UNITY MULTIPLAYER GAME

Institute of Technology Blanchardstown
Third Year Computing Project

Individual Project
by
Dylan Byrne

Submmitted in part fulfillment for the degree of
B.Sc in Computing in Information Technology
School of Informatics and Engineering,
Insititute of Technology Blanchardstown,
Dublin, Ireland

May 2017

DECLARATION

I herby certify that this material, which I now submit for this assessment on the program leading to the award of Bachelor of Science in Computing in Information Technology in the Institute of Technology Blanchardstown, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other academic institution other than in partial fulfillment of the requirements of that stated above

Signed_____

Dated_____

ABSTRACT

Acknowledgements

Thanks to …

My project Supervisor Matt Smith.

My friends Danut Hij, Owen Flannery and Karl Jones for being more than happy to lend a hand whenever they were available.

My mother and father for all the support they have given me throughout my college terms.

And finally, thank you to the Institute of Technology Blanchardstown for the resources provided to me throughout the years and for furthering my education for now and hopefully in the future when I continue in fourth year.

Table of Contents

DECLARATION i

ABSTRACT i

Acknowledgements i

1 Project Introduction 1
- 1.1 Project Objectives . . . 1
- 1.2 Overview of Project . . . 2
- 1.3 Overview of Report . . . 3

2 Research Conducted 5
- 2.1 Unity and Video Game Engines . . . 5
- 2.2 Multi-Player Video Games . . . 6
- 2.3 Genres of Video-Games . . . 8
  - 2.3.1 Action-Adventure Games . . . 8
  - 2.3.2 Party Games . . . 8
  - 2.3.3 Platform Games . . . 8
  - 2.3.4 Strategy Games . . . 8
  - 2.3.5 Puzzle Games . . . 9
- 2.4 Conclusion . . . 10

3 System Analysis 11
- 3.1 Functional Requirements . . . 11
- 3.2 Use Case Diagrams . . . 11
  - 3.2.1 Protagonist Use Case . . . 12
    - 3.2.1.1 Start Game . . . 12
    - 3.2.1.2 Move Player . . . 12
    - 3.2.1.3 Shoot Projectile . . . 13
    - 3.2.1.4 View and Update Inventory . . . 13
    - 3.2.1.5 Quit Game . . . 14
  - 3.2.2 Antagonist Use Case . . . 14

3.2.2.1 Move Fireballs . . . 15
3.2.2.2 Destroy Player . . . 15
3.3 Class Diagrams . . . 15
3.3.1 Protagonist and Antagonist Classes . . . 16
3.3.2 Enemy AI Classes . . . 16

4 Design and Implementation of System **19**
4.1 Overview of Implementations . . . 19
4.2 Implementation Components . . . 19
4.2.1 Controller Input . . . 20
4.3 Graphical Components & Player Display . . . 25
4.4 Projectile System . . . 32

5 Testing **35**

6 Conclusions **37**

I Appendices **39**

A Screen Shots **41**

B Program Listings **43**
B.1 List of programs Included . . . 43

List of References **53**

1

Project Introduction

1.1 Project Objectives

The main goal of this project was to create a simple multi-player based party game in the Unity video game engine, akin to Mario Party for example. The game has to have entertainment value. Have simple rules, be easy to follow and must be capable of being completed, the game must not be too difficult.

The main goals for the game are as follows

- There are a minimum of three players.
- One player controls an antagonist type character represented by a series of fire balls. These fireballs are lined up horizontally and vertically.
- The fireballs lined up vertically can only move from left to right, and the the fireballs lined up horizontally can only move up and down.
- Two or more players control the protagonist characters. It is their job to each try and survive the level for 60 seconds.
- It is the player controlling the antagonist's role to move the fireballs around in an attempt to kill the protagonist players.
- While it is the protagonists job to avoid these fireballs while also attempting to hinder the other player(s).
- There are pickups which can be collected at certain intervals throughout the level.

- One pickup is a shield which provides the player with invincibility for a short period of time from the fireballs.
- One pickup is an ice blast, which when picked up adds to an inventory and is used as a projectile to freeze the other

player in place.

- There are two game over screens. One which displays when all protagonist are killed, and another which displays when a protagonist(s) survive for 60 seconds.

1.2 Overview of Project

For the third year project I decided to develop a game within the Unity engine that would have multi-player elements to it. When the game was completed it would be tested within the engine using Xbox controllers and the keyboard on the PC itself.

The game consists of a minimum of 3 players, one player controlling the antagonist represented by a series of fireballs, and at least two players controlling the protagonists consisting of the "Gravity Guys". The two players try so survive the level, while also hindering the other players with pickups which can be collected throughout the level.

The project was originally supposed to be a group project, it eventually became an individual project. The project started out as a group deciding what would be the best way to undergo research in how to develop this game? What type of a game would it be? How could this game be developed into an easy to play, and most of all, a fun multi-player experience?

Research was started by looking at the Unity engine itself and previous games which have been developed in the engine and the capabilities shown in these games and if it would be possible to implement these features into the game I was developing. More research was conducted into multi-player player games in general, such as their popularity and how can you and your fellow players can enjoy the experience as much as possible?

Also looked at was the issues that some of these games had, and how exactly could they be improved upon in the game that was being developed for this project. The final game would then hopefully be completed using the information gathered from the conducted research. A demo of the project was displayed in December to show how the basics of how the game would work, with a full working version being finished by the end of April.

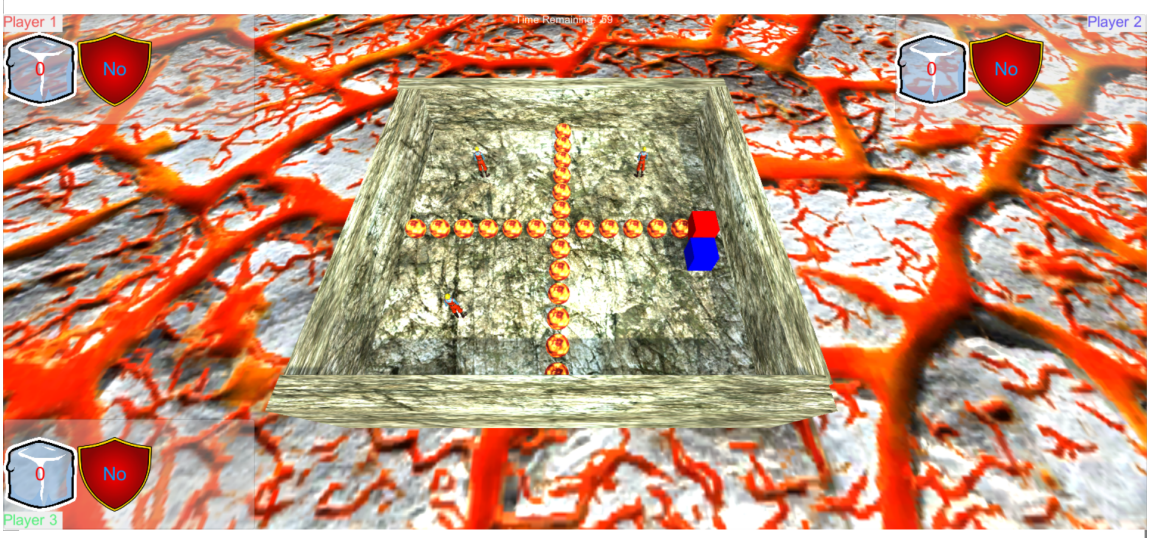


Figure 1.1: Overview of Project

1.3 Overview of Report

Chapter 2 will contain the research which was conducted into the development of this game, such as looking at video-game engines and how they work, what are multi-player games and the different genres of video games. Chapter 3 will contain a system analysis of how the game works and the features of the game. While in chapter 4 will look at the key components of the project and how they were implemented. Chapter 5 will focus on the testing of the games features to see if they all work correctly. Chapter 6 will be a conclusion of the entire final project, what could be improved in the future. Followed by all this will be appendix.

