VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Khanh Hong

DEVELOPING A 2D PLATFORM ENDLESS RUNNER GAME WITH UNITY ENGINE

School of Technology
2023

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Khanh Hong
Title	Developing a 2D Platform Endless Runner Game with Unity Engine
Year	2023
Language	English
Pages	41
Name of Supervisor	Jani Ahvonen

The topic of this Bachelor's thesis is about the making of a 2D platform endless runner game using the Unity Engine. This thesis involves the designing and developing of various aspects of the game, including level design, character movements, obstacle design, procedural generation, and graphics. This thesis also provides an overview of 2D platform endless runner games and the making of it by introducing the tools and resources for creating this type of game. By breaking down its process, this thesis will provide the details of what step is needed to create a 2D platform endless runner game.

The result product for the thesis is a prototype game called "Kaizo", a 2D platform endless runner. The game was mostly completed within the available predetermined time, with the only missing feature being the background music and sound effects. Additional playtesting will be needed to further balance the game mechanics, mostly from the player's movement such as speed, jump force and gravity.

Keywords Unity, 2D Platformer, endless runner, and procedural generation.

CONTENTS

LIST OF FIGURES AND TABLES

1 INTRODUCTION

Endless runner games are undoubtedly one of the most well-known and played games in the mobile game scene, many famous titles could be named such as Temple Run, Temple Run 2 and Subway Surfer. These games feature a main playable character who is constantly running forward, while avoiding obstacles, with the goal of surviving as long as possible.

Similarly, 2D platformer is one of the most well-known and played genre in the gaming scene in general, with the most famous title being the famous plumber: Mario, and another nostalgic game of many 90s kids: Maple Story.

The aim of this thesis is to create and introduce the steps to create a 2D platform endless runner game, a genre that combines endless runner and 2D platformer, using the Unity Engine. By the end of this thesis, the reader will have a basic understanding of the Unity Engine, what it takes to create a game in general, and what is needed for the creation of a 2D platform endless runner game.

2 RELEVANT TECHNOLOGIES

This chapter introduces all the used technologies in the final product, a general description of the technologies, what purpose they serve to this thesis, and the reason they are picked for the final product.

2.1 Unity Game Engine

A game engine is a software tool that is designed specifically for game development and is optimized and configured in a way that simplifies and optimizes the process of game development. They usually come with support for different programming languages, 2D and 3D graphics renderers, a physics engine to simulate real-world physics in game, and sound engine to handles all sound-related aspects of a game, as well as many features such as animations, multiplayer support, and many more. (Arm, n.d.).

The chosen game engine for this project is the Unity Engine, developed by Unity Technologies. Unity is a game engine, which was created in 2005, that supports both 2D and 3D game development. They were created to provide a game development tool for game developers to access easily in the game development path. Over the years, the engine has made significant improvements to keep up with the latest practices and technologies (GameDev Academy, n.d.).

The Unity Game Engine was chosen for a few main factors. Because this is a simple 2D game, it would be wise to choose a Game Engine that is simple to use, beginner friendly, suitable for the development of a simple game. Unity meets all these criteria, and with the huge popularity of Unity, community support is a huge factor, in providing instant support and feedback in problem situations.

Unity mainly uses C# as its main scripting language, but it also allows its users to use JavaScript instead, making it highly suitable for almost every coder, since JavaScript is one of the most popular languages there is.

Access to the Unity Asset Store is one of the main reasons for the selection of the game engine for this thesis. The Unity Asset Store provides various game assets and sprites, paid or free, and can be imported and implemented directly into the game.

All the above reasons combined, along with previous experience of the author with Unity, makes it a very suitable, reasonable, and time-saving game engine for this thesis product.

2.1.1 Unity Asset Store

Unity Asset Store is a place where creators and game developers can provide and access game assets easily through a digital store. Game creators can create game faster with premade asset that is provided, and in return, game is created with less complexity as well. Premade asset also can be used as a placeholder while you create your game, leaving the burden of creating an asset behind while you are working on the logic and focus on the actual gameplay to implement a prototype as fast as possible. (Unity, n.d.).

In this thesis, a free Unity Asset Store package, called “Pixel Adventure 1”, was used for the purpose of providing assets and sprites for the levels and character that are created in the final product.

2.2 Version Control – Git – GitHub.

Version control, in software engineering, also known as a source code management, is a tool for controlling changes that were made to a program. Changes are stored usually as a number or letter code, known as a “revision”, and are associated with the person who made the changes, along with the timestamp at which the changes have been made. Revision can be compared, rewind, and combined with other revisions. (Wikipedia, n.d.).

Git is a distributed version control system and is widely use amongst coders to track system changes in the development of a software team. Its main feature include being very fast and support parallel work running on multiple machines. (Atlassian, n.d.).

GitHub is a hosting service for software and version control development using Git. It allows working with Git, along with bug tracking, software feature request, tasks management, and wikis for individual projects. It also allows hosting for open-source software projects and is currently the most used source code host as of November 2021, with over 100 million developers and 372 million repositories, with 28 million being public (Wikipedia, n.d.).

The reason for this combination of version control is that they are easy to get started on, and they are so popular and well-known, along with their huge user base, that every problem encountered will already have a fix to them.

2.3 JetBrains Rider

JetBrains Rider is a .NET IDE with strong support for languages used in .NET development, including C#, VB.NET, F#, ASP.NET syntax, XAML, XML, JavaScript, TypeScript, JSON, HTML, CSS, and SQL, along with many powerful tools that helps you develop code quickly and better. (Jetbrains, 2022).

The reason that JetBrains Rider is the chosen IDE for this thesis was its capability to integrate with Unity. It provides many additions feature when using in combination with Unity, including quick fixes, context actions and inspections, code completion, syntax highlight, debugging with Unity, running Unity tests, and many more. (Jetbrains, 2022).

Another reason is its integration with version control systems. JetBrains Rider includes a full-fledged VCS client that supports all major version control systems. It enables working with Git right on the Rider IDE without the need for another software. In this case, the Rider IDE was used to handle Git version control, the chosen version control for this thesis.

2.4 C#

C# is a high-level programming language that supporting multiple paradigms, static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines (Wikipedia, n.d.).

C# was chosen because it is the official language that Unity supports. C# works in Unity as a scripting language. Scripting tells the game objects how to behave, it is the scripts and components attached to the game objects, and how they interact with each other, that creates the gameplay.

3 APPLICATION DESCRIPTION

The objective of the thesis was to create a 2D platform endless runner game, using the above discussed technologies. In this chapter, the vision and development plan for the final product will be discussed.

3.1 Game Design and Objectives

The game design for this game was straightforward, combining two genres: “2D platformer” and “endless runner” together.

A platform game is a type of action video game where the main goal is to move the player character from one point to another in a given environment. These games have levels with different types of terrain and platforms at different heights. To progress, the player needs to jump and climb on these platforms. The challenge lies in maneuvering the character across these platforms, to reach a goal (Wikipedia, n.d.) Key takeaways point here is “to reach a goal”.

An endless runner, however, does not have a goal. An endless runner is a type of game where the player's character keeps running without stopping, and the goal is to survive and achieve a high score. The game generates the level or environment in a continuous and random way. Endless runners can be viewed from the side, top, or in 3D, but the player is always in a never-ending level where the character moves forward automatically. The player's only control is to make the character dodge obstacles by either moving or pressing buttons. (Wikipedia, n.d.). The genre exploded on mobile platforms following the success of Temple Run, with Jetpack Joyride, Subway Surfer, and Canabalt being other popular examples.

The objective of this thesis is to combine the two genres together, with the source of inspiration from Canabalt, to create a Mario-like environment for the main character to navigate through (the platformer aspect of this game), but without having to stop, or have an ending like various platformer game (the endless runner aspect of this game), and instead, through a way of level generation, can continues to play and survive, without falling behind the map or falling down out of the map, for the longest time possible.

3.2 Game Assets and Sprites

A game asset is a digital component used in video games, like models, textures, animations, sounds, music, and more. They are created by artists and developers to make the game look and sound better and provide the necessary elements for gameplay. Game assets enhance the overall gaming experience. (Unity, n.d.)

Sprites are 2D images used as game objects, while in 3D they are used as textures. In this thesis, when sprites are mentioned, it is implied that they are 2D sprites. (Unity, n.d.)

For this thesis, a free asset pack from the Unity Asset Store, mentioned in chapter 2.1.1, called "Pixel Adventure 1", was used for the purpose of the whole game theme. This package consisted of many different sprites, including 4 characters sprite, along with its respective animations; 12 different terrain tileset combinations; and many small elements such as traps, power ups, and boxes. The main elements from this asset pack used in the thesis includes a main character sprite, and its animation, to create a main character for the game and all its movement; 12 different terrain tileset, for the creation of obstacles, that are used for the main level generation method; and the background color set, to create a moving background behind the game as the main background.

3.3 Level Generation

Being an endless runner game, world generation pre-game is not an idea choice, since technically speaking, there is not a way to generate "endless" or "infinity". With pre-generation is not a possible choice, another more sensible option, being the most approached method amongst the endless runner genre, is procedural generation, which is also the method of choice for the level generation of this thesis.

Procedural generation in computing is a way of creating data using algorithms instead of manual work. It involves using a mix of human-made assets, algorithms, and computer-generated randomness and processing power. In computer graphics, it's often used to make textures and 3D models. In video games, it's used to automatically generate a lot of content for the game. The benefits of procedural generation can include smaller file sizes, more content, and randomness for unexpected gameplay. (Wikipedia, n.d.).

In simple terms, procedural generation is when code is used to generate game content, like levels and characters, based on specific parameters. It allows games to create content that follows the rules of the game while being randomly generated. (LinkedIn, n.d.).

