Technical Guide

Project Name: Collider

Student Name: Ore Ibikunle

Student Number: 15351216

Supervisor Name: Charles Daly

Date Finished: 19/05/2019

Table of Contents

Glossary

- **Sprite** – is a computer graphic which can be moved on-screen and otherwise manipulated by a single entity.
- **Sprite Sheet** – is an image that consists of several smaller images (**sprites**) and/or animations.
- **Spawning** – is a live creation of an object.
- **Vector** – is a quantity that has a magnitude and direction. It is used to determine the position of one point in space relative to another.
- **Euler Angles** – are three angles used to describe the orientation of a rigid body with respect to a fixed coordinate system.
- **Components** – are classes that you create and are attached to game objects in Unity to add functionality.
- **Collision** – an instance of one moving object striking violently against another.
- **Collision Detection** – is the computational problem of detecting the intersection of two or more objects.
- **Lag** – is a time delay between a player's action and the game's reaction to that input.
- **Health** – is an attribute assigned to entities such as characters and objects in games that indicate their continued ability to function. It is usually measured in Health Points (HP), which lowers by set amounts when the entity is attacked or injured.
- **Mothership** – is a large spacecraft or ship.

Introduction

Abstract

For my fourth-year project, I have developed an online 2-D space shooting game built using the Unity Engine. The game is based on classic arcade games like Space Invaders and Galaga. The objective of the player is to battle and destroy incoming enemy ships. The player will have health points and can collect hearts to regain health. The game has features such as collision detection, dynamic difficulty adjustment, and an online database where players will be ranked based on their scores.

For this project, I plan to demonstrate an in depth understanding and showcase the fundamental design, testing and optimization techniques that are used in game development with the Unity Engine.

Motivation

Growing up, I have always loved playing games, and it was my interest in games that actually allowed me to develop an interest in computer science. Although I played a lot of games, as a computer science student, I had a very basic understanding of game development leading up to fourth year. I really enjoy independent learning and the course did not teach game development to us as a module, so my goal was to do a project that would allow me to develop an understanding of game development.

I also have other creative interests such as music, photo editing, and video editing. These other interests are very applicable in game development. Games are also very popular and can generate a lot of revenue, so I am also interested in the profitable side of game development.

Research

Going into this, I had very limited knowledge of the best tools to use for designing a high-quality game. At first, I opted to develop the game in *Java* using the *Eclipse* IDE, but then I realized that it would not suffice for the game I wanted to develop. I then decided to use the *Unity Engine*. I felt like it was better suited towards my needs and the type of game that I wanted to develop for two reasons:

1. I wanted the GUI for the game to be well designed and the *Unity Engine* was better suited for that.
2. I wanted the game to have animations and sprites. Doing this with *Unity* saves a lot of time so that I can focus on the more important aspects of the project.

- C# Programming Language and Unity Engine:

  During my internship at *Spanish Point Technologies* I was part of the development team and we primarily used C#. So, I already had a good understanding of the syntax of the language. Although implementing C# in the *Unity Engine* is a bit different, having a general understanding of the syntax allowed me to quickly debug errors.

  I had to do a lot of research about the physics engine and how to use it because the structure in which code is designed is very different. For example, the way instance variables where initialised where different, i.e. constructors are not used in classes that derive from the *Monobehaviour* class. These were the type of changes that I had to get used to.

  I had also to accustom myself to some of the physics terminology used in game development.

  Reference: Unity API

- Design Pattern and Types of Testing:

  I had to establish myself with the type of testing that is commonly used when developing games in *Unity*. The most common type of testing in software engineering is *Test Driven Development*. In *Unity* however, *Data Driven Development* is used because it involves testing the behaviour of the game objects when they are attached to their components, rather than what is done in TDD, where you just write tests to validate if functions work correctly.

  I also had to design the game in a way that I could test it. For this I had to follow and be accustomed to the *Humble Object Pattern*.