2

Research Conducted

In this chapter I will present all the relevant research relating to the project that has been conducted. This includes research of the Unity engine, games developed in the engine, networked and non-networked multi-player games, and the different genres of video-games.

2.1 Unity and Video Game Engines

Almost every one on planet earth has probably come into contact with video games at some point in their life, whether it through playing them, watching someone else play them or even seeing a movie previously based off of a video game. Yes, it is impossible to avoid video games on modern society. But how exactly are video games made? How does what you interact with on screen from traversing through space, to saving your princess from castle to castle, to even playing a virtual sport without ever having to leave your house made possible? It's simple really, these are all made possible through the software frameworks which are video game engines. These engines are used to design the video games in which you play on your consoles, your PCs and even your mobile devices. These frameworks provide for you, the tools to make your video game such as, the physics of your game, the animations, the scripting and the AI. Popular video game engines include the Frostbite engine, the Unreal engine, and of course the Unity engine.

The Unity Engine was first announced in 2005 originally only to be used on Mac OS due to its popularity it has since been extended to 27 platforms such as Desktop, Games Consoles, Virtual Reality and even Smart Tv's. The popularity of the platform increased even more with the launch of Unity 5 in 2015 with more than 5 billion games developed in Unity downloaded in the third

quarter of 2016 alone, with the technology being popular among developers more than any other third party game development software, with developers including the likes of Ubisoft, Square Enix, Warner Bros, Obsidian and even Coca Cola.

Many popular games in today's gaming culture have been developed in Unity including the likes of, Slender: The Eight Pages, a popular survival horror game in which a player must collect eight pages scattered throughout a map all while avoiding someone who is following them. 7 Days to Die, another survival horror game set in an open world where the player must craft and survive in the wilderness while fighting against zombies. Cities Skylines, a simulator in which the user builds entire cities and the rules in which the city operates, and recently the extremely popular mobile game Pokemon GO was developed in Unity, a game in which you walk around with your phone and then encounter Pokemon and use the camera in your phone to capture them.

All of this shows that there are many different features in which can be implemented into games made using this engine and just how creative a developer can be when developing a game which is one the goals of my project was to try make the game as creative as possible.

2.2 Multi-Player Video Games

Multi-Player is one of the most significant features incorporated into video games today. Recent demographics show that 56% of gamers in America play games predominantly with other players, while 54% of gamers play a multi-player game weekly. But what is a multi-player game? What makes them so predominantly played and what types of multi-player games are there?

A multi-player game is a video game in which two or more people can play the same game, in the same environment simultaneously. Depending on the nature of the game, the two players can either be playing as part of a team, or can be playing as foes. There are two ways in which a multi-player game can be achieved; either through a non-networked system, or a networked system.

Non-Networked multi-player games are games which do not require an online internet connection for people to play with each other. One of the earliest games that became popular for its multi-player game-play was the sports game "Pong". First released in 1972, "pong" (See Figure 2.1) was a game in which two players would hit a ball between them in an effort to try and knock the ball past their opponents in order to score points, similar to that of table tennis.

Networked multi-player games are games in which an online internet connection is required and in real time the game is played with multiple other users. These type of games have become very popular in recent times through the success of first person shooters such as the "Call of Duty" (See Figure 2.2) series and the "Battlefield" series. The purpose of these games is that you compete against other players in different game types, which have different goals. But the ultimate goal in most game types is the player or team with the most kills wins.

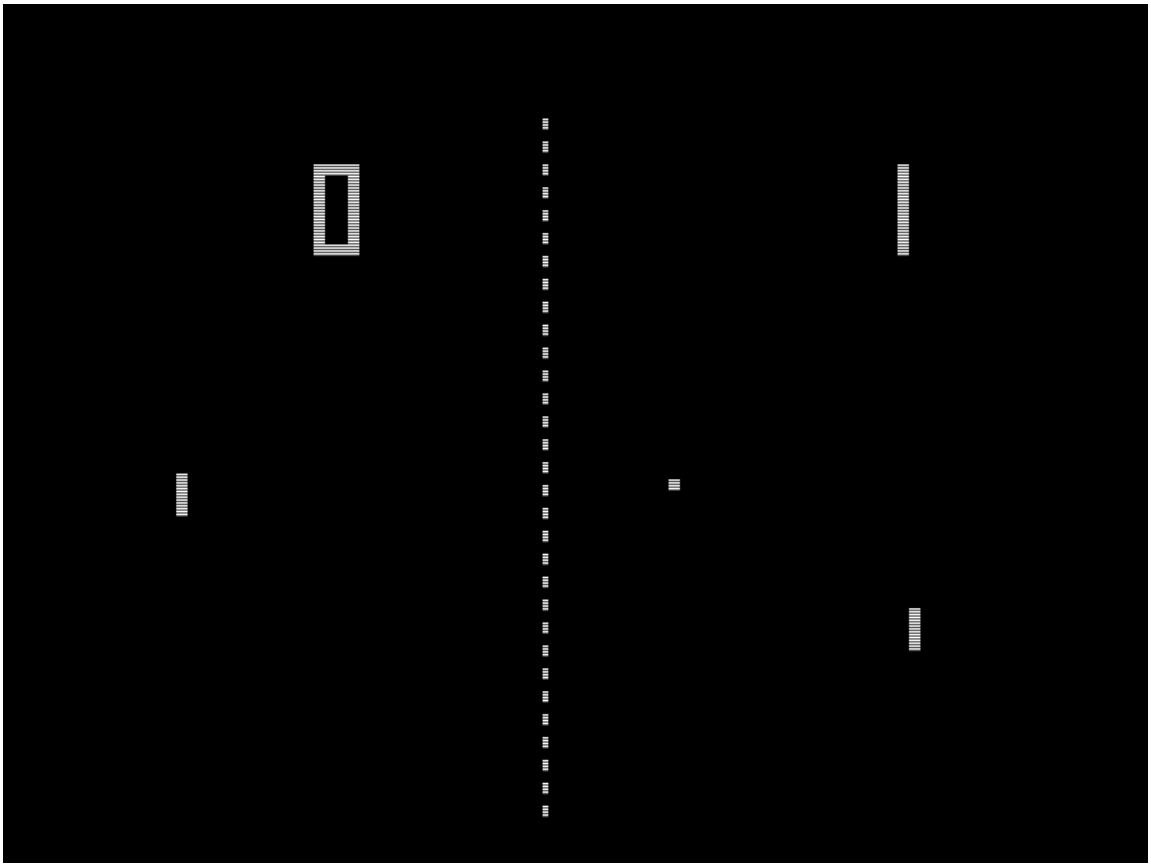


Figure 2.1: Pong

Figure 2.2: Call of Duty

2.3 Genres of Video-Games

There were many different genres of video games to consider when developing my game and which genre would be best suited to the game I was trying to make. Some of these genres included

2.3.1 Action-Adventure Games

Action-Adventure games are games which contain both a mix of games from the action genre and the adventure genre such as puzzles and scenes where your character must have to fight their way out of a situation. This genre may require the skills that are involved with playing through an action game, but also share's some of its elements from adventure games like a story line, inventory system and multiple characters. Some popular action-adventure games include the likes of the Tomb Raider series and the Uncharted (See Figure 2.3) series.

2.3.2 Party Games

Party games are video-games that are normally developed solely for multi-player use to play either locally or online with friends. They are commonly designed as a collection of simple mini-games, designed to be intuitive and easy to play. Examples of these video-games are games such as Mario Party (See Figure 2.4) and the Jackbox collection

2.3.3 Platform Games

Platform games are games that have been popularised in the 2D world although can also be developed in 3D. These games normally require the user to move simply from one end of the map to the other, normally while avoiding obstacles or enemies. They may also have platforming features and require the player jump onto higher or lower ground. This genre has been popularised by games such a Super Mario and Donkey Kong.

2.3.4 Strategy Games

Strategy games are games which require careful planning,strategic thinking and logistical thinking to win. Normally they require players These are two type of strategy games, these being real time and turn based strategy games. Real time is when both players play at the same time, while in turn based strategy games the players take turns. Examples of video games in this genre are Tower Defence and Sim City.