In the context of this thesis, levels, or obstacles, are being generated and destroyed continuously throughout the gameplay, with certain parameters within the obstacles itself, to create an endlessly generated environment for the player to navigate through endlessly, or until they fail, hence the endless runner genre.

3.4 Player’s Movement

The core objective of almost any platformer game is to move the player character between points in an environment, and they are characterized by levels that consist of uneven terrain and suspended platforms of varying height that require jumping to traverse. The most common movement options in the genre are walking, running, jumping, attacking, and climbing (Wikipedia, n.d.). For the development of this game, the author decided to keep the movement simple, and the core player’s movement mechanic of this game are running and jumping. To make up for the simplicity of the movement mechanic, the level design and generation method will be more complex.

4 IMPLEMENTATION

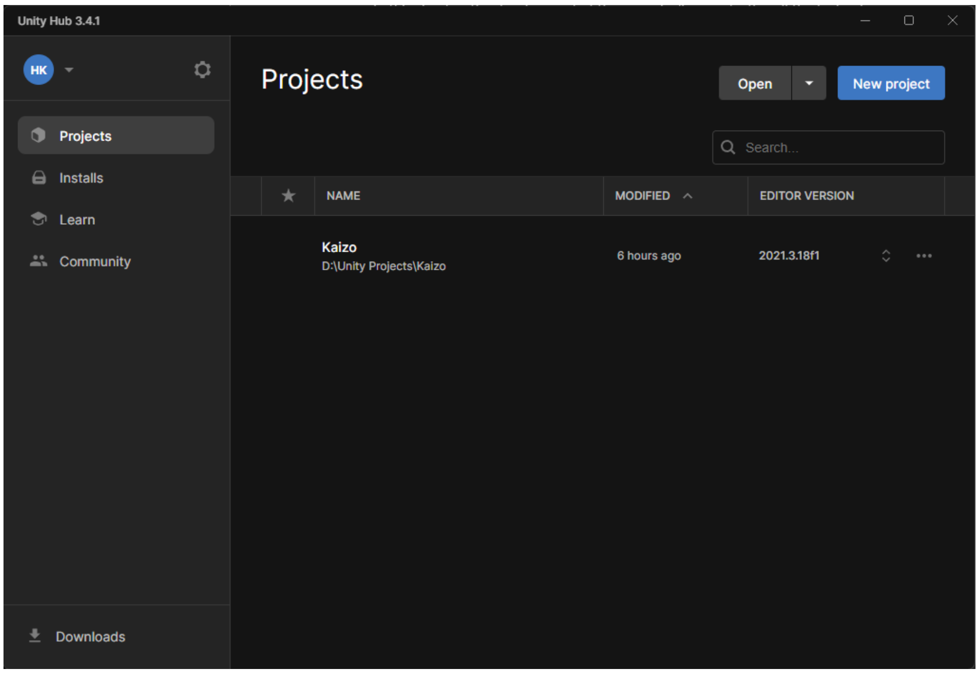
In this chapter, the development of the game is discussed, after all the technologies and game design choices are being chosen, along with its implementation methods. In this chapter, it is assumed that all the prerequisite technologies are already installed and set up.

4.1 Creating the Project

To create a game using the Unity Engine, one must first create a project in the Unity Hub, which is the main application to access the Unity ecosystem, including giving access to the user's projects, and to create new ones.

In Unity Hub, simply click "New project" to create a new project. A new template panel will pop up to let the user choose which template they want to work with, and since the aim is to make a 2D game, the chosen template would be "2D core". This template will give us all that we need to begin the development of a 2D game.

The input of the project name is also required in this step.



Kaizo
D:\Unity Projects\Kaizo
6 hours ago
2021.3.18f1

Figure 1. Unity Hub main interface

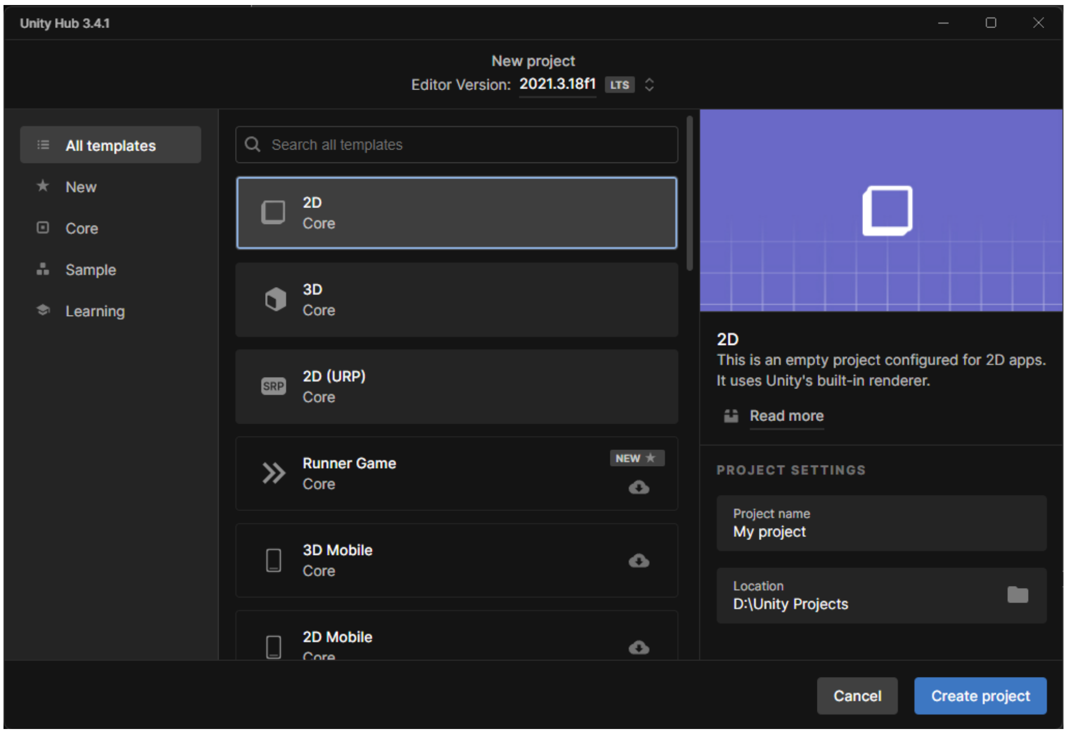


Figure 2. Template selection interface of the Unity Hub new project.

4.2 Folder Structure of the Project

The default folder structure of a Unity project consists of several different folders, each with its own purpose, but only two of those folders are actually being used and displayed in Unity itself, which is Asset and Package.

Asset is the main folder that every game developer is going to spend their most time in. Here lies all the assets that a game developer is going to use for their game, from imported or created models, characters sprite, audio files, back-grounds images, to scripts and prefabs that the developers are going to write and create. Upon creating a new project, this folder is empty.

The package folder is where every package that is installed with the project is stored, and they are controlled by the Unity Package Manager. The developer can ignore this folder, as every modification needed for the packages would be done using the Unity Package Manager.

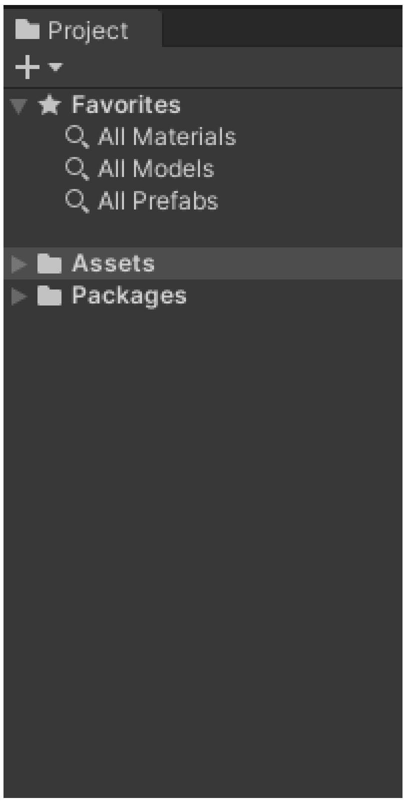
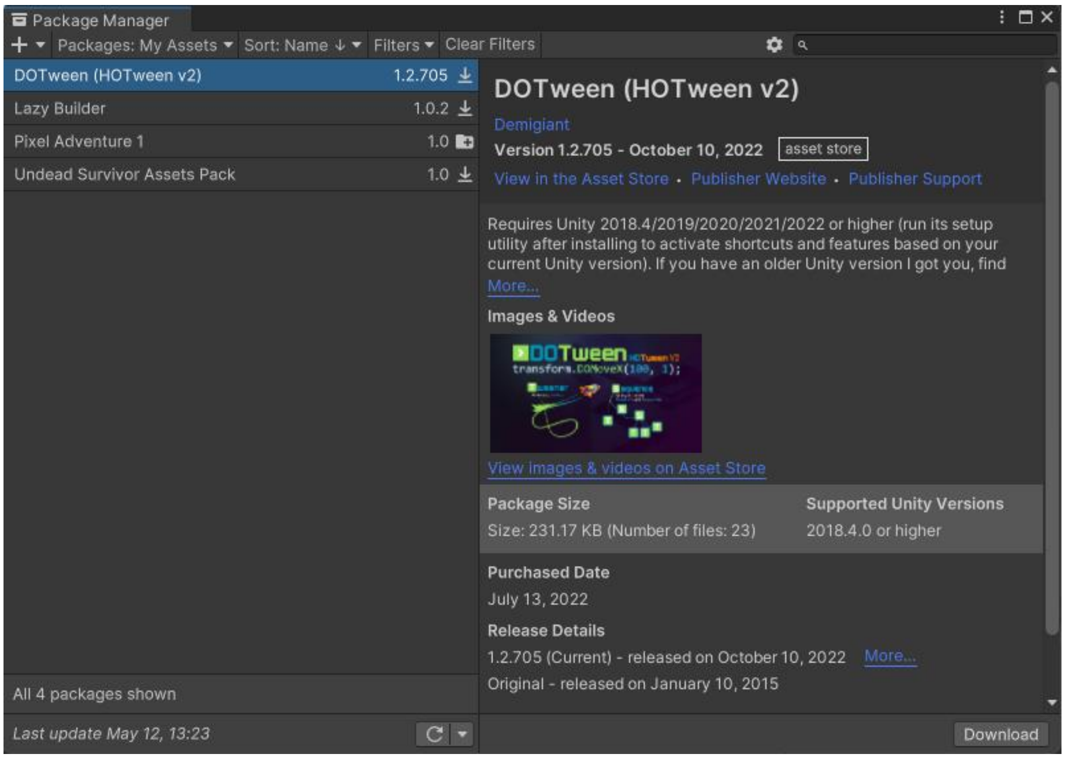