  References:

  Data Driven Development [Skip to 7:00]

  Humble Object Pattern

- Maths and Physics using the Unity Engine:

  The physics engine in *Unity* is very complicated. If you do not have a general understanding of physics, it is very difficult to use the engine. So, I made sure to grasp a general understanding of some of the theorems in physics that would be useful towards my project, (i.e. *Euler Angles* and *Vector Maths*).

  References:

  Euler Angles

  Vector Maths

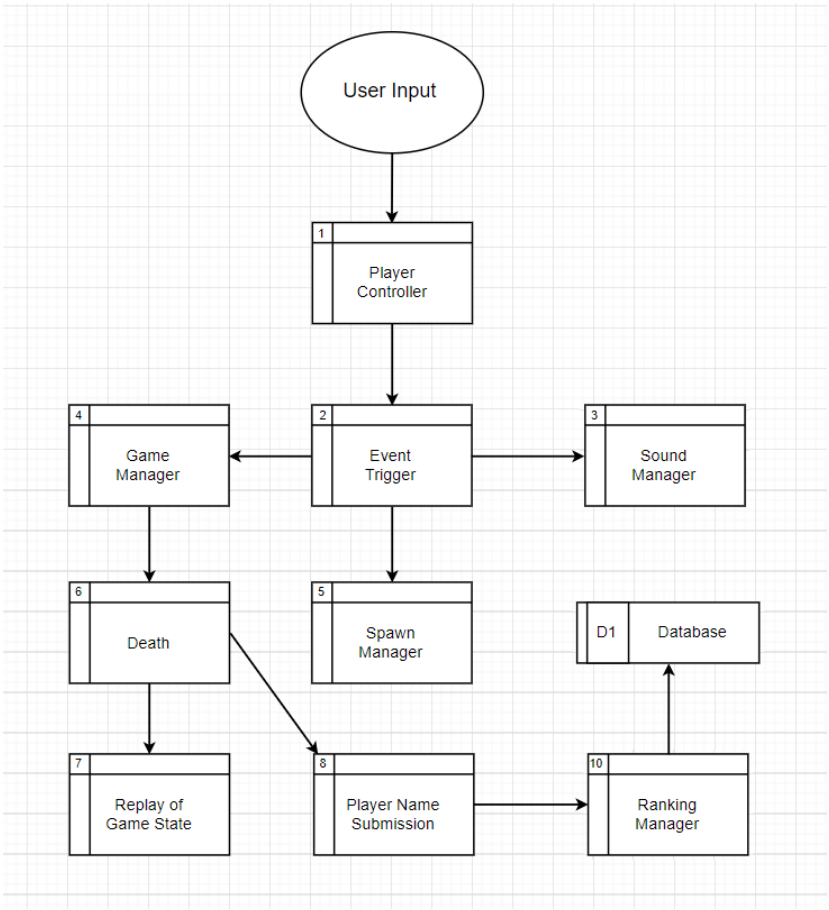
Design

Originally, the aim of the game was for the player to manoeuvre a spaceship around the screen and try to avoid obstacles. These obstacles were asteroids and meteoroids. The idea was that when the meteoroids collided with the asteroids, the trajectory of the smaller asteroids would change, making it harder for the player to avoid. The player could destroy meteoroids but not asteroids and the game would become more difficult as the player continues, e.g. both types of asteroids move faster with other levels of complexity.