2.3.5 Puzzle Games

Puzzle games are games in which the user must use their brains to solve puzzles such as mazes. Games in this genre include the likes of Boulder Dash.

There are many other genre of games, but these are the genres which I considered for the development of my game.

Figure 2.3: Uncharted

Figure 2.4: Mario Party

2.4 Conclusion

As a result of the research and analysis carried out it was decided to develop a non-networked multi-player game that would combine many of these genres with each level having at least 3 players. The game would be developed within the Unity engine, while the scripts would be compiled in C# and JavaScript.

3

System Analysis

3.1 Functional Requirements

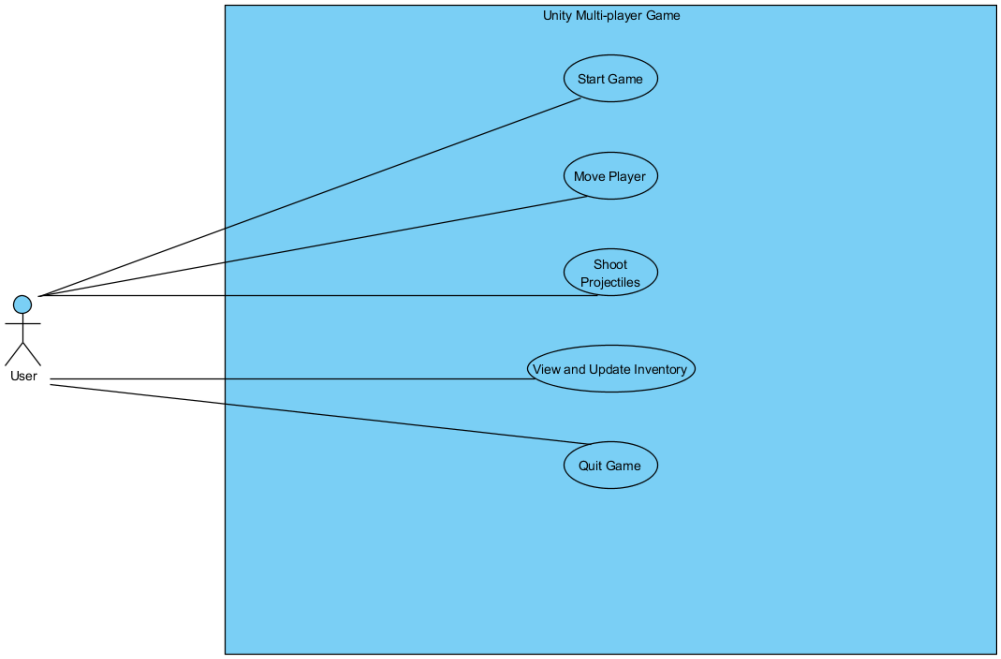
There are a number of requirements for the project. These requirements consist of:

1. There is a protagonist and antagonist
2. Users control movement of characters.
3. Users can shoot at opponents.
4. Users can view their inventory.
5. Users can quit game whenever they want.

3.2 Use Case Diagrams

These are formal UML use case diagrams and specifications. They describe the actions that are triggered but the actors, being the users in the system.

3.2.1 Protagonist Use Case



3.2.1.1 Start Game

Summary: This use case describes the starting of the game by the player from a menu.

Precondition: The user has not yet started the game

Triggers: The user executes the game program.

Course of Events

1. The user boots up the game.
2. The user is taken to the opening menu of the game.
3. The user presses a button which activates the code to start the level.
4. The user begins playing the game.

Post Condition: The game is initialised and the user starts to play.

3.2.1.2 Move Player

Summary: This use case describes the player moving the player that he is controlling.

Precondition: The game is initialised.

Triggers: The user presses or releases a movement key (or analog on controller).

Course of Events:

1. The user decides in which direction to move the character, whether it be horizontally or vertically.
2. Based on the input configuration set in the Unity Project settings
3. The character will move according to their speed.
4. When the user releases the movement key the character will stop and the characters position will be updated.

Post Condition: The character has moved accordingly.

3.2.1.3 Shoot Projectile

Summary: This use case will describe the user firing the projectile which has been picked up.

Precondition: The user has collected a blue freeze block and has at least one in their inventory.

Triggers: The user presses the fire button (being tab if using the keyboard, or the right bumper if using the controller.)

Course of Events:

1. The user collects a freeze block which is then added to the characters inventory.
2. The user can then attempt to fire the freeze projectile at the other players as desired.
3. If a user is hit then they are frozen in place for 3 seconds.
4. The inventory is then updated and the user will have one less freeze block.
5. If the user has zero projectiles left they can't fire.

Post Condition: The user has fired their projectile and will now have one less.

3.2.1.4 View and Update Inventory

Summary: The user should always be able to see their inventory and see when it updates .

Precondition: The user has started the game and then picks up a red or blue block.

Triggers: Once a red or blue block is collected the user's inventory will update and tell them which block has been collected.

Course of Events:

1. Once the game has started the user can constantly see their inventory at the top of the screen.
2. The UI will update accordingly depending on the type of block which has been collected.

3. If a blue block has been collected, the UI will update to show one has been added to the users projectiles.
4. If a red block has been collected, the UI will update to show the user now has a shield which will last for 10 seconds.

Post Condition: The UI will update accordingly to the blocks collected.

3.2.1.5 Quit Game

Summary: This use case describes termination of the current game session or application.

Precondition: The game has been initialised.

Triggers: The user presses the quit button (In this case it is the escape key on the keyboard).

Course of Events:

1. The current game session will be closed and the user will be taken back to the main menu.
2. From the main menu the user can then quit the application if needed.

Post Condition: The session is terminated.

3.2.2 Antagonist Use Case

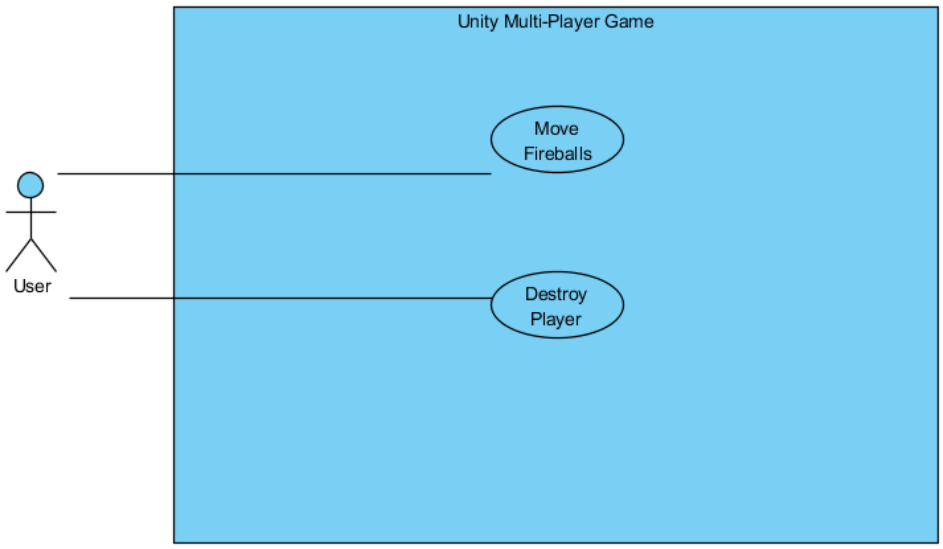


Figure 3.1: Antagonist Use Case

3.2.2.1 Move Fireballs

Summary: This use case describes the movements made by the user controlling the antagonist (Represented by fireballs on the map). The antagonist in this situation can only move horizontally and vertically (Using the arrow Keys).

Preconditions: The game must be initialized.

Course of Events:

1. The user decides which way to move the fireballs in order to try and destroy the protagonists.
2. The movement is based on the input settings in Unity. In this case they are controlled by the arrow keys.
3. The fireballs move according to their set speed.
4. When the user releases the movement key the character will stop and the characters position will be updated.
5. After 60 seconds these fireballs wills disappear as the game enters phase 2.

