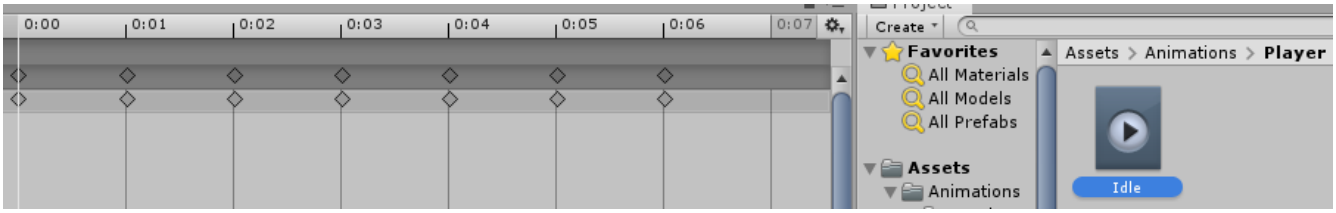
- Visually pleasant effects

How does it accomplish it?

The combination of several sprites pulled together into the heirarchy creates an animation for your sprite



Before



After (the images keep looping)

What I added:

- Sounds for actions and music

Why I added it:

- Helper to see if actions have be executed, adds another dimension to the program, satisfying

How does it accomplish it?

```
public class SoundManager : MonoBehaviour
{
    public AudioSource efxSource;
    public AudioSource musicSource;
    public static SoundManager instance = null;

    0 references
    void Awake()
    {
        if (instance == null)
        {
            instance = this;
        }
        else
        {
            Object.Destroy(gameObject);
        }
        DontDestroyOnLoad(instance);
    }
```

The variables allow you to input the sounds and music that you want by dragging onto the SerializeVariable

The if statement creates an instance of SoundManager if there isn't any

- Since it's possible to return back to the main menu, you need to check that there aren't any duplicate copies
- By using the DontDestroyOnLoad() method allows it to be used for the rest of the levels on

```
public void PlaySingle(AudioClip clip)
{
    efxSource.clip = clip;
    efxSource.Play();
}
```

Makes it so that when clicked, it will play the sound effect.

```
public void ToggleSound() ////true = muted, false = unmuted
{

    efxSource.mute = !efxSource.mute;


}

1 reference
public void ToggleMusic()
{
    musicSource.mute = !musicSource.mute; //true = muted, false = unmuted
}
```

All three of these are examples of Manager-type methods that you can insert into the buttons script to and execute these functions on click