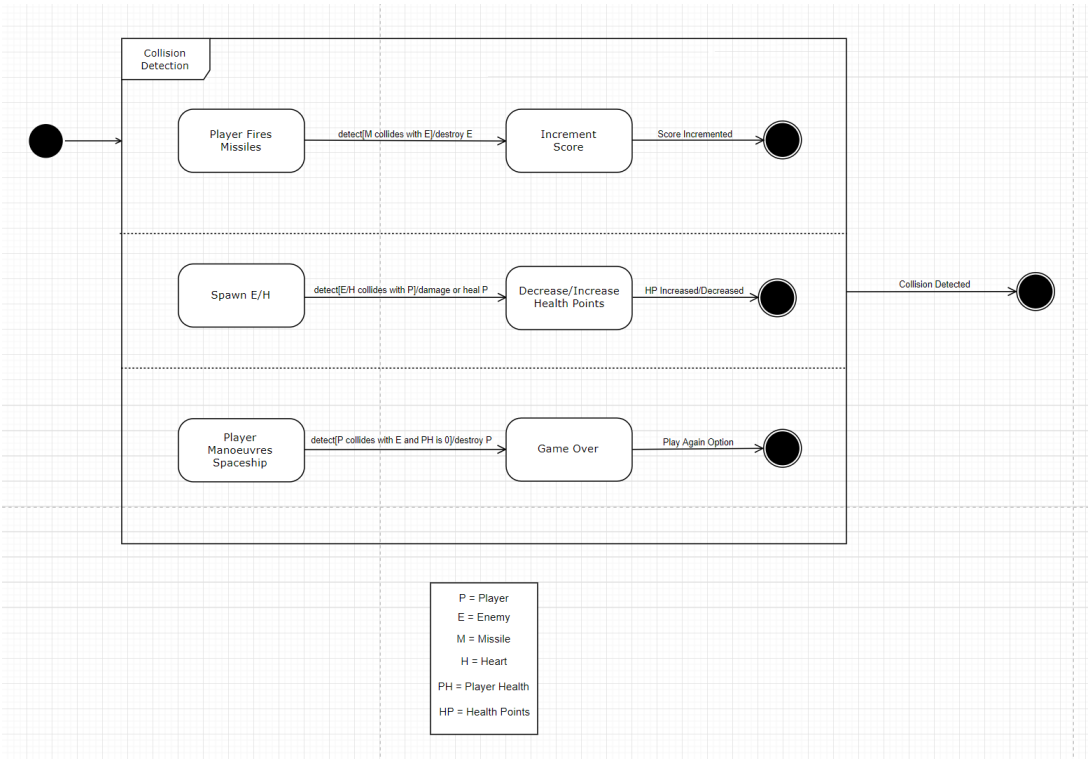
However, after some thought and some careful consideration, I concluded that this wouldn't work because as a person who plays games myself, I wanted the game to be fun. I felt like the idea of the player moving around trying to avoid asteroids and meteoroids that were colliding with each other was a bit too basic and that I need to add more functionality to the game (i.e. more enemies). Since I was using the *Unity Engine*, I felt like I could add a lot more complexity to my game, i.e. types of enemies that behave differently to each other. For players to want to play the game, it's important that the enemies displayed at least some level of intelligence.

Initially, since I planned to develop the game using *Eclipse*, my design was tailored towards the *Eclipse* IDE. So, after understanding the *Unity Engine*, I made some crucial design changes.