Figure 3. Initial folder structure of the project



DOTween (HOTween v2) 1.2.705
Lazy Builder 1.0.2
Pixel Adventure 1 1.0
Undead Survivor Assets Pack 1.0

All 4 packages shown

Last update May 12, 13:23

DOTween (HOTween v2)

Demigiant

Version 1.2.705 - October 10, 2022 asset store

View in the Asset Store · Publisher Website · Publisher Support

Requires Unity 2018.4/2019/2020/2021/2022 or higher (run its setup utility after installing to activate shortcuts and features based on your current Unity version). If you have an older Unity version I got you, find

More...

Package Size

Size: 231.17 KB (Number of files: 23)

Supported Unity Versions

2018.4.0 or higher

Purchased Date

July 13, 2022

Release Details

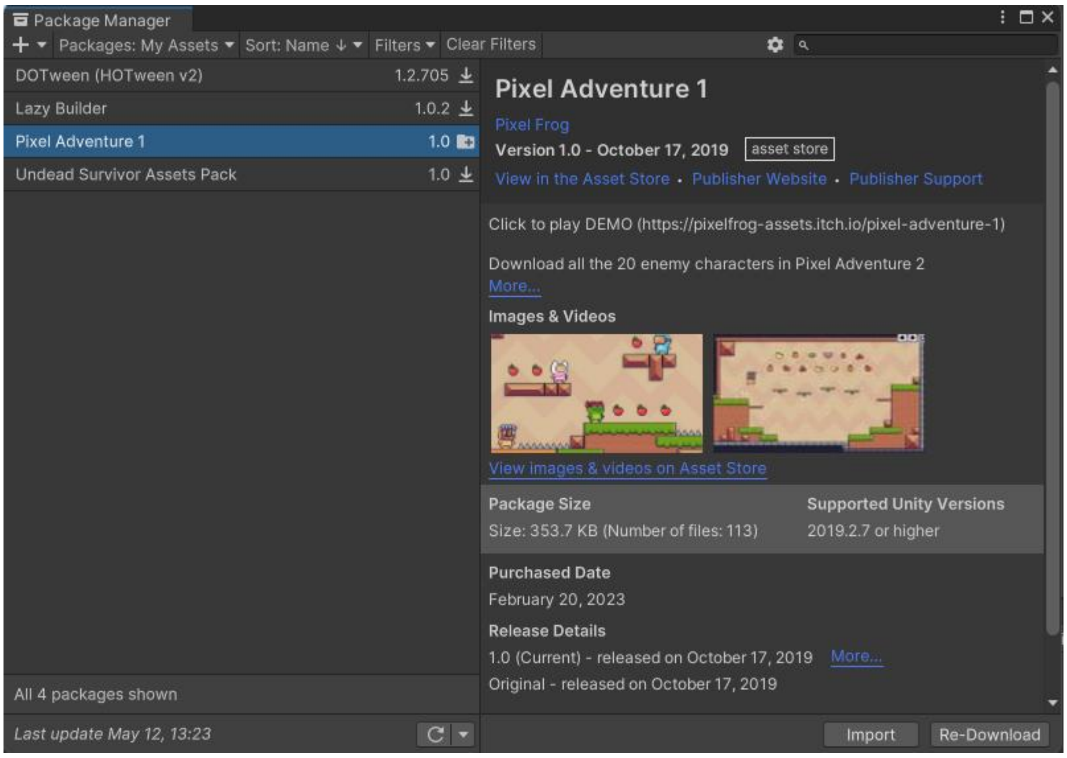
1.2.705 (Current) - released on October 10, 2022 More...

Original - released on January 10, 2015

Figure 4. Example of the Unity Package Manager interface

4.3 Importing Asset(s)

As discussed above, the asset pack that is used in this project is the “Pixel Adventure 1” package, from Pixel Frog. It is a free to use asset pack, and could be easily imported from the Unity Package Manger, either by adding the package in the Unity Asset Store to one’s Unity account, and import them from the “My Assets” section, or by searching the name directly onto the Unity Package Manager.



Click to play DEMO (https://pixelfrog-assets.itch.io/pixel-adventure-1)

Download all the 20 enemy characters in Pixel Adventure 2
More...

Package Size
Size: 353.7 KB (Number of files: 113)

Supported Unity Versions
2019.2.7 or higher

Purchased Date
February 20, 2023

Release Details
1.0 (Current) - released on October 17, 2019 More...
Original - released on October 17, 2019

Figure 5. Import of "Pixel Adventure 1"

After importing the asset pack, there should be a new folder created called “Pixel Adventure 1”, which contains everything from the asset pack itself, ready for us to use.

4.4 Developing the Game

The start of every development of a new game is the creation of a new scene. In this case, the main game scene where all the action is happening is created first. A new folder called “Scenes” in first created for structuring purposes, and a new

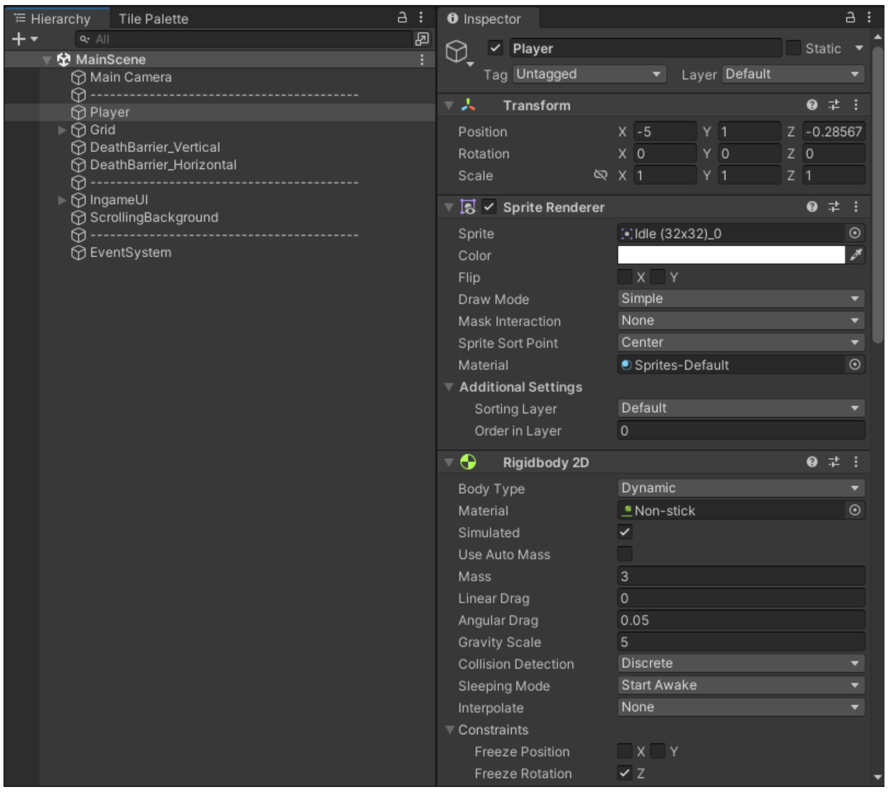
Scene inside of it named “MainScene” is created next. We now have a new scene template to start creating the game in.

4.4.1 Main character

The main character is first implemented by creating a new game object, called Player, by right clicking the Hierarchy window and Create New.

A component called Sprite Renderer is required to display the character’s sprite on the actual game. Simply select the Player, press Add component on the Inspector window, and choose Sprite Renderer. A character sprite from the Pixel Adventure 1 is required, simply drag the desired character onto the Sprite section on Sprite Renderer, and we now have a character ready to be worked with.