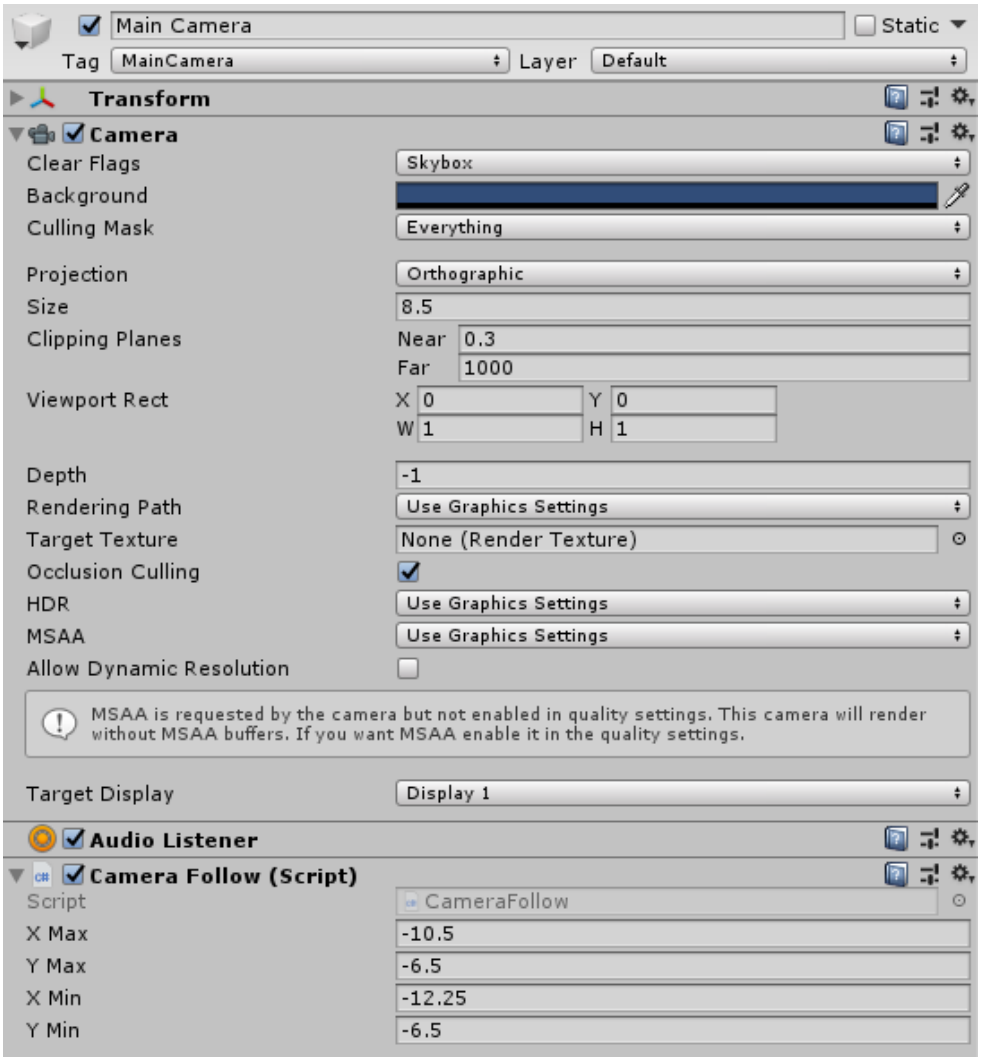
What I added:

- Changing camera view

Why I added it

- Dramatic effect on visual view

How does it accomplish it?



```
public class CameraFollow : MonoBehaviour
{
    [SerializeField] private float xMax;
    [SerializeField] private float yMax;
    [SerializeField] private float xMin;
    [SerializeField] private float yMin;
    private Transform target;
    private Camera cam;

    0 references
    void Start()
    {
        target = GameObject.FindGameObjectWithTag("Player").transform;
        cam = Camera.main;
    }

    0 references
    void LateUpdate()
    {
        transform.position = new Vector3(Mathf.Clamp(target.position.x, xMin, xMax), Ma

        if(target.position.x > -9.5)
        {
            cam.transform.position = new Vector3(0, 0, -10);
            cam.orthographicSize = 10;
        }
    }
}
```

```
Mathf.Clamp(target.position.y, yMin, yMax), transform.position.z);
```

Essentially, the code sets up the max and min values that you want to your camera to not go pass (e.g. don't look pass the border). It sets up a target using the transform of the player. It then loops the following code:

- Setting the position of the camera to the position of target (the player)
- The camera is restricted by the xMin, xMax, etc. concealling a lot of details in the map
- It checks if the position of the x is over -9.5 meaning that the door (of the boss room) has been passed and at that point switches it back to it's original location so that the full map
- This combination of two different cameras as the element of surprise as well as indicates that something serious/special will happen considering that the camera has been fixed the during the other levels in the game)

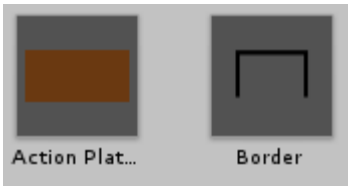
What I added:

- Skins for the platforms

Why I added:

- Visually appealing

How does it accomplish it?



The original set of prefabs converted to

The difficulty of creating platform objects is that they collided with each other in different ways meaning that different sprites were appropriate for different situations. There was an an attempt to prefab some combinations that popped by but it soon was too inconsist to use efficientively

How the game is played:

The concept is very simple. Using the arrow keys/AWSD to move around (with W being jump) you have to attempt to reach the portal game object by platform hopping be sent to next level 2. Difference obstacles will attempt to cause you to restart, however, there is no life restrictions. That is all.