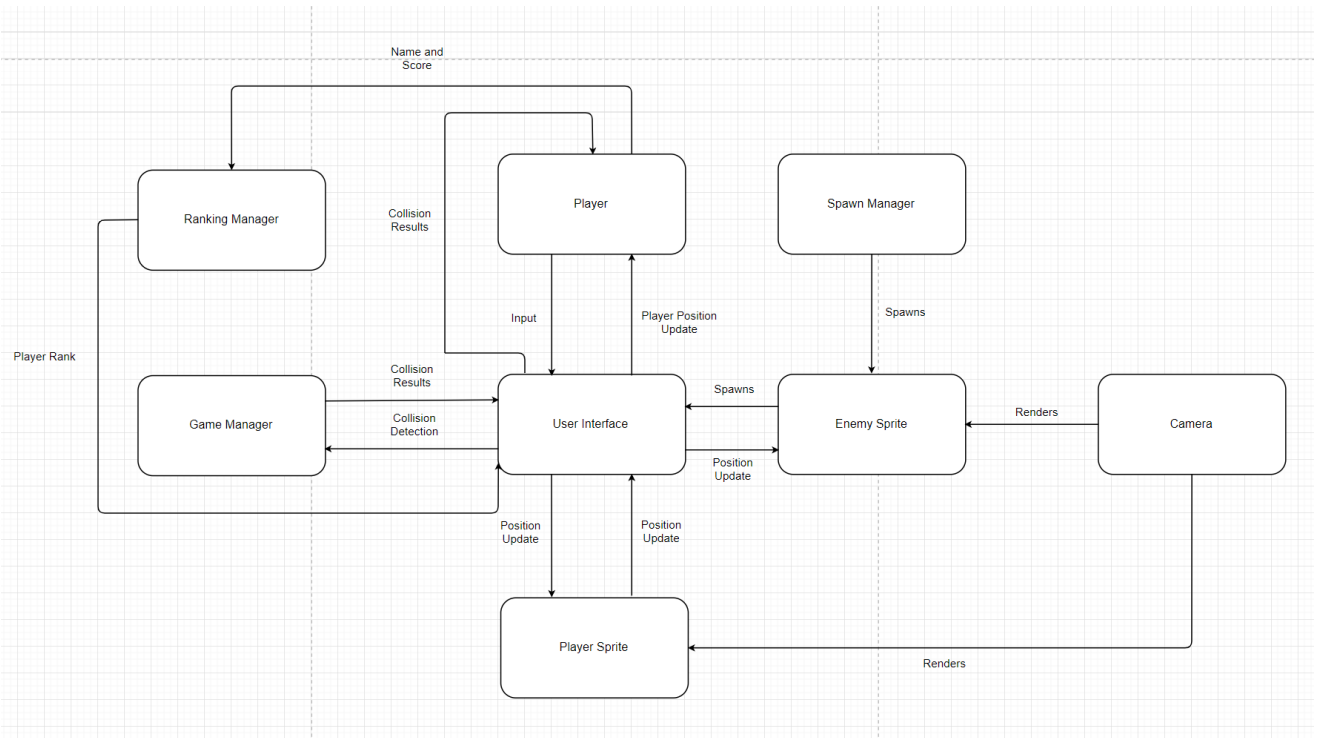
Level 0 DFD



State Machine



Context Diagram



Implementation

To begin the game, I first set up the menu scene and started with a very basic game scene to add user input. I did some research on what the most common keyboard/mouse inputs were for computer games. For games, the most common layout for user inputs is based on *Fitts's* Law (discusses thoroughly in my 6th blog).

- Fitts's Law – is predictive model of human movement primarily used in human-computer interaction and ergonomics. To keep it short and simple, the fewer the distance to the button and the bigger the button, the more accessible the button is.

  Reference: Fitts's Law

Based on this law, I decided that the controls for the game would be:

- WASD or Arrow Keys to move the player.
- Space bar to fire missiles.
- Escape button to pause.

I added the option of allowing the arrow keys to move the player to give the players that are primarily right-handed an option simply because people who don't commonly play computer games and are right-handed may not be used to using their left hand for the WASD keys.

Since the design theme of the game was going to be centred around space, I focused on adding some features in the background to make the player feel like they are really travelling through space. The scene consists of a moving background, planets, space rocks, and stars. I only added one game scene because I wanted to give it that classic arcade feeling, to give the player a feeling of nostalgia of the days that they used to play games such as Space Invaders. I didn't want the background to be basic, so I made it possible to add as many background objects to the game as I wanted without the game lagging. To implement this, I used *Object Pooling*.

- Object Pooling – is when you reuse objects that have been already instantiated before gameplay, rather than allocating and destroying them on demand.

To add sound to the game, I made my own sound manager class. *Unity* has a component that is already made called an Audio Source. This component is responsible for playing an audio clip. It has properties such as the:

- Volume – the volume of the audio source.
- Pitch – the pitch of the audio source.
- Spatial blend – to control whether or not you want the sound to be played in 2D or 3D space.

This audio source component can be added to each object and you can make a call to this component when it is attached to a game object.

I opted not to use the Audio Source component because it is very inefficient. For every object that I added to my scene, if I wanted to add sound to that object, I would have to repeat the process of adding the audio source component to each object one by

one. There could be hundreds of objects in a game, so doing it this way would obviously be time consuming. There are also sounds that are not tied to game objects, i.e. the “game over” voice you hear when you lose a game. This would mean that I would have to create empty game objects and add the audio source component to each of them and then access the sound through code on that game object. The idea of my sound manager class is that it will hold a list of *Sounds* (a class I created) that I can easily add and remove. When I want to add a sound or music to my game, I just have to add it to the list of sounds in my SoundManager class which is attached to my SoundManager object. This class will hold a set of public functions that I can call outside this class to play or pause specific sounds. When I call a function from the class, it will iterate through the list and find the sound with the appropriate name.