Repeat the step with another component called Rigidbody 2D, which gives the character the ability to interact with gravity in the Unity Engine, making it possible o move around, jump, and fall. A 2D box collider is also required to detect collisions.



```
MainScene
  Main Camera
  ---------------------------------------
  Player
  Grid
  DeathBarrier_Vertical
  DeathBarrier_Horizontal
  ---------------------------------------
  IngameUI
  ScrollingBackground
  ---------------------------------------
  EventSystem
```

```
✓ Player                          Static
Tag Untagged          Layer Default

Transform
  Position    X -5   Y 1   Z -0.28567
  Rotation    X 0    Y 0   Z 0
  Scale       X 1    Y 1   Z 1

✓ Sprite Renderer
  Sprite                 Idle (32x32)_0
  Color
  Flip                   X  Y
  Draw Mode              Simple
  Mask Interaction       None
  Sprite Sort Point      Center
  Material               Sprites-Default
  Additional Settings
    Sorting Layer        Default
    Order in Layer       0

Rigidbody 2D
  Body Type              Dynamic
  Material               Non-stick
  Simulated              ✓
  Use Auto Mass
  Mass                   3
  Linear Drag            0
  Angular Drag           0.05
  Gravity Scale          5
  Collision Detection    Discrete
  Sleeping Mode          Start Awake
  Interpolate            None
  Constraints
    Freeze Position      X  Y
    Freeze Rotation      ✓ Z
```

Figure 6. Player game object and its components

4.4.2 Player's Movement

The character is created but does not have the capability to move around and jump. The character's movement is added to the character by scripting and attach the script to the character as a component. This allows for the character to have the ability to move around when the user is pressing the arrow keys and jump when the user press space. The script also allows for the player's animation to change to a specific animation corresponding to what the character is doing at that time.

```
using UnityEngine;
using UnityEngine.UI;

public class PlayerMovement : MonoBehaviour {

    public float speed = 5f;
    public float jumpForce = 10f;
    [SerializeField] private Button replayButton;
    [SerializeField] private Button backButton;
    private bool isGrounded;
    private Rigidbody2D rb;
    private BoxCollider2D col;
    private Animator anim;

    void Start() {
        rb = GetComponent<Rigidbody2D>();
        col = GetComponent<BoxCollider2D>();
        anim = GetComponent<Animator>();
    }

    void Update() {
        float moveInput = Input.GetAxis("Horizontal");

        rb.velocity = new Vector2(moveInput * speed, rb.veloc-
ity.y);

        anim.SetFloat("Speed", Mathf.Abs(moveInput));

        isGrounded = IsGrounded();
        anim.SetBool("IsJumping", !isGrounded);

        if (isGrounded && Input.GetKeyDown(KeyCode.Space)) {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
        }
    }
    private bool IsGrounded()
    {
        RaycastHit2D raycastHit = Phys-
ics2D.Raycast(col.bounds.center, Vector2.down, col.bounds.ex-
tents.y + .3f);
        return raycastHit.collider != null;
    }
}
```

Figure 7. Player's movement script

A problem while developing is that if the player presses the space button continuously, the character will constantly be jumping midair and could be flying up off the map. To prevent this, a function called isGrounded() has been created, using raycast, to track whether the character is touching something below it, in this case, the ground. If the raycast detects nothing below the character, which means the character is now in midair, we disable the character’s ability to jump, and vice versa.

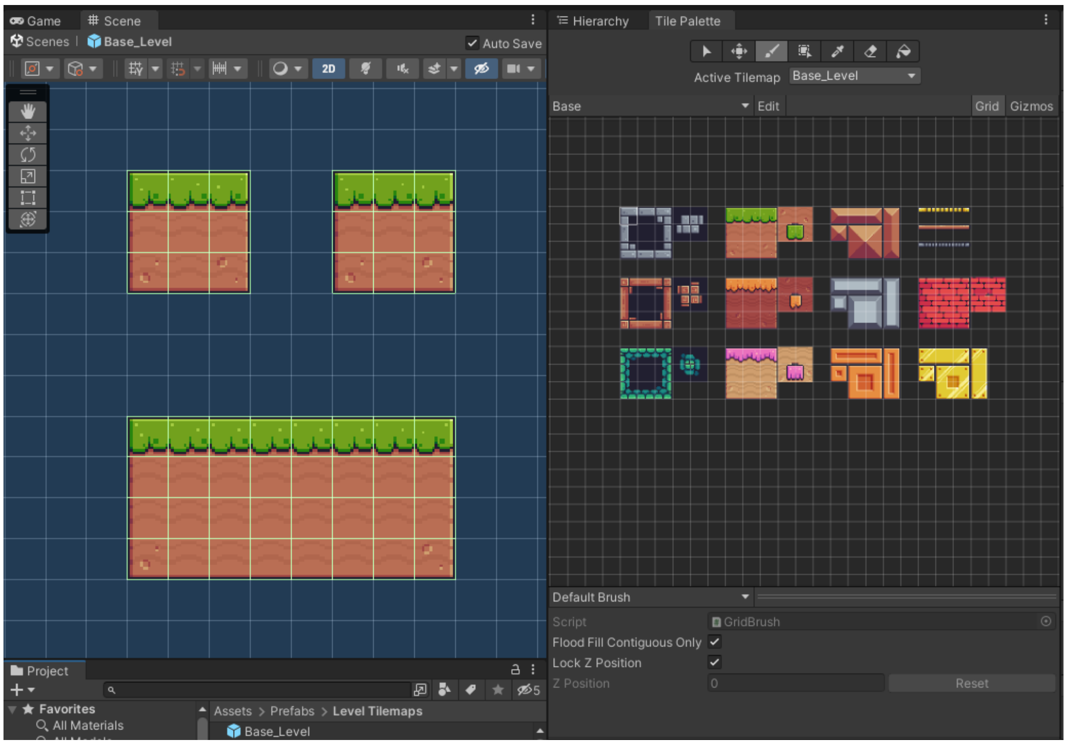
4.4.3 Obstacle Prefabs

Obstacles are created using Unity’s Grid and Tilemap system. The Grid component is a tool that helps align game objects, like tiles, in a chosen layout. It takes grid cell positions and converts them into the appropriate coordinates for the game object. The Transform component then converts these local coordinates to global or world space coordinates. (Unity, n.d.). Unity's Tilemap system is a feature that manages and stores tile assets for creating 2D levels. It simplifies the process of creating and iterating level designs within Unity, making it easier to design and modify game levels. (Unity, n.d.).

The tile assets that are used in this game are included in the Pixel Adventure 1 pack, with 12 different textures, that later is utilized to differentiate the three types of obstacles discussed above. They are used to create a new tile palette to create the obstacles with.

To create the obstacles that will be generated continuously for this game, a method of using various prefabricated pieces of the game world and stitching them together, conditionally, is used for the illusion of an infinitely long world.

Each obstacle prefab is an 8 x 10 area, with 8 being the width and 10 being the height of the obstacle. An obstacle will also have a few properties, such as the type of the obstacle, either Room, Maze or Sky, the opening position, and the ending position of it.



Active Tilemap Base_Level
Script GridBrush
Flood Fill Contiguous Only ✓
Lock Z Position ✓
Z Position 0

Figure 7. A base obstacle painted with tile palette.

Using the base obstacle prefab, along with the different terrain textures of the tile palette, 27 more obstacles have been created. There are 3 types of obstacles currently available in the game:

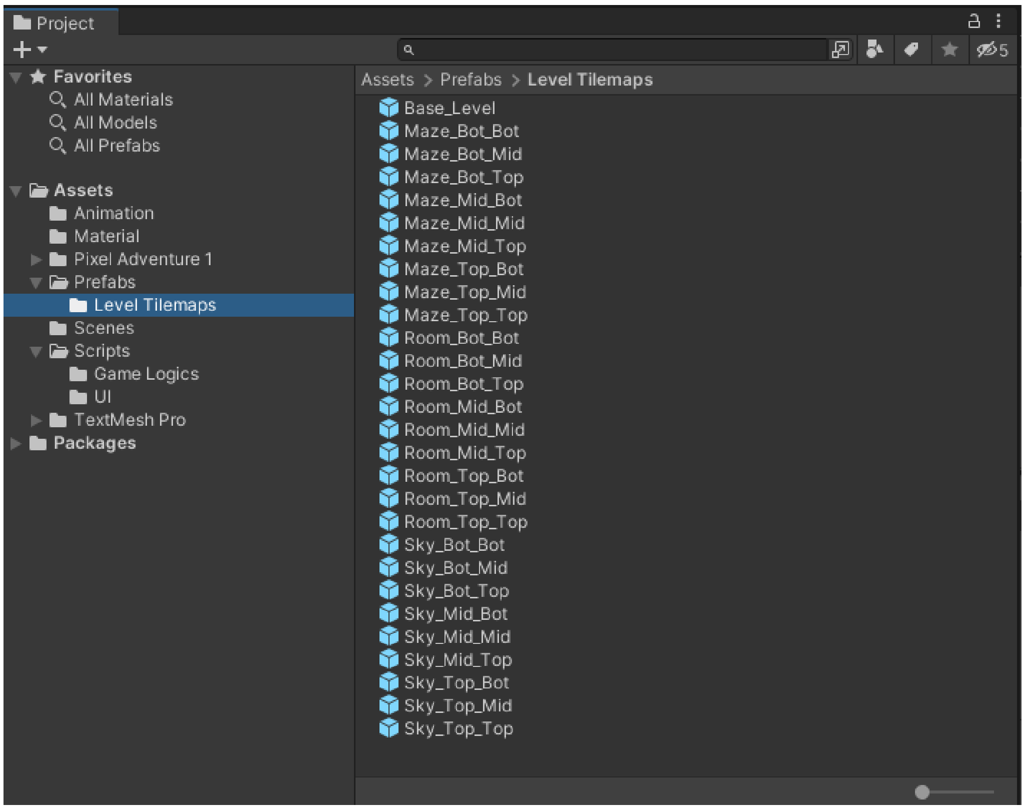
- Room: Room obstacles resemble a small room, very simplistic obstacle course that could be seen on many similar 2d platformers, like Mario. Imagine Room obstacles a small section of some random Mario world. Room levels are very simple to navigate through and do not require many skills from the player.

- Maze: Maze obstacles resemble a maze, characterized by multiple possible pathing, tight-navigated spaces, and some blocked paths. Maze levels are arguably the hardest type of room in Kaizo, requiring the player to solve the maze in the spot to find the right path to take, and being very tightly spaced, requires a lot of maneuver skill from the player as well.
- Sky: Sky obstacle is very similar to a Room obstacle, but instead, there is no support platform below the obstacle, and by being so, it creates another lose condition for the game, which is falling off the map. Sky levels are the medium difficulty obstacle of this game, requiring a bit more movement skill from the player, to not fall and ending the game.