Post Condition: The fireballs disappear after 60 seconds if they are unable to destroy the protagonist characters as the game moves into its second phase.

3.2.2.2 Destroy Player

Summary: Once the protagonist character comes into contact with a fireball they are destroyed.

Preconditions: There must be at least 2 users, one controlling the protagonist and one controlling the antagonist.

Course of Events:

1. A user is controlling the antagonist.
2. The user controlling the antagonist comes into contact with the user controlling the protagonist.
3. The protagonist game object is then destroyed.

Post Condition: The user controlling the protagonist character is destroyed and out of the game.

3.3 Class Diagrams

These are static structured diagrams that are used to describe the structure within a system.

3.3.1 Protagonist and Antagonist Classes

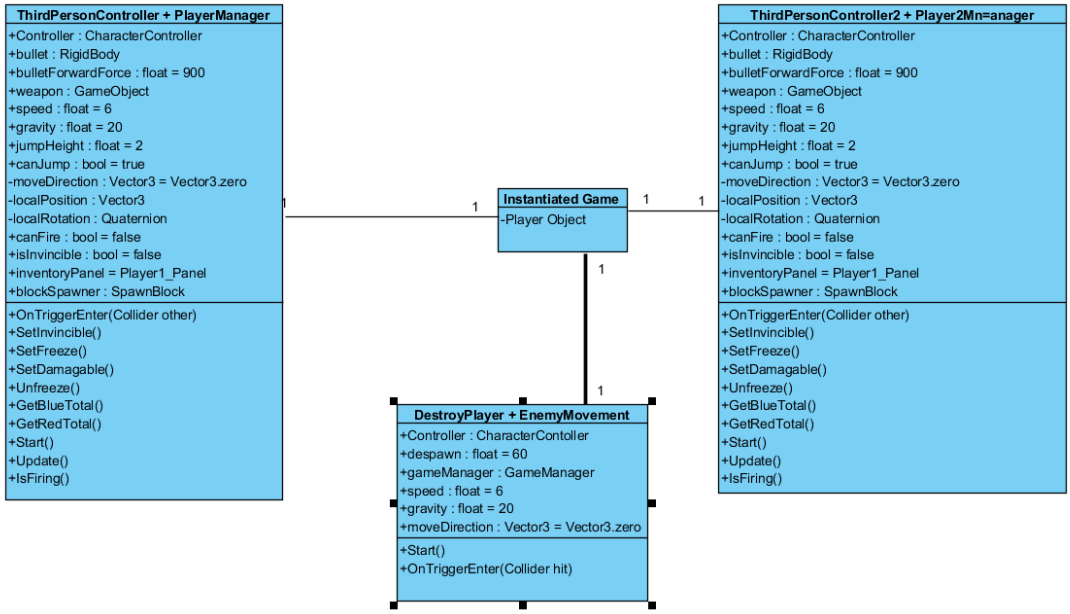


Figure 3.2: Protagonists and Antagonist Class Diagrams

The ThirdPersonController and PlayerManager classes are responsible for all of the player one and player two's actions in game. In this the players movement is tracked, if they fire a projectile, if they have a shield equipped etc is all implemented into these classes.

Also located in this is diagram is the classes for the antagonist. These classes are responsible for the antagonists controls, the movements of the antagonists and destroying the antagonist players.

3.3.2 Enemy AI Classes

The EnemyAIShooting class is responsible for the turrets in game shooting at the players. The turret tracks the movements of the characters and shoots a bullet every 2 seconds at the players last known position. The turrets will then despawn 130 seconds into the game. The projectile fired by the turret has the DestroyPlayer script equipped. This script will destroy the player character when it comes into contact with the player.

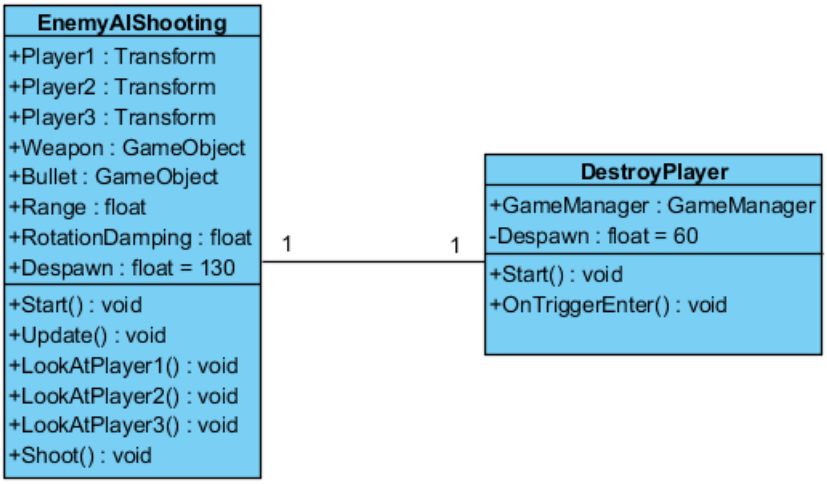


Figure 3.3: EnemyAIShooting and Destroy Player Classes

4

Design and Implementation of System

4.1 Overview of Implementations

The system was implemented using pre written game 3D characters models called "Gravity Guy" that was provided to us at the start of the second year Interactive Multi-Media course, these models already came with the scripts implemented, the game was implemented in the Unity game engine as mentioned in Chapter 2. To implements the various aspects of this I used some pre existing models from the Unity store for the environment, such as skins for the walls, platforms etc. All other scripts for the in-game components were all written and developed by me using the C# programming language.

4.2 Implementation Components

There were several components that needed to be implemented in order to provide a fully functioning game. These included:

- Controller Input
- Graphical Components & Player Display
- Collision Detection
- Projectile System
- Shield System
- Progression

4.2.1 Controller Input

Problem: The Gravity Guy character models already came with a third person controller attached, but I had to make edits as to accommodate for other players, as if the controls were similar for every player they would all move synchronously as opposed to independently. I also needed to implement a control for the player who would be controlling the antagonist.

Implementation: The input was implemented by copying the original ThirdPersonController.js script that came with the original gravity guy. Then I changed edited the scripts so that they could be changed in the InputManager that is available within Unity. The InputManager within Unity can be used to assign different keys to each control that is set on the character when the input function is called upon.

Code for player one to call upon InputManager

```
var right = Vector3(forward.z, 0, -forward.x);

var v = Input.GetAxisRaw("Vertical");
var h = Input.GetAxisRaw("Horizontal");
var right = Vector3(forward.z, 0, -forward.x);

var v = Input.GetAxisRaw("Vertical4");
var h = Input.GetAxisRaw("Horizontal4");
```

Once this is done you can assign keys to the variables listed through the name they have been given in the input manager eg “Vertical 4”.

Antagonist Controls

The four directional arrows up, down left and right are used by the antagonist to move the fireballs. The fireballs that are laid out horizontally can only move up and down using the up and down arrow keys, while the fireballs laid out vertically can move left and right using the left and right arrow keys. The public float “speed” is used to control how fast the enemy can move.

```
using UnityEngine;
using System.Collections;

public class EnemyNav2 : MonoBehaviour
{
    public float speed = 6.0F;
    public float gravity = 20.0F;
    private Vector3 moveDirection = Vector3.zero;
    void Update()
    {
        CharacterController controller = GetComponent<CharacterController>();
```

```
        if (controller.isGrounded)
        {
            moveDirection = new Vector3(Input.GetAxis("Horizontal1"), 0, Input.GetAxis("Vertical1"));
            moveDirection = transform.TransformDirection(moveDirection);
            moveDirection *= speed;

        }
        moveDirection.y -= gravity * Time.deltaTime;
        controller.Move(moveDirection * Time.deltaTime);
    }
}
```

```
using UnityEngine;
using System.Collections;

public class EnemyNav3 : MonoBehaviour
{
    public float speed = 6.0F;
    public float gravity = 20.0F;
    private Vector3 moveDirection = Vector3.zero;
    void Update()
    {
        CharacterController controller = GetComponent<CharacterController>();
        if (controller.isGrounded)
        {
            moveDirection = new Vector3(Input.GetAxis("Horizontal2"), 0, Input.GetAxis("Vertical2"));
            moveDirection = transform.TransformDirection(moveDirection);
            moveDirection *= speed;

        }
        moveDirection.y -= gravity * Time.deltaTime;
        controller.Move(moveDirection * Time.deltaTime);
    }
}
```