As aforementioned, I planned to make some changes to my original idea for my game. Here are some of the new ideas that I added to my game:

- The player will have 3 lives in total and 100 HP **per life.**
- I will be removing the asteroid and meteoroid enemies and adding enemy ships instead.
- I plan to have 4 enemy ships and 1 mother ship. They will fire missiles at the player. All enemies (including mother ship) will display some level of intelligence. Direct collision between an enemy ship and player or an enemy missile and a player cause the player to lose health points. Collision with an enemy ship cause the player to lose more health points than collision with enemy missiles. Collision with the mother ship causes more damage to the player than collision with a normal enemy ship. This mother ship will also move faster than normal enemy ships and have faster missiles. The enemies will also have health points that will decrease when they are attacked by the players missiles.
- Like my original idea, there will still be an online leader board where players can post their scores and view the scores of other players.
- The player will be able to pick up lives while playing to ensure that they don't die so quickly.

When creating the enemies, I wanted to showcase the power of the *Unity Engine* and how much control it can give you as a game developer. To do this I wanted to show some form of polymorphism with the engine. I wanted the enemies to have set of core components that they all have attached to their game object (these components are explained thoroughly in my 14th blog). Having the enemies use a set of core specific components would allow me to add enemies very quickly. Some of these components are also reusable for other objects (i.e. my *BlinkObject* component). The different types of enemies and their behaviours are discussed blog 17.

Enemies and heart collectables will be spawned by my *SpawnManager* component. At the beginning, I was spawning objects at random positions, this was ok, but it doesn’t make the game interesting from the players perspective. So, I decided to change it and make objects spawn based on the players current position.

I did this for 3 reasons:

1. It makes the game more interesting.
2. To give the player an advantage, so that the heart collectables are spawned based on their position, making it easier for them to regain some HP.
3. To also add more difficulty because now enemies will also spawn based on the players position.

To add more intelligence to my *SpawnManager*, I decided to make objects spawn based on weights. To implement this, I used the *Cumulative Distribution Function*.

Why I decided to use weights:

1. For example, since the *SpawnManager* has a list of all the enemy object, if the player has just started the game and I was to select an enemy from this list at random without using weights, there is a chance that the mother ship will be the first enemy spawned. I didn't want something like this to happen, because this mother ship enemy is the boss. In games, the player should only have to fight against the boss if they are performing really well.
2. Having weights allowed me to add an element of dynamic difficulty adjustment, because now I can give the player even more of a chance to perform well. For example, increasing the probability of the heart being spawned if they have very low HP.
3. Using weights, I could also increase the chance of the mother ship being spawned if the player is performing really well.

The weight of the heart collectable will be adjusted based on the number of lives and HP that the player has left. For example, if the player was performing bad in the game and they had a very low number of lives and HP remaining then there is a higher chance that a heart will be spawned. Whereas, the mother ship's weight will be adjusted based on the score of the current player and the average of the current high scores online.

To put the game online, I used a No SQL and No PHP server called *DreamLo*. I used HTTP GET requests in C# to fetch and put the scores and the player names on the sever.

Reference: DreamLo Server

To finish off the GUI, after all enemies were created, I made a little video, which I used for the background of the main menu. I made some of the images in the GUI myself using my photoshop skills. One of the key aspects of game development is design, which is why I wanted to also show off some of my non-programming skills as well.