Amongst the three types of obstacles, each of them has their own unique set of opening and ending position as well: Bot – Mid – Top.

- Bot: referring to the bottom position (opening or ending). The bottom position is located at the 3rd and 4th position from the bottom up out of the 10-height obstacle.
- Mid: referring to the middle position (opening or ending). The middle position is located at the 5th and 6th position from the bottom up out of the 10-height obstacle.
- Top: referring to the bottom position (opening or ending). The top position is located at the 7th and 8th position from the bottom up out of the 10-height obstacle.

Combining all these properties, 27 different and unique obstacles were created, to make the game feels as random and unique as possible, and to make sure that every new game a player plays will never feel the same, like they have played this game world before. To differentiate them, different naming was made for each of the obstacles, and it is rather simple: Obstacle type (either Room, Maze, or Sky) follows by its opening position (Bot, Mid or Top) and ends with its ending position. Another folder was created, named Prefabs, and inside it, Level Tilemaps, to store all of the Obstacle prefabs for ease of access later on.



Project

Favorites
- All Materials
- All Models
- All Prefabs

Assets
- Animation
- Material
- Pixel Adventure 1
- Prefabs
  - Level Tilemaps
- Scenes
- Scripts
  - Game Logics
  - UI
- TextMesh Pro

Packages

Assets > Prefabs > Level Tilemaps

- Base_Level
- Maze_Bot_Bot
- Maze_Bot_Mid
- Maze_Bot_Top
- Maze_Mid_Bot
- Maze_Mid_Mid
- Maze_Mid_Top
- Maze_Top_Bot
- Maze_Top_Mid
- Maze_Top_Top
- Room_Bot_Bot
- Room_Bot_Mid
- Room_Bot_Top
- Room_Mid_Bot
- Room_Mid_Mid
- Room_Mid_Top
- Room_Top_Bot
- Room_Top_Mid
- Room_Top_Top
- Sky_Bot_Bot
- Sky_Bot_Mid
- Sky_Bot_Top
- Sky_Mid_Bot
- Sky_Mid_Mid
- Sky_Mid_Top
- Sky_Top_Bot
- Sky_Top_Mid
- Sky_Top_Top

Figure 8. Prefabricated world obstacles

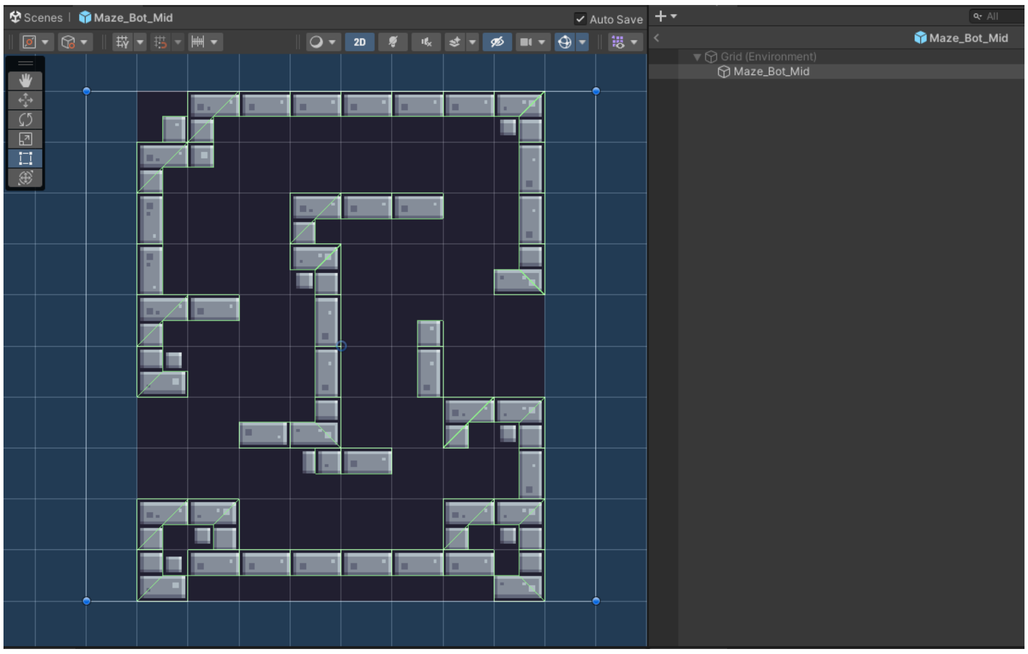


Figure 9. Example of an obstacle, a Maze with Bottom opening and Middle ending

Each of the obstacles is further given a component, called Tilemap Collider 2D, to help it interacts with the physics system, and can interact and collide with different GameObjects, in this case, the Player. With out this component, all the obstacles created above would be nothing but a picture.

Each of the terrain tile have its different texture, so custom physics shape for them is required. With out it, all the tiles will have their own collider that takes the whole tile, and no part of an obstacles will be moveable and can be navigated through. To achieve this, navigate to the original asset where the terrain tiles is stored, and open its Sprite Editor. A sprite editor of that terrain texture will be opened, and in the top left corner, select "Custom Physics Shape". Now each individual tile is ready to be drawn its physic shape.

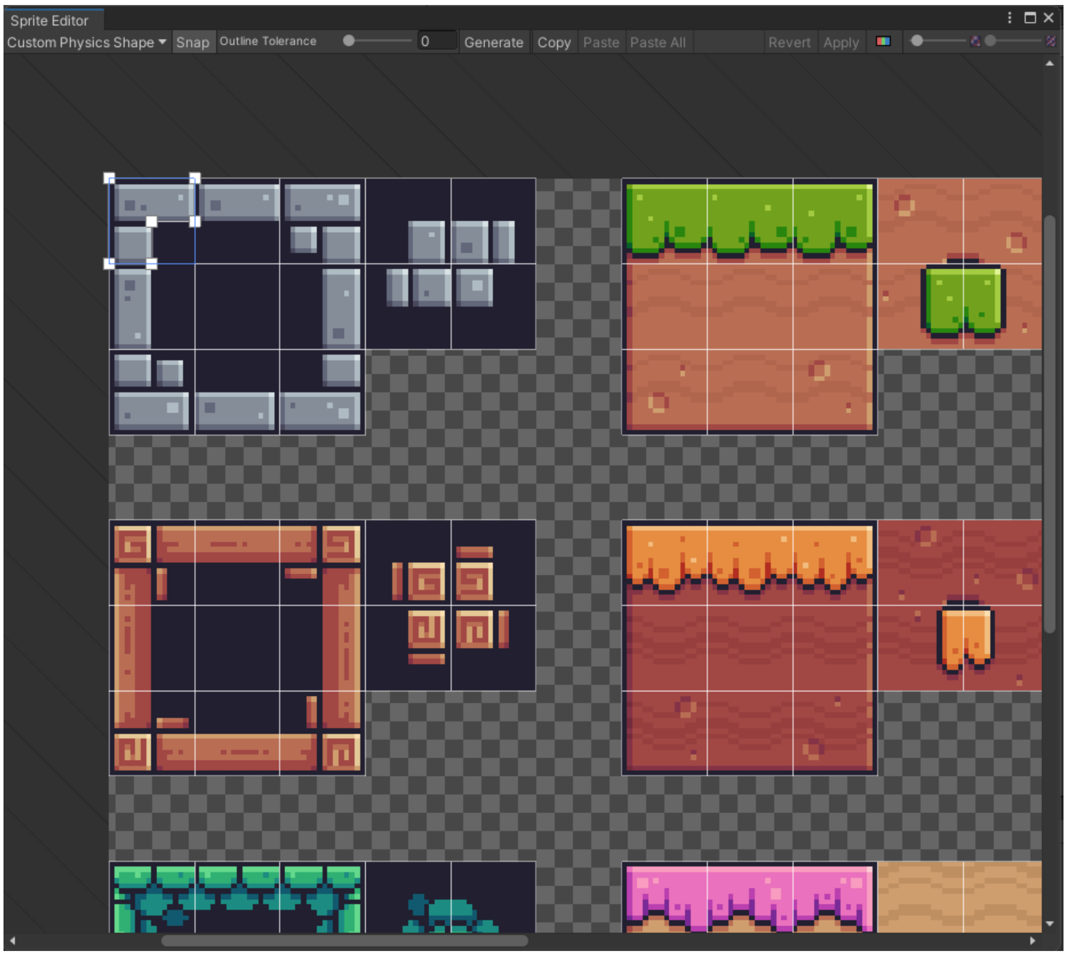


Figure 10. Example of a custom physic shape drawn using Sprite Editor

It is good to note that the dark middle part of the terrain just represents an empty space, but if left untouched, Unity will automatically fill the physics shape as the whole tile. To prevent this, collapsing its physic shape into a single point is needed.