Then once I completed that code and called the function "Input.GetAxis(Vertical1)", Vertical one now shows up in my input manager and the up and down keys are assigned to it to make the characters go up and down. The same is then done for where I called "Horizontal2" and the left and right keys are assigned to make the enemy to be able to move left and right. (See Figures 4.1, 4.2)

Protagonist Controls

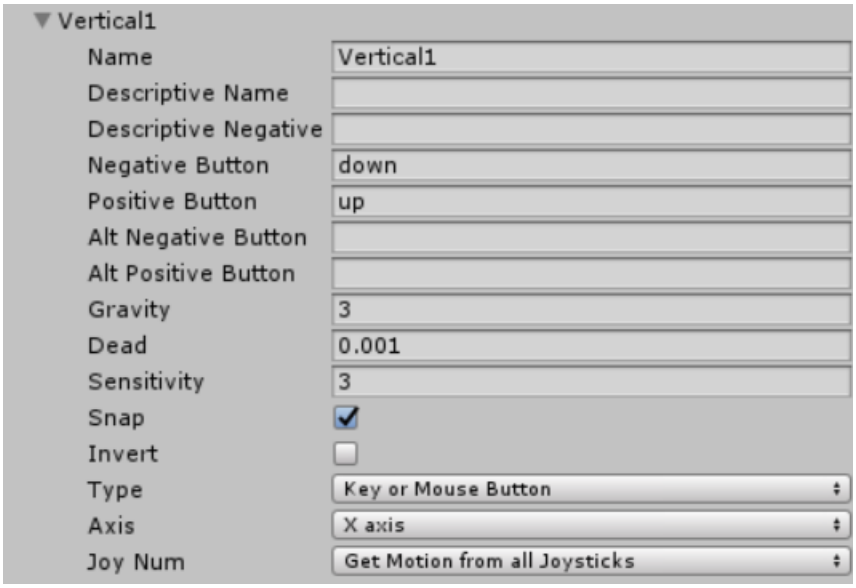


Figure 4.1: Input Manager where up and down keys are assigned

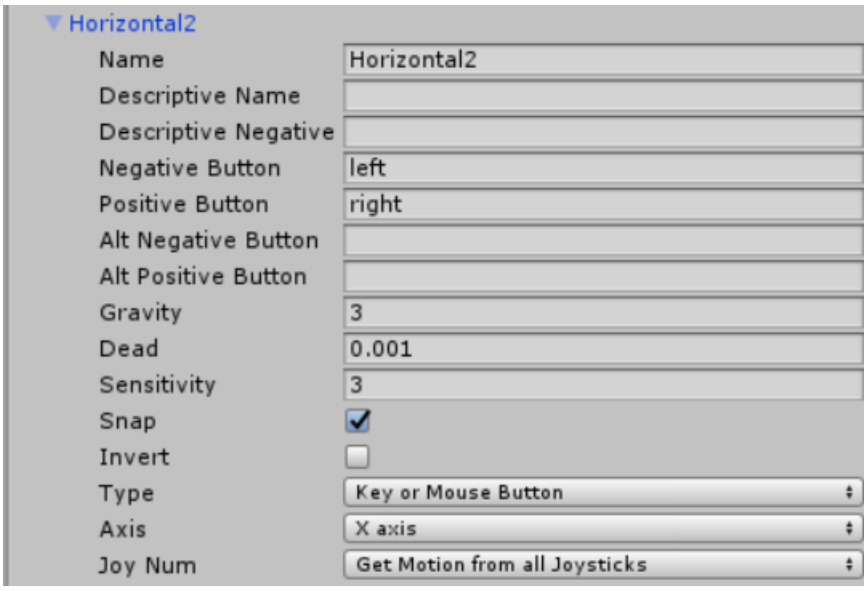


Figure 4.2: Input Manager where left and right are assigned

The protagonist can run, jump and fire projectiles at other protagonists in order to hindrance them. While the protagonists all share similar scripts, the controls had to be altered slightly so that one key would not make all the players move in sync with each other. As explained earlier I did this through the InputManager, and assigned different keys to each players particular movement. (See Figures 4.3, 4.4, **??**)

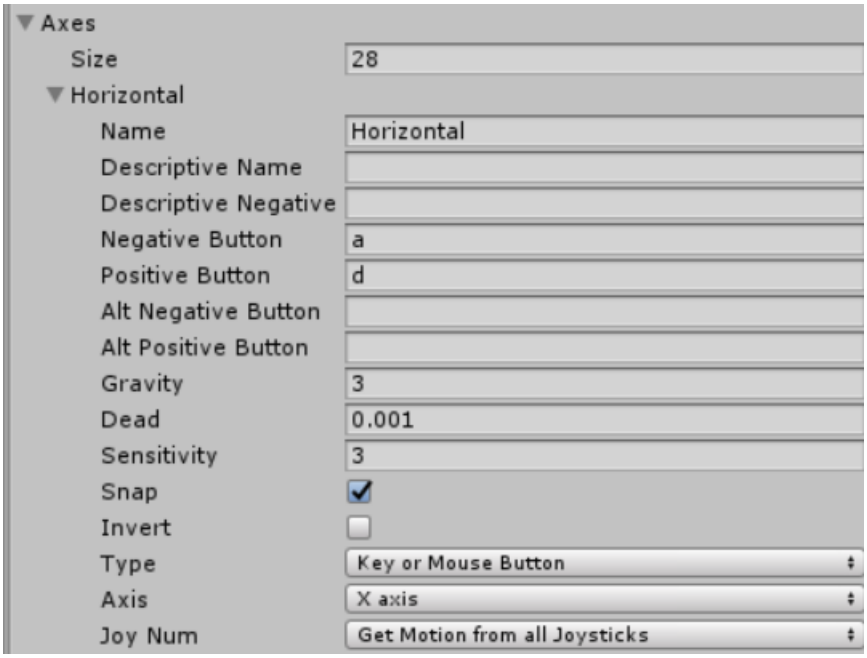


Figure 4.3: Player 1 Inputs to move left and right

As you can see from the figures above, the controls for player 1 are “a” moves him left, “d” moves him right, “s” moves him down, “w” moves him up and the space bar makes him jump.

Each player can also fire a projectile when it is picked up, unlike the previous controls this one is hardcoded into the code for convenience purposes because I needed it to comply with a public bool that was set up. So that if the player is able to fire their projectile, they can press the “tab” key to do so.

```
if (Input.GetKeyDown("tab") && canFire == true)
    IsFiring();
playerDisplay.UpdateBlue();
```