Sample Code

- *BackgroundObjectPooler.cs*

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

public class BackgroundObjectPooler : MonoBehaviour
{
    private Queue<GameObject> queueForBackgroundObjects;
    public GameObject[] arrayForBackgroundObjects;


    private void Start()
    {
        BuildQueue();

        InvokeRepeating("DequeueStationaryObjects", 0,25f);
    }


    private Queue<GameObject> BuildQueue()
    {
        queueForBackgroundObjects = new Queue<GameObject>();

        for (int i = 0; i < arrayForBackgroundObjects.Length; i++)
        {
            queueForBackgroundObjects.Enqueue(arrayForBackgroundObjects[i]);
        }

        return queueForBackgroundObjects;
    }


    private void EnqueueStationaryObjects()
    {
        int i = 0;
        while (i < arrayForBackgroundObjects.Length)
        {
            if((!arrayForBackgroundObjects[i].GetComponent<BackgroundObject>().IsNotStationary) && (0 > arrayForBackgroundObjects[i].transform.position.y))
            {
                arrayForBackgroundObjects[i].GetComponent<BackgroundObject>().RetransformBOPosition();
                queueForBackgroundObjects.Enqueue(arrayForBackgroundObjects[i]);
            }
            i++;
        }
    }


    private void DequeueStationaryObjects()
    {
        EnqueueStationaryObjects();

        if(queueForBackgroundObjects.Count != 0)
        {
            GameObject instanceOfBO = queueForBackgroundObjects.Dequeue();

            instanceOfBO.GetComponent<BackgroundObject>().IsNotStationary = true;
        }
        else if(queueForBackgroundObjects.Count == 0)
        {
            return;
        }
    }
}
```

- *Playership.cs*

```
using System.Collections;
using UnityEngine;

public class PlayerShip : MonoBehaviour
{
    public GameObject playerShipMissileMovement;
    public GameObject playerMissilePosition1;
    public GameObject playerMissilePosition2;
    public GameObject playerExplosionAnimationObject;

    public Boundary boundary;

    private int maxNumMissiles;
    private int currentNumMissiles;
    private float reloadTime;

    private bool isReloading;

    private void Awake()
    {
        this.maxNumMissiles = 5;
        this.reloadTime = 0.5f;
        this.isReloading = false;
    }


    private void Start()
    {
        this.currentNumMissiles = this.maxNumMissiles;
    }


    private void Update()
    {
        FireMissiles();
        MoveInDirection();
    }


    private void MoveInDirection()
    {
        Vector2 position = FindObjectOfType<PlayershipController>().playerPosition;

        float ratio = (float)Screen.width / (float)Screen.height;
        float orthographicWidth = ratio * Camera.main.orthographicSize;

        if(position.x + this.boundary.xMax > orthographicWidth)
        {
            position.x = orthographicWidth - this.boundary.xMax;
        }

        if(position.x - this.boundary.xMin < -orthographicWidth)
        {
            position.x = -orthographicWidth + this.boundary.xMin;
        }

        if(position.y + this.boundary.yMax > Camera.main.orthographicSize)
        {
            position.y = Camera.main.orthographicSize - this.boundary.yMax;
        }

        if(position.y - this.boundary.yMin < -Camera.main.orthographicSize)
        {
            position.y = -Camera.main.orthographicSize + this.boundary.yMin;
        }

        transform.position = position;
    }
```

```
    private void OnTriggerEnter2D(Collider2D collider)
    {
        if(collider.tag == "EnemyTag" || collider.tag == "EnemyMissileTag")
        {
            FindObjectOfType<HealthPoints>().DecreaseHP(FindObjectOfType<DamagePoints>().damagePoints);
            if(FindObjectOfType<HealthPoints>().currentLives == 0)
            {
                Destroy(gameObject);
                PlayershipExplosion();
                FindObjectOfType<SoundManager>().Play("PlayerDies");
                FindObjectOfType<GameManager>().GameOver();
            }
            gameObject.GetComponent<BlinkObject>().BlinkColor = Color.red;
            gameObject.GetComponent<BlinkObject>().Blink();
            FindObjectOfType<SoundManager>().Play("PlayerDamage");
        }
    }


    private void PlayershipExplosion()
    {
        GameObject instanceOfExplosion = (GameObject)Instantiate(playerExplosionAnimationObject);
        instanceOfExplosion.transform.position = transform.position;
    }


    private IEnumerator ReloadNumMissiles()
    {
        this.isReloading = true;

        yield return new WaitForSeconds(this.reloadTime);
        this.currentNumMissiles = this.maxNumMissiles;

        this.isReloading = false;
    }


    private void FireMissiles()
    {
        if(this.isReloading)
        {
            return;
        }

        if(this.currentNumMissiles <= 0)
        {
            StartCoroutine(ReloadNumMissiles());
            return;
        }

        if(Input.GetKeyDown(KeyCode.Space) && !GameMenu.Paused)
        {
            FindObjectOfType<SoundManager>().Play("PlayerShoots");

            this.currentNumMissiles--;