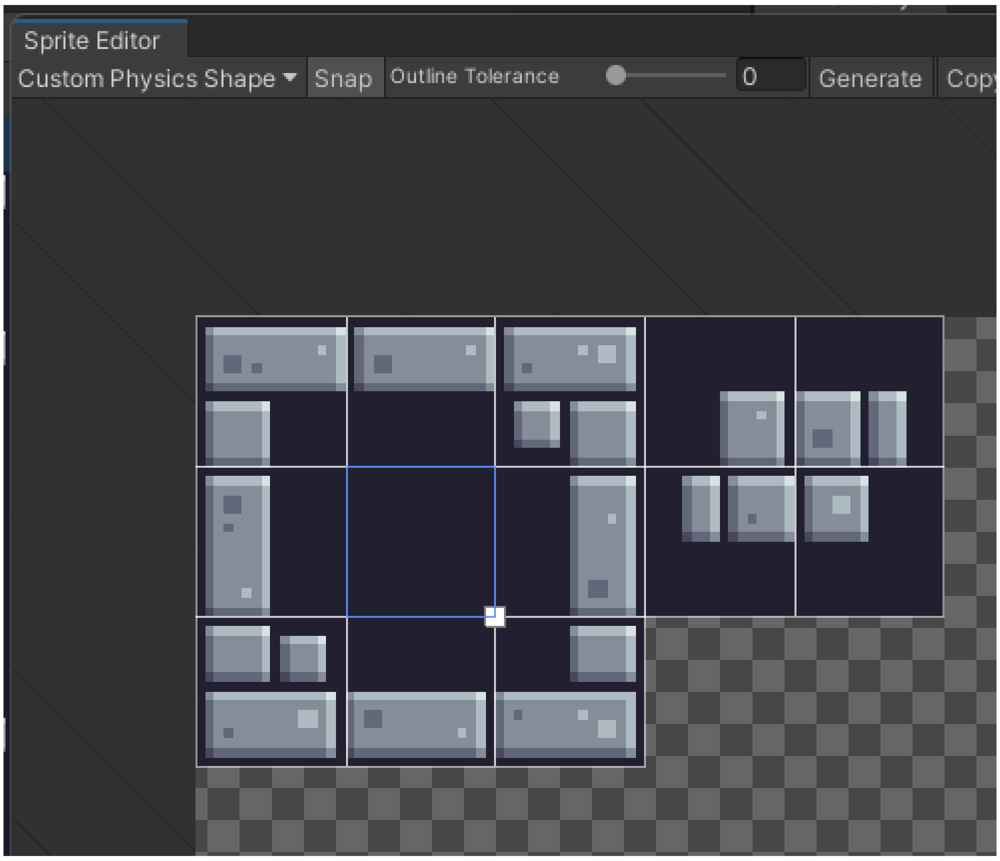


Figure 11. Collapsing the empty tile's physic shape into a point

4.4.4 World Movement

In this game, the same method of world movement of Flappy Bird, where instead of the bird moving forward through the pipes, the bird is fixated in the center of the screen and the pipe is instead moving backwards, to gives off the illusion that the bird is moving forward. The same methods of obstacles moving backward is used here in the game, but the player, however, is not fixated, and instead able to move freely while every is moving back, to let the player navigate through some obstacles that requires moving backwards or any kind of repositioning.

A script called “Moveable” is created and added as a component to almost every GameObjects (including the Player, as discussed later). The Moveable script were coded to move the GameObject that it is attached to, to the left at a certain speed, and also give it the ability to self-destroy when it is already pass the screen to the left, as that obstacle is already finished and is no longer in use. The reason that the player itself, although having its own movement, get the Moveable component as well, is that if it does not have the component, it will stand still at one place while everything is moving to the left underneath its feet, until there is a collider ahead, and then it will get pushed backwards, but we do not want this since if the player is standing still, we would want them to move with the ground beneath it as well.

```
using UnityEngine;

public class Moveable : MonoBehaviour
{
    public float speed;

    private void Update()
    {
        // Move the pipe to the left
        transform.position += Vector3.left * speed * Time.del-
taTime;

        // Check if pipe is offscreen
        if (transform.position.x < -20f)
        {
            // Destroy the pipe
            Destroy(gameObject);
        }
    }
}
```

Figure 12. Moveable script

4.4.5 Obstacle Generation

Now that everything is ready, the last thing to be done is the actual generation of obstacles.

A new empty GameObject and script is created, called WorldSpawner, to handle all the spawning logic of the obstacles. The script includes:

- All of the Obstacles prefabs, categorized into 3 arrays, botOpeningPrefabs, midOpeningPrefabs, and topOpeningPrefabs. As their names suggest, they contain the prefabs that which have the opening position correspond to the array name. Since there are 27 unique Obstacles, we know that there is 9 Obstacles in each of the array.
- A float called spawnIntervals, which dictate the rate that Obstacles spawns.
- An update function, that is being called every frame, to check for intervals of spawn timer, and calls the functions that instantiate Obstacles at that spawn interval.
- A GameObject variable called currentObstacles, that holds the info of the most recently spawned Obstacle, and a variable called currentObstacleEndPosition, to hold the info of the ending of that Obstacle.
- Three functions called InstantiateBotOpening, InstantiateMidOpening, and InstantiateTopOpening, to instantiate a random Obstacles from the Obstacles arrays, that have a specific opening position.

```
using UnityEngine;
using Random = UnityEngine.Random;

public class WorldSpawner : MonoBehaviour
{
    private enum LastObstacleEndPosition
    {
        Bot,
        Mid,
        Top
    }

    //[SerializeField] private Grid baseGrid;
    [SerializeField] private GameObject[] botOpeningPrefabs;
    [SerializeField] private GameObject[] midOpeningPrefabs;
    [SerializeField] private GameObject[] topOpeningPrefabs;

    [SerializeField] private float spawnIntervals;

    private GameObject _currentObstacle;
    private LastObstacleEndPosition _currentObstacleEndPosition;
    private float _timer;

    private void Start()
    {
        InstantiateMidOpening();
    }

    void Update() {

        _timer += Time.deltaTime;

        if (_timer >= spawnIntervals) {
            switch (_currentObstacleEndPosition)
            {
                case LastObstacleEndPosition.Bot:
                    InstantiateBotOpening();
                    break;
                case LastObstacleEndPosition.Mid:
                    InstantiateMidOpening();
                    break;
                case LastObstacleEndPosition.Top:
                    InstantiateTopOpening();
                    break;
            }
            _timer = 0f;
        }
    }

    void InstantiateBotOpening()
    {
        _currentObstacle = Instantiate(botOpeningPrefabs[Ran-
dom.Range(0, botOpeningPrefabs.Length)], transform);
        _currentObstacleEndPosition = (LastObstacleEndPosi-
tion)_currentObstacle.GetComponent<Obstacle>().getEndPosition();
    }
    void InstantiateMidOpening()
    {
        _currentObstacle = Instantiate(midOpeningPrefabs[Ran-
dom.Range(0, midOpeningPrefabs.Length)], transform);
        _currentObstacleEndPosition = (LastObstacleEndPosi-
tion)_currentObstacle.GetComponent<Obstacle>().getEndPosition();
    }
    void InstantiateTopOpening()
    {
        _currentObstacle = Instantiate(topOpeningPrefabs[Ran-
dom.Range(0, topOpeningPrefabs.Length)], transform);
        _currentObstacleEndPosition = (LastObstacleEndPosi-
tion)_currentObstacle.GetComponent<Obstacle>().getEndPosition();
    }
}
```

Figure 13. WorldSpawner script

The Obstacle spawning logic is as follows:

At the start of the game, a random Obstacles which openings position is middle is always spawned, then its info is stored, including its ending position. At the next spawning interval, depending on the ending position of the last Obstacle, a corresponding function will be called to spawn a new Obstacle that opening position is the same as the ending position of the last Obstacle, to ensure they are connected. Then its info is stored, and the loop continues, until the game ends.

4.4.6 Lose Condition

Now that the game is working, the last step is to check for the losing conditions of the player. In this game, the player will lose if they fail to keep up to the tracks and falls too far behind, or they fell down out of the world in one of the many Sky Obstacles.

We can achieve this by creating two separate colliders, with a custom tag called "Barrier", at the left most part and the bottom part of the screen, but on the outside. The collider is then set to Is Trigger, which indicates that when touched, it will send a trigger signal to the GameObject that triggered it, in this case, the Player.

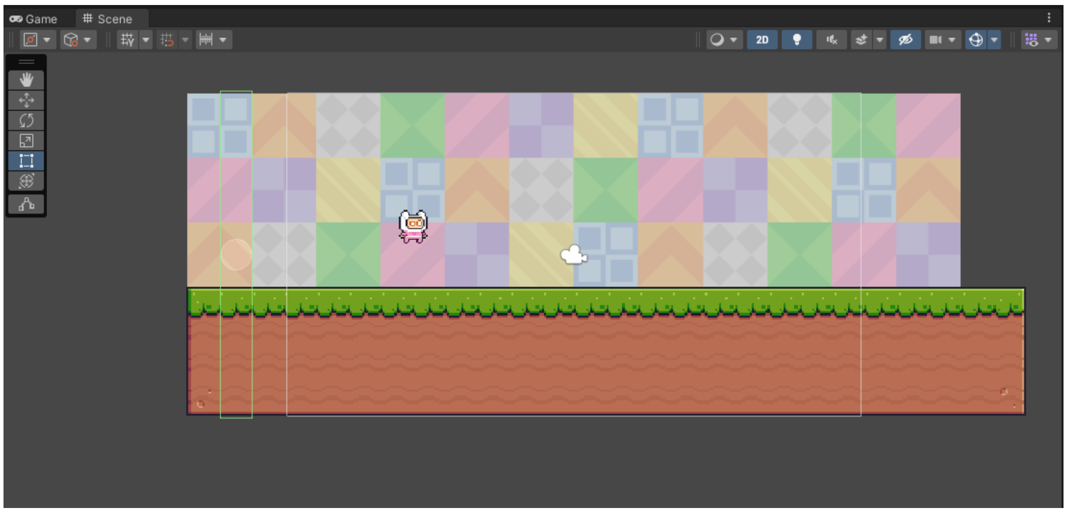


Figure 14. Left death-barrier (in green)