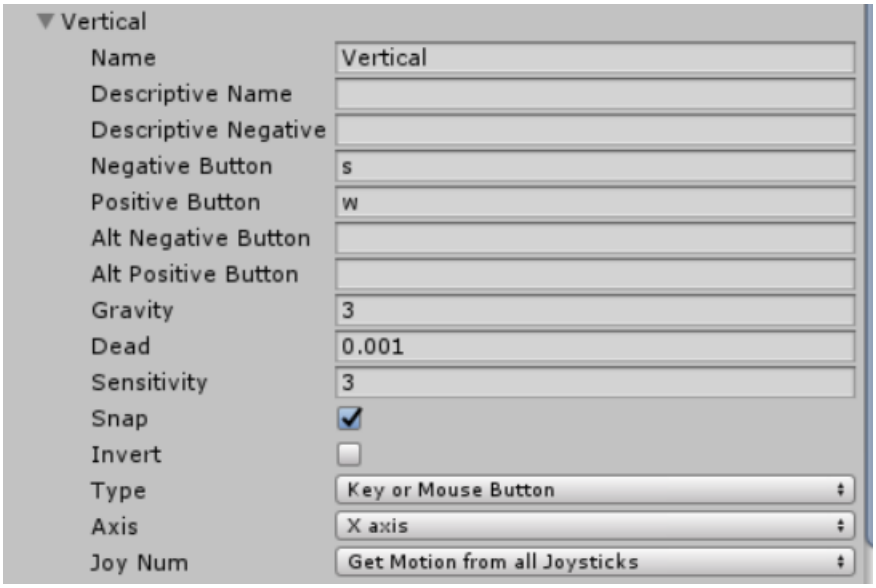


Figure 4.4: Player 1 Inputs to move up and down

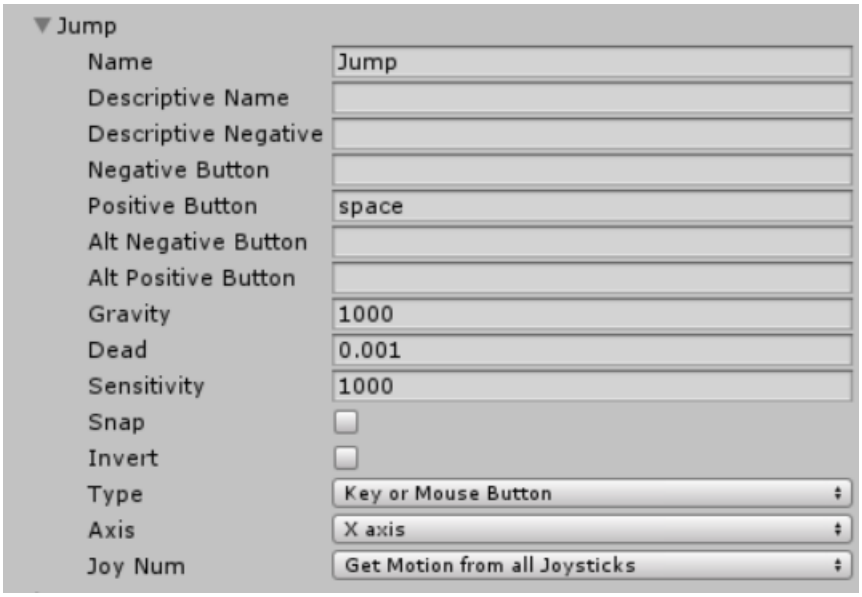


Figure 4.5: Player 1 Inputs to jump

4.3 Graphical Components & Player Display

Problem: Nowadays a lot of games are sold on their graphics and how the game looks. Graphics are used to represent what is going on in the game while hopefully catching the players eye while doing so. This includes the in game scenery and user inputs, collisions with the antagonist and with the items you can pick up should all effect what the user is seeing on screen at any moment in time.

Implementation: A lot of the skins for the in game scenery were got off the unity store and simply applied onto plain cubes and prefabs I made for the platform and the walls of the platform. As the game is the protagonist players running away from enemy fireballs, I decided to make the game look like it was set it a lava type environment.

3D Objects 3D objects included in this game are of course; the players, the platform, the walls for the platform, the pickups that spawn into the game, the antagonist's fireballs and the projectiles that are fired at other players. These are all implemented using in game objects and then applied using pre-rendered skins from the Unity Store. The pickups that are used in game are designed to only spawn in every few seconds. There are two type of pickups in game, the first being an "Ice Blast" you can pick up and then stun your enemy when you fire it at them in the form of a projectile(as mentioned before). The second pickup available is a shield, which makes you immune to all enemmy damage for 10 seconds after it is picked up.

Code to spawn in pickup blocks

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

public class spawnBlock : MonoBehaviour {

    public GameObject freeze_Block;
    public GameObject shield_Block;
    public Transform canvas;
    int spawnNum = 1;

    void SpawnFreeze()
    {
        for(int i=0; i<spawnNum; i++)
        {
            SpawnFreezeBlock();
        }
    }
```

```
    void SpawnShield()
    {
        for (int i = 0; i < spawnNum; i++)
        {
            SpawnShieldBlock();
        }
    }

    // Use this for initialization
    void Start () {
        SpawnFreeze();
        SpawnShield();

    }

    public void SpawnOneBlockAfterDelay(int delaySeconds)
    {
        Invoke("SpawnFreezeBlock", delaySeconds);
        Invoke("SpawnShieldBlock", delaySeconds);
    }

    public void SpawnFreezeBlock()
    {
        Vector3 blockPos = new Vector3(canvas.position.x + Random.Range(-5.0f, 7.5f),
                                       canvas.position.y + Random.Range(2.6f, 2.6f),
                                        canvas.position.z + Random.Range(6.5f, -6.6f));
        Instantiate(freeze_Block, blockPos, Quaternion.identity);
    }

    public void SpawnShieldBlock()
    {
        Vector3 blockPos = new Vector3(canvas.position.x + Random.Range(-5.0f, 7.5f),
                               canvas.position.y + Random.Range(2.6f, 2.6f),
                                canvas.position.z + Random.Range(6.5f, -6.6f));
        Instantiate(shield_Block, blockPos, Quaternion.identity);
    }

}
```

This code spawns one of each blocks after a certain number of time has passed. It will also spawn them within a range that is located on the platform as defined in both the “SpawnFreezeBlock()”

and "SpawnShieldBlock()" methods.

Player Display The players display is used to show the player(s) how much time is remaining in the level and if they have any pickups equipped. Every time a player collides with a block their display will update to show that they have picked up block. If the user currently has a shield picked up the shield display will turn from yes to no and if a player pick up an ice blast it will add one to their inventory that can then be fired as a projectile.

Player Display Code

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

public class PlayerDisplay : MonoBehaviour {
    public Text textBlue;
    public Text textRed;

    private PlayerManager player;

    private void Awake()
    {
        player = GetComponent<PlayerManager>();
    }

    public void UpdateBlue()
    {
        textBlue.text = "" + player.GetBlueTotal();
    }

    public void UpdateRed()
    {
        textRed.text = "No" + player.GetRedTotal();
    }

}
```

Code that updates Ice Blast

```
  blueTotal++;
  playerDisplay.UpdateBlue();
  Update();
```

Code that updates shield

```
playerDisplay.UpdateRed();
SetInvincible();

public void SetInvincible()
    {
        isInvincible = true;
        CancelInvoke("SetDamagable");
        Invoke("SetDamagable", invincibleTime);
        redNo.text = " ";
        redYes.text = "Yes";
    }

    void SetDamagable()
    {
        isInvincible = false;
        gameObject.tag = "Player";
        gameObject.layer = 12;
        redYes.text = "No";
    }
```

So if a red block is picked up it activates the shield and changes the text from “No” to “Yes” and then invokes the method “SetDamagable()” after a certain time. Once this is invoked the text will change back to “No”.

Figure 4.6: Player display before blocks are picked up

Collision Detection System

Problem: To define when in the system there is a collision between two in game objects, whether it be the player and the fireballs, the player and a projectile etc. The system has to recognise when a collision takes place and there must be different reactions depending on what objects are colliding.

Implementation: Unity is a very good engine for making it simple and convenient for picking up when two objects should interact after a collision. This is done by making sure that the “Is Trigger” option is ticked on whatever collider component has been selected.

Figure 4.7: Player display after blocks are picked up

Figure 4.8: Time Remaining

There are many different collider options that take place within this level. Being the pickups and when a player gets hit by the enemy, when a player gets hit by a projectile. With the pickups it must recognize when a player first makes contact with the block which will activate this ability.