            GameObject missileAtPosition1 = (GameObject)Instantiate(this.playerShipMissileMovement);
            missileAtPosition1.transform.position = this.playerMissilePosition1.transform.position;
            missileAtPosition1.transform.parent = transform;

            GameObject missileAtPosition2 = (GameObject)Instantiate(this.playerShipMissileMovement);
            missileAtPosition2.transform.position = this.playerMissilePosition2.transform.position;
            missileAtPosition2.transform.parent = transform;
        }
    }
}
```

- *SpawnManager.cs*

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

public class SpawnManager : MonoBehaviour
{
    public List<CustomObject> objectsList;

    public float startTime;
    public float maxSpawnRatePerSecond;
    public float minSpawnRatePerSecond;
    public float timeToIncrease = 180;

    GameObject playership;
    GameObject score;
    Vector3 positionOfCamera;

    int[] originalWeights;

    int average;

    private void Awake()
    {
        this.positionOfCamera = Camera.main.transform.position;
        this.playership = GameObject.Find("PlayershipMove");
        this.score = GameObject.Find("ScoreText");
    }


    private void Start()
    {
        StartSpawning();
        this.originalWeights = OriginalWeight(objectsList);
        if(PlayerPrefs.GetInt("AverageScore") == 0)
        {
            this.average = 10000;
        }
        else
        {
            this.average = PlayerPrefs.GetInt("AverageScore");
        }
    }


    private void Update()
    {
        AdjustWeights();
    }


    public void StartSpawning()
    {
        Invoke("SpawnObjectOnScreen", this.startTime);

        InvokeRepeating("IncreaseRate", 0f, this.timeToIncrease);
    }



        public void StopSpawning()
        {
            CancelInvoke("SpawnObjectOnScreen");
            CancelInvoke("IncreaseRate");
        }
```

```
private void IncreaseRate()
{
    if(this.minSpawnRatePerSecond == this.maxSpawnRatePerSecond)
    {
        CancelInvoke("IncreaseRate");
    }

    if(this.minSpawnRatePerSecond < this.maxSpawnRatePerSecond)
    {
        this.maxSpawnRatePerSecond--;
    }
}


private void ScheduleSpawn()
{
    float spawnPerSeconds;

    if(this.minSpawnRatePerSecond < this.maxSpawnRatePerSecond)
    {
        spawnPerSeconds = this.maxSpawnRatePerSecond;
    }

    else
    {
        spawnPerSeconds = this.minSpawnRatePerSecond;
    }

    Invoke("SpawnObjectOnScreen", spawnPerSeconds);
}


private GameObject ObjectWithBestProbrability()
{
    int total = 0;
    for(int i = 0; i < objectsList.Count; i++)
    {
        total += objectsList[i].weight;
    }

    int rand = Random.Range(0, total);
    GameObject instanceOfObject = null;
    foreach (CustomObject co in objectsList)
    {
        if(rand < co.weight)
        {
            instanceOfObject = co.customeObject;
            break;
        }
        rand -= co.weight;
    }
    return instanceOfObject;
}
```

```
    public int[] OriginalWeight(List<CustomObject> lst)
    {
        int[] weights = new int[lst.Count];

        for(int i = 0; i < lst.Count; i++)
        {
            weights[i] = lst[i].weight;
        }

        return weights;
    }


    public void ResetWeights()
    {
        for(int i = 0; i < this.objectsList.Count; i++)
        {
            this.objectsList[i].weight = this.originalWeights[i];
        }
    }


    public void AdjustWeights()
    {
        int max = objectsList.Max(m => m.weight);
        int maxIndex = 0;
        for(int i = 0; i < objectsList.Count; i++)
        {
            if(objectsList[i].weight == max)
            {
                maxIndex = i;
            }

            if(playership != null)
            {
                if(playership.GetComponent<HealthPoints>().currentLives < 2 &&
playership.GetComponent<HealthPoints>().DamageHP < 75)
                {
                    if(objectsList[i].customeObject.name.Equals("HeartPickUp"))
                    {
                        objectsList[maxIndex].weight = Mathf.Clamp(objectsList[i].weight+1,1,1);
                        objectsList[i].weight = max;
                    }
                }

                if(playership.GetComponent<HealthPoints>().currentLives > 1 &&
playership.GetComponent<HealthPoints>().DamageHP > 10)
                {
                    ResetWeights();
                }

                if(this.score != null)
                {
                    if(this.score.GetComponent<PlayerScore>().Score > this.average)
                    {
                        if(objectsList[i].customeObject.name.Equals("MotherShip"))
                        {
                            objectsList[maxIndex].weight = 21;
                            originalWeights[i] = 8;
                        }
                    }
                }
            }
        }
    }
```

```
    private void SpawnObjectOnScreen()
    {
        GameObject obj = ObjectWithBestProbrability();

        GameObject instanceOfObject = (GameObject)Instantiate(obj);

        if (this.playership != null)
        {
            float difference = positionOfCamera.z - this.playership.transform.position.z;

            Vector3 camToWorld = Camera.main.ScreenToWorldPoint(new Vector3(0, Camera.main.pixelHeight, difference));
            if (instanceOfObject.GetComponent<SpawnPoints>() != null)
            {
                camToWorld.x = instanceOfObject.GetComponent<SpawnPoints>().x1 * this.playership.transform.position.x;
                camToWorld = camToWorld + Vector3.up * instanceOfObject.GetComponent<SpawnPoints>().y1;
                instanceOfObject.transform.position = new Vector3(camToWorld.x, camToWorld.y, camToWorld.z);
            }

            instanceOfObject.transform.parent = transform;
        }
        ScheduleSpawn();
    }
}
```