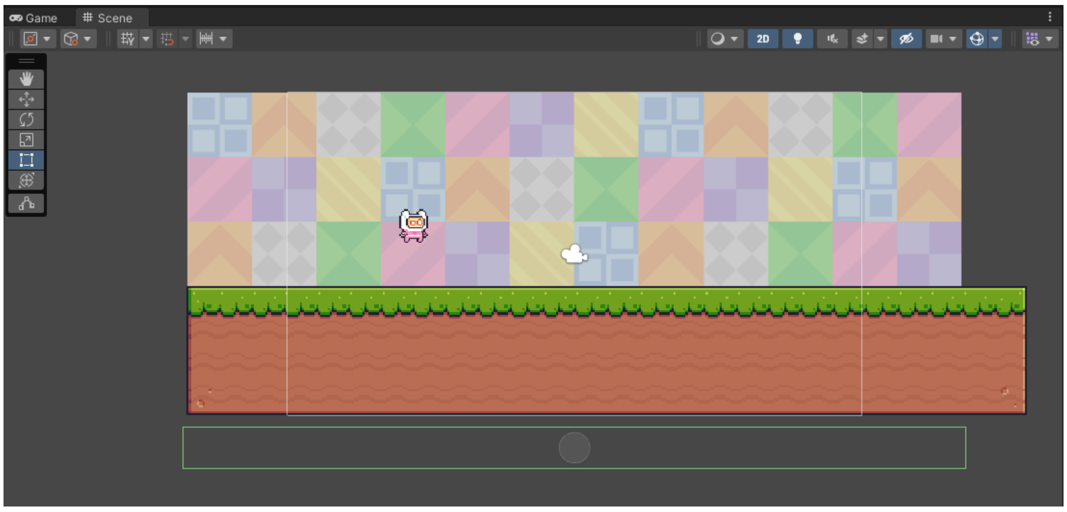


Figure 15. Bottom death-barrier (in green)

The player’s script is added a new function, called OnTriggerEnter2D, that detects whether if it enters any trigger called “Barrier”, and if so, meaning the player is falling to far behind, or fall below out of the world, the game will be stopped.

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Barrier")
    {
        GameManager.Instance.EndGame();
    }
}
```

Figure 16. Player's OnTriggerEnter2D

The script is referencing another script called GameManager, which is created to handle general in game events, such at when the game is ended, and later on when a UI system is created, to handle button clicks to start the game, and to get back to the main menu.

For now, the only job of the GameManager script is to freeze the game when the player dies, indicating the game is over. By changing the timescale to zero, we can achieve this effect.

```
public void EndGame()
{
    Time.timeScale = 0;
}
```

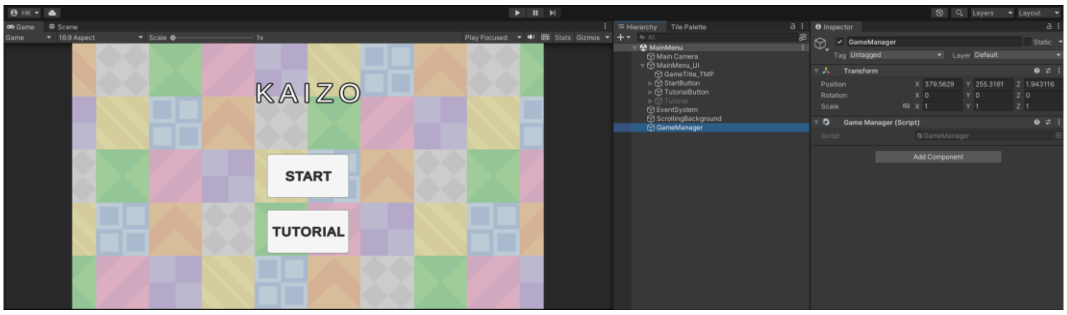
Figure 17. Freezing the game with timescale

With that finished, the core game is completed and ready to be played.

4.5 Game UI

Although the core gameplay is finished, the game is still not fully functional as a game because as soon as the program is started, the game starts immediately.

First, we can create another scene for the main menu, called MainMenu. Using this new Scene, we can create a Canvas, and to that adding different UI components such as texts, buttons and images to create a simple main menu UI.



GameManager
Tag Untagged Layer Default
Transform
Position X 379.5629 Y 255.3161 Z 1.943116
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Game Manager (Script)
Script GameManager

Figure 18. MainMenu scene

To handle buttons and events, a GameManager script is created as mentioned above, and since GameManager controls the important functions for the main events, such as changing scenes, getting into the game, and ending the game.

The GameManager scripts follow the Singleton pattern. The Singleton pattern is a software design pattern that restricts the instantiation of a class to a singular instance. One of the well-known "Gang of Four" design patterns, which describes how to solve recurring problems in object-oriented software, the pattern is useful when exactly one object is needed to coordinate actions across a system. More specifically, the Singleton pattern allows objects to ensure they only have one instance, provide easy access to that instance, and control their instantiation (Wikipedia, n.d.).

It is often used in game development to manage core systems, such as input, audio, or networking, that need to be shared across multiple scenes and scripts (LinkedIn, n.d.).

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    public static GameManager Instance { get; private set; }

    //private bool gameEnded = false;

    void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
        DontDestroyOnLoad(gameObject);
    }

    public void EndGame()
    {
        Time.timeScale = 0;
    }

    public void BackToMainMenu()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(0);
    }

    public void StartGame()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(1);
    }

}
```

Figure 19. GameManager script

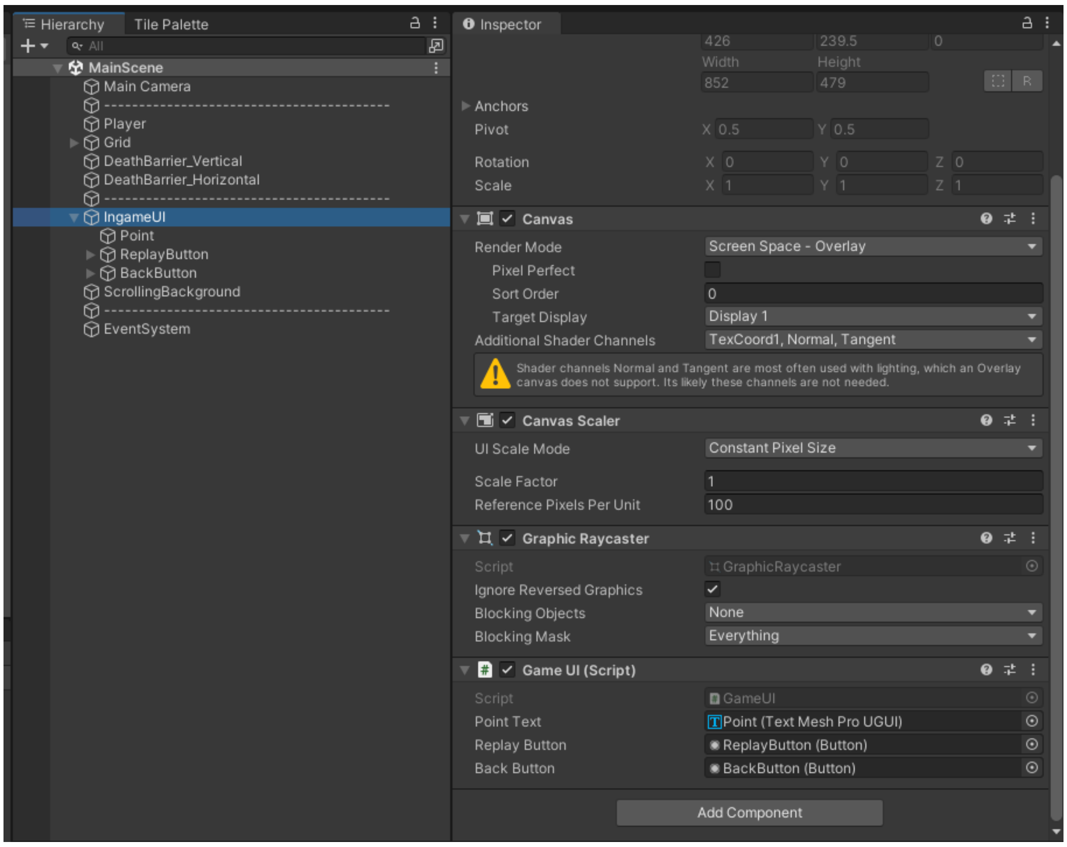
The GameManager script, when it was first initiated, will call on itself a function called DontDestroyOnLoad, which does as what it is named, make itself not be destroyed when another scene is loaded, which helps the GameManager object persist even when we change the scene to the main GameScene. This step is necessary, because GameManager also contains the functions to handle ending the game, and loading back to the MainMenu, which could only be happened in the GameScene, since GameManager started in the MainMenu, we need a way for GameManager to persist into another scene when it is loaded.

A canvas in the game scene is also added, for displaying the duration that which the player survived, and also to display buttons for the player to replay, or go back to the main menu, having died. A script called GameUI is attached to the canvas, to handle all of the logics relate to the UI, including counting the time survived by the player, and to navigate using buttons.

```
public class GameUI : MonoBehaviour
{
    private float point;
    [SerializeField] private TextMeshProUGUI pointText;
    [SerializeField] private Button replayButton;
    [SerializeField] private Button backButton;
    void Start()
    {
        point = 0;
        replayButton.gameObject.SetActive(false);
        backButton.gameObject.SetActive(false);
        replayButton.onClick.AddListener(() =>
        {
            GameManager.Instance.StartGame();
        });
        backButton.onClick.AddListener(() =>
        {
            GameManager.Instance.BackToMainMenu();
        });
    }

    // Update is called once per frame
    void Update()
    {
        point += Time.deltaTime;
        pointText.text = (Mathf.Round(point * 100f) /
100f).ToString();
    }
}
```

Figure 20. GameUI script



Hierarchy | Tile Palette

All

MainScene
- Main Camera
- --
- Player
- Grid
- DeathBarrier_Vertical
- DeathBarrier_Horizontal
- --
- IngameUI
  - Point
  - ReplayButton
  - BackButton
- ScrollingBackground
- --
- EventSystem

Inspector

426 | 239.5 | 0
Width | Height
852 | 479

Anchors
Pivot: X 0.5 Y 0.5
Rotation: X 0 Y 0 Z 0
Scale: X 1 Y 1 Z 1

Canvas
Render Mode: Screen Space - Overlay
Pixel Perfect: ☐
Sort Order: 0
Target Display: Display 1
Additional Shader Channels: TexCoord1, Normal, Tangent
Shader channels Normal and Tangent are most often used with lighting, which an Overlay canvas does not support. Its likely these channels are not needed.