Code for pickup abilities

```
public void OnTriggerEnter(Collider other)
   {
       if (other.CompareTag("Blue"))
       {
           Destroy(other.gameObject);
           blockSpawner.SpawnOneBlockAfterDelay(timeBetweenBlueSpawns);
           blueTotal++;
           playerDisplay.UpdateBlue();
           Update();
           print("Blue Total Equals" + blueTotal);
       }

       if (other.CompareTag("Red"))
       {
           gameObject.tag = "Invincible";
           gameObject.layer = 13;
           Destroy(other.gameObject);
           blockSpawner.SpawnOneBlockAfterDelay(timeBetweenRedSpawns);
           playerDisplay.UpdateRed();
           SetInvincible();
```

```
    }
```

This is done through different tags that have been given to the blocks, “Blue” being for the ice block and “Red” being for the shield block. Once a block is collided with it is destroyed then activates whichever method is defined within the blocks code. So when a blue block is collided with it will destroy the blue block, add one to the total number of blue blocks collected, update the players display and then call on the method “Update()” which we will talk about later when talking about the projectile system in game. When the red block is collided with it changes both the players tag and their layer. This is done to activate the invincibility on the player so enemy blocks will not destroy any object with the tag “Invincible” and will not collide or interact with anything that has the layer of “13”, this is done through the “Layer Collision Matrix” (See Figure 4.9). It will then destroy the red block and call upon the method “SetInvincible()”.

SetInvincible method

```
public void SetInvincible()
{
    isInvincible = true;
    CancelInvoke("SetDamagable");
    Invoke("SetDamagable", invincibleTime);
    redNo.text = " ";
    redYes.text = "Yes";
}
```

This code then sets the public bool for the player being invincible. It will then invoke the method “SetDamagable” after a certain amount of time has passed.

SetDamagable method

```
void SetDamagable()
{
    isInvincible = false;
    gameObject.tag = "Player";
    gameObject.layer = 12;
    redYes.text = "No";
}
```

This method will get called after the player is not invincible anymore and will set the player back to layer 12 and set the players tag back to “Player” so they can be destroyed again.

When a player collides with an enemy the player object will be destroyed.

Enemy kills player script

```
void OnTriggerEnter (Collider hit)
{
    if (hit.CompareTag("Player"))
```

```
        {
            PlayerManager playerManager = hit.gameObject.GetComponent<PlayerManager>();
            if (!playerManager.isInvincible)
            {
                Destroy(hit.gameObject);
            }
        }
    }
```

This code is placed on the fireballs and if the fireballs collide with an in game object with the tag of "Player" it will destroy that object, unless the the bool in the player manager is set to "isInvicible".

When a player collides with a projectile it must freeze them in place. To accomplish this I had the projectile set to a trigger so when in collided with a game object with a certain tag it would disable the "Player Controller" component on the player.

Script on projectile to freeze player

```
public void SetFreeze()
    {
        CancelInvoke("Unfreeze");
        this.GetComponent<CharacterController>().enabled = false;
        Invoke("Unfreeze", freezeTime);
    }
```

Method which unfreezes player

```
    void Unfreeze()
    {
        this.GetComponent<CharacterController>().enabled = true;
    }
```

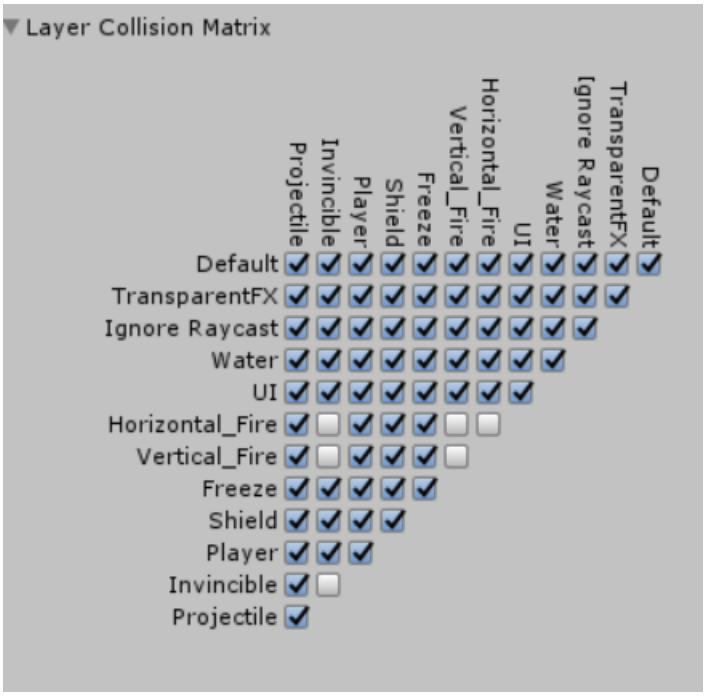


Figure 4.9: Layer Collision Matrix

4.4 Projectile System

Problem: To add another feature to the game so that there wasn't solely one pickup to the game that would benefit players.

Implementation: To give the game more depth I decided to add a small inventory system, along with a projectile that could be fired while the player had the right pickup in their inventory.

So to attempt and give one player an advantage over another for a short period of time, I decided to add a projectile that would freeze a player in place once it hits them. The main problem to try and overcome while coding this section was getting the inventory system to work effectively so that if a player had no items picked up, they could not fire their projectile. To start off with this system I added some game objects being the bullet and a weapon, a public bool which when activated meant that the player could fire their weapon and finally a float which would determine how fast the projectile would travel.

Public gameobjects, bool and float

```
public GameObject weapon;
public GameObject bullet;
public bool canFire = false;
public float Bullet_Foward_Force;
```

Then in the update method I defined that if the total of blue blocks picked up is equal or less than

0 the "canFire" bool is set to false, but if it is greater than 0 the bool is set to true. From there if the tab key is pressed it fires activates the "IsFiring()" method and the "UpdateBlue()" method found in "PlayerDisplay" class.

Update method

```
public void Update()
{
    if (blueTotal > 0)
        canFire = true;
    if (blueTotal <= 0)
        canFire = false;

    if (Input.GetKeyDown("tab") && canFire == true)
        IsFiring();
    playerDisplay.UpdateBlue();
}
```

Once the "isFiring()" method is activated it checks if the total number of blue blocks picked up is great than 0 and if it is it sets the "canFire" bool to true and takes one away from the total every time it is called. From there you then instantiate the bullet, which creates a clone of the prefab in the game. The next line of code retrieves the Rigidbody component and the bullet is told to be pushed forward by whatever is set in the public float.

Is firing method

```
public void IsFiring()
{
    if (blueTotal > 0 && canFire == true)
       blueTotal--;
    GameObject Temporary_Bullet_Handler;

    Temporary_Bullet_Handler=Instantiate(bullet, weapon.transform.position, weapon.transform.rotati
    Temporary_Bullet_Handler.transform.Rotate(Vector3.left * 90);

    Rigidbody Temporary_RigidBody;
    Temporary_RigidBody = Temporary_Bullet_Handler.GetComponent<Rigidbody>();
    Temporary_RigidBody.AddForce(transform.forward * Bullet_Foward_Force);

    if(blueTotal ==0)
    {
        canFire = false;
    }
```

```
}
```

5

Testing

6

Conclusions

Part I

Appendices

A

Screen Shots

B

Program Listings

B.1 List of programs Included

- Buttons.cs
- CountDownTimer.cs
- DestroyPlayer.cs
- EnemyNav2.cs
- EnemyNav3.cs
- GameManager.cs
- PlayerDisplay.cs
- PlayerManager.cs
- spawnBlock.cs

```
using UnityEngine;
using System.Collections;
using UnityEngine.SceneManagement;

public class Buttons : MonoBehaviour
{
    public void MENU_ACTION_LoadDemo_GamePlaying() // method to call inside unity
```

```
    {
        SceneManager.LoadScene("Demo");  // load the scene inside the brackets
    }
}
```

```
using UnityEngine;
using System.Collections;

public class CountdownTimer : MonoBehaviour
{
    private float countdownTimerStartTime;
    private int countdownTimerDuration;

    public int GetTotalSeconds()
    {
        return countdownTimerDuration;
    }

    public void ResetTimer(int seconds)
    {
        countdownTimerStartTime = Time.time;
        countdownTimerDuration = seconds;
    }

    public int GetSecondsRemaining()
    {
        int elapsedSeconds = GetElapsedSeconds();
        int secondsLeft = (countdownTimerDuration - elapsedSeconds);
        return secondsLeft;
    }

    public int GetElapsedSeconds()
    {
        int elapsedSeconds = (int)(Time.time - countdownTimerStartTime);
        return elapsedSeconds;
    }

    public float GetProportionTimeRemaining()
    {
        float proportionLeft = (float)GetSecondsRemaining() / (float)GetTotalSeconds();
        return proportionLeft;
```

```
    }
}
```

```
using UnityEngine;
using System.Collections;

public class DestroyPlayer : MonoBehaviour
{

    void OnTriggerEnter (Collider hit)
    {
        if (hit.CompareTag("Player"))
        {
            PlayerManager playerManager = hit.gameObject.GetComponent<PlayerManager>();
            if (!playerManager.isInvincible)
            {
                Destroy(hit.gameObject);
            }
        }
    }
}
```

```
using UnityEngine;
using System.Collections;

public class EnemyNav2 : MonoBehaviour
{
    public float speed = 6.0F;
    public float gravity = 20.0F;
    private Vector3 moveDirection = Vector3.zero;
    void Update()
    {
        CharacterController controller = GetComponent<CharacterController>();
        if (controller.isGrounded)
        {
            moveDirection = new Vector3(Input.GetAxis("Horizontal1"), 0, Input.GetAxis("Vertical1"));
            moveDirection = transform.TransformDirection(moveDirection);
            moveDirection *= speed;

        }
        moveDirection.y -= gravity * Time.deltaTime;
        controller.Move(moveDirection * Time.deltaTime);
```

```
    }
}
```

```
using UnityEngine;
using System.Collections;

public class EnemyNav3 : MonoBehaviour
{
    public float speed = 6.0F;
    public float gravity = 20.0F;
    private Vector3 moveDirection = Vector3.zero;
    void Update()
    {
        CharacterController controller = GetComponent<CharacterController>();
        if (controller.isGrounded)
        {
            moveDirection = new Vector3(Input.GetAxis("Horizontal2"), 0, Input.GetAxis("Vertical
            moveDirection = transform.TransformDirection(moveDirection);
            moveDirection *= speed;

        }
        moveDirection.y -= gravity * Time.deltaTime;
        controller.Move(moveDirection * Time.deltaTime);
    }
}
```

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    public Text Time_Remaining;
    private CountdownTimer countDownTimer; //reference to countdown timer
    private int totalSeconds = 60;

    void Start()
    {
        countDownTimer = GetComponent<CountdownTimer>();//
        countDownTimer.ResetTimer(totalSeconds);//
    }
```

```
    void Update()
    {
        string msg = "" + countDownTimer.GetSecondsRemaining();
        Time_Remaining.text = msg;
        checkGameOver();
    }

    private void checkGameOver()
    {
        if(countDownTimer.GetSecondsRemaining()<0)
        {
            SceneManager.LoadScene("scene1_Over");
        }
    }
}
```

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

public class PlayerDisplay : MonoBehaviour {
    public Text textBlue;
    public Text textRed;

    private PlayerManager player;

    private void Awake()
    {
        player = GetComponent<PlayerManager>();
    }

    public void UpdateBlue()
    {
        textBlue.text = "" + player.GetBlueTotal();
    }

    public void UpdateRed()
    {
        textRed.text = "No" + player.GetRedTotal();
```

```
    }

}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PlayerManager : MonoBehaviour
{

    private int blueTotal = 0;
    private int timeBetweenBlueSpawns = 10;
    public Text redNo;
    public Text redYes;
    private int timeBetweenRedSpawns = 10;

    public float freezeTime = 3f;
    public float invincibleTime = 10f;
    public bool isInvincible = false;

    public spawnBlock blockSpawner;
    public GameObject inventoryPanel;
    public GameObject[] inventoryIcons;
    private PlayerDisplay playerDisplay;

    public GameObject weapon;
    public GameObject bullet;
    public bool canFire = false;
    public float Bullet_Foward_Force;

    public void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Blue"))
        {
            Destroy(other.gameObject);
            blockSpawner.SpawnOneBlockAfterDelay(timeBetweenBlueSpawns);
            blueTotal++;
            playerDisplay.UpdateBlue();
            Update();
```

```
            print("Blue Total Equals" + blueTotal);
        }

        if (other.CompareTag("Red"))
        {
            gameObject.tag = "Invincible";
            gameObject.layer = 13;
            Destroy(other.gameObject);
            blockSpawner.SpawnOneBlockAfterDelay(timeBetweenRedSpawns);
            playerDisplay.UpdateRed();
            SetInvincible();
        }

        if(other.CompareTag("Projectile"))
        {
            SetFreeze();
            DestroyObject(other.gameObject);
        }
    }

    public void SetInvincible()
    {
        isInvincible = true;
        CancelInvoke("SetDamagable");
        Invoke("SetDamagable", invincibleTime);
        redNo.text = " ";
        redYes.text = "Yes";
    }

    public void SetFreeze()
    {
        CancelInvoke("Unfreeze");
        this.GetComponent<CharacterController>().enabled = false;
        Invoke("Unfreeze", freezeTime);
    }

    void SetDamagable()
    {
        isInvincible = false;
        gameObject.tag = "Player";
        gameObject.layer = 12;
```

```
    redYes.text = "No";
}

void Unfreeze()
{
    this.GetComponent<CharacterController>().enabled = true;
}

public int GetBlueTotal()
{
    return blueTotal;
}

public string GetRedTotal()
{
    return redYes.text;
}

public void Start()
{
    playerDisplay = GetComponent<PlayerDisplay>();
    playerDisplay.UpdateBlue();
    playerDisplay.UpdateRed();
}


public void Update()
{
    if (blueTotal > 0)
        canFire = true;
    if (blueTotal <= 0)
        canFire = false;

    if (Input.GetKeyDown("tab") && canFire == true)
        IsFiring();
    playerDisplay.UpdateBlue();
}

public void IsFiring()
{
    if (blueTotal > 0 && canFire == true)
```

```
            blueTotal--;
        GameObject Temporary_Bullet_Handler;

        Temporary_Bullet_Handler=Instantiate(bullet, weapon.transform.position, weapon.transform.rotati
        Temporary_Bullet_Handler.transform.Rotate(Vector3.left * 90);

        Rigidbody Temporary_RigidBody;
        Temporary_RigidBody = Temporary_Bullet_Handler.GetComponent<Rigidbody>();
        Temporary_RigidBody.AddForce(transform.forward * Bullet_Foward_Force);

        if(blueTotal ==0)
        {
            canFire = false;
        }

    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class spawnBlock : MonoBehaviour {

    public GameObject freeze_Block;
    public GameObject shield_Block;
    public Transform canvas;
    int spawnNum = 1;

    void SpawnFreeze()
    {
        for(int i=0; i<spawnNum; i++)
        {
            SpawnFreezeBlock();
        }
    }

    void SpawnShield()
    {
        for (int i = 0; i < spawnNum; i++)
        {
```

```
            SpawnShieldBlock();
        }
    }

    // Use this for initialization
    void Start () {
        SpawnFreeze();
        SpawnShield();

    }

    public void SpawnOneBlockAfterDelay(int delaySeconds)
    {
        Invoke("SpawnFreezeBlock", delaySeconds);
        Invoke("SpawnShieldBlock", delaySeconds);
    }

    public void SpawnFreezeBlock()
    {
        Vector3 blockPos = new Vector3(canvas.position.x + Random.Range(-5.0f, 7.5f),
                                       canvas.position.y + Random.Range(2.6f, 2.6f),
                                        canvas.position.z + Random.Range(6.5f, -6.6f));
        Instantiate(freeze_Block, blockPos, Quaternion.identity);
    }

    public void SpawnShieldBlock()
    {
        Vector3 blockPos = new Vector3(canvas.position.x + Random.Range(-5.0f, 7.5f),
                               canvas.position.y + Random.Range(2.6f, 2.6f),
                                canvas.position.z + Random.Range(6.5f, -6.6f));
        Instantiate(shield_Block, blockPos, Quaternion.identity);
    }

}
```

List of References