Canvas Scaler
UI Scale Mode: Constant Pixel Size
Scale Factor: 1
Reference Pixels Per Unit: 100

Graphic Raycaster
Script: GraphicRaycaster
Ignore Reversed Graphics: ✓
Blocking Objects: None
Blocking Mask: Everything

Game UI (Script)
Script: GameUI
Point Text: Point (Text Mesh Pro UGUI)
Replay Button: ReplayButton (Button)
Back Button: BackButton (Button)

Add Component

Figure 21. Ingame UI hierarchy

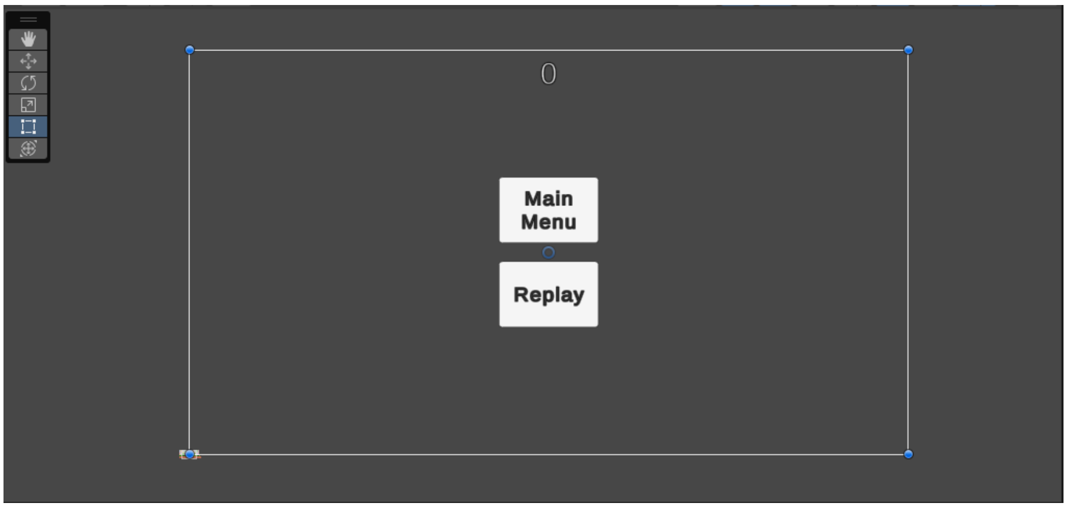


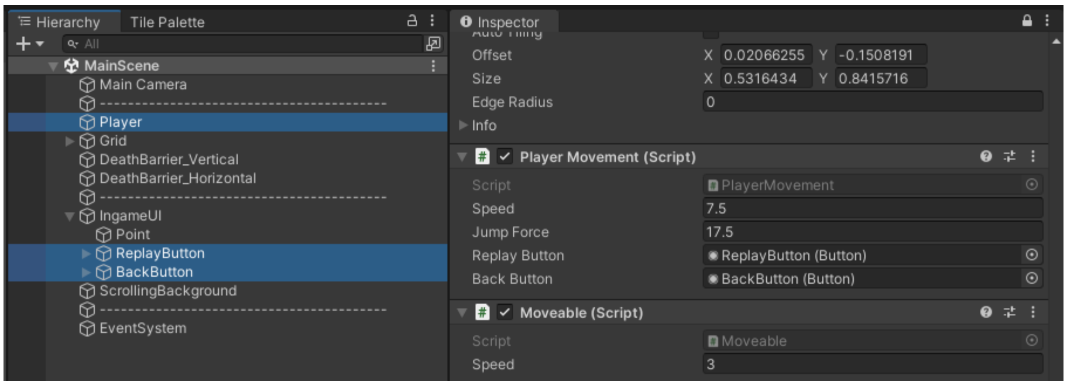
Figure 22. Ingame UI

The MainMenu and Replay buttons are linked to functions in the GameManager, and since GameManager is a Singleton as previously discussed, they can access the correspond GameManager's function directly through its instance. At start, the MainMenu and Replay buttons will be turned off, out of display, since we only want them to show up when the player actually dies. To do that, we need to reference the two button inside of the Player scripts, and when they die, which is also when there OnTriggerEnter2D is called, we can toggle them back on.

```
[SerializeField] private Button replayButton;
[SerializeField] private Button backButton;


private void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Barrier")
    {
        GameManager.Instance.EndGame();
        replayButton.gameObject.SetActive(true);
        backButton.gameObject.SetActive(true);
    }
}
```

Figure 23. Updated Player script to handle UI



```
MainScene
  Main Camera
  ----------------------------------------
  Player
  Grid
  DeathBarrier_Vertical
  DeathBarrier_Horizontal
  ----------------------------------------
  IngameUI
    Point
    ReplayButton
    BackButton
  ScrollingBackground
  ----------------------------------------
  EventSystem
```

Offset	X 0.02066255 Y -0.1508191
Size	X 0.5316434 Y 0.8415716
Edge Radius	0
Info	

Player Movement (Script)

Script	PlayerMovement
Speed	7.5
Jump Force	17.5
Replay Button	ReplayButton (Button)
Back Button	BackButton (Button)

Moveable (Script)

Script	Moveable
Speed	3

Figure 24. Referencing UI buttons into Player script

5 CONCLUSIONS

The game product was completed as planned, and this paper provides a good foundation and instruction on how to create a 2D platform endless runner game using the Unity Game Engine.

Although the project was completed as planned, it cannot be considered as a fully developed game, yet. More game balance could be implemented, to make the feels of player movement as smooth as possible. More Obstacles could be added, and future implements could contain traps and even moving enemies. A sound system is a must have feature later on, and a way to save and show the leader-boards of players who have survived the longest is a nice to have feature.

REFERENCES

Arm, n.d. *What is a Gaming or Game Engine?.* [Online]
Available at: https://www.arm.com/glossary/gaming-engines#:~:text=A%20gaming%20engine%20is%20a,a%20variety%20of%20programming%20languages.
[Accessed 01 05 2023].

Atlassian, n.d. *What is git.* [Online]
Available at: https://www.atlassian.com/git/tutorials/what-is-git
[Accessed 02 05 2023].

GameDev Academy, n.d. *What is Unity? – A Guide for One of the Top Game Engines.* [Online]
Available at: https://gamedevacademy.org/what-is-unity/#What_is_Unity
[Accessed 01 05 2023].

Jetbrains, 2022. *Get started.* [Online]
Available at: https://www.jetbrains.com/help/rider/Introduction.html
[Accessed 03 05 2023].

LinkedIn, n.d. *How do you use the singleton pattern in game development?.* [Online]
Available at: https://www.linkedin.com/advice/1/how-do-you-use-singleton-pattern-game-development
[Accessed 08 05 2023].

LinkedIn, n.d. *What is procedural generation? Why do game developers use it? What is the advantage of it over a hand-crafted map?.* [Online]
Available at: https://www.linkedin.com/pulse/what-procedural-generation-why-do-game-developers-use-advantage-
[Accessed 05 05 2023].

Unity, n.d. *Game Development Terms.* [Online]
Available at: https://unity.com/how-to/beginner/game-development-terms#:~:text=glossary%20for%20details.,%2C%20maps%2C%20environments%2C%20etc
[Accessed 05 05 2023].

Unity, n.d. *Introduction to asset store.* [Online]
Available at: https://unity.com/pages/introduction-to-asset-store
[Accessed 01 05 2023].

Unity, n.d. *Manual: Create with Tilemaps.* [Online]
Available at: https://docs.unity3d.com/Manual/Tilemap.html
[Accessed 06 05 2023].

Unity, n.d. *Manual: Grid.* [Online]
Available at: https://docs.unity3d.com/Manual/class-Grid.html
[Accessed 06 05 2023].

Wikipedia, n.d. *C Sharp (programming language).* [Online]
Available at: https://en.wikipedia.org/wiki/C_Sharp_(programming_language)
[Accessed 03 05 2023].

Wikipedia, n.d. *Endless runner.* [Online]
Available at: https://en.wikipedia.org/wiki/Endless_runner
[Accessed 04 05 2023].

Wikipedia, n.d. *GitHub.* [Online]
Available at: https://en.wikipedia.org/wiki/GitHub
[Accessed 02 05 2023].

Wikipedia, n.d. *Platform game.* [Online]
Available at: https://en.wikipedia.org/wiki/Platform_game
[Accessed 04 05 2023].

Wikipedia, n.d. *Procedural Generation.* [Online]
Available at: https://en.wikipedia.org/wiki/Procedural_generation
[Accessed 05 05 2023].

Wikipedia, n.d. *Singleton pattern.* [Online]
Available at: https://en.wikipedia.org/wiki/Singleton_pattern
[Accessed 08 05 2023].

Wikipedia, n.d. *Version control.* [Online]
Available at: https://en.wikipedia.org/wiki/Version_control
[Accessed 01 05 2023].