Problems Solved

- Building an efficient game with minimal lag:

  To make sure that the game did not lag, I took the appropriate steps and worked with the *Unity Profiler* as much as I could. The profiler allowed me to check the performance of my game. After performing user acceptance testing and noticing that a couple of players still noticed a bit of lag, I modified the design of the user interface to ensure that objects that did not move, were statically batched (explained in the optimization section of this document). I also used *Object Pooling* so that I could add as many background objects to my scene as I wanted without effecting the games performance.

- Code Efficiency:

To ensure that my code was efficient, when I implemented my custom *SoundManager* component, I used lambda expressions using LINQ to iterate through the list of sounds.

As explained in blog 5, the *Update* function which is part of the *Unity API* runs for every frame. To solve this problem, I placed code that only needed to be run once in the *Start* function. This function allows code to be executed only on the frame when a class is enabled just before any *Update* function is called the first time.

- Learning the Physics Engine:

To be able to implement the behaviour that I wanted my enemies to have, I had to make sure that I understood parts of physics such as vector arithmetic, rotations, Euler angles, and Quaternions. I also had to learn how to use collision detection with the *Unity Engine*. To solve this learning barrier, I read some articles and watched some videos so that I could understand these theorems. For collision detection, I used the *Unity API Documentation*.

References:

Vectors

Euler Angles and Quaternions

Collision Detection

- Spawning Objects:

  Spawning objects randomly into the scene is straightforward, but spawning objects in a way that would make the game interesting from the players perspective was difficult.

  To solve this problem, I made sure that objects were spawned intelligently in my scene. I used weights to make sure that objects that should not be spawned frequently are not spawned (i.e. the mothership) and I spawned objects based on the players position. Doing it this way allowed me to add dynamic difficulty adjustment to the game based on the players performance.

- Keeping the difficulty level at intermediate:

  To keep the level of difficulty at an intermediate level, when I carried out user acceptance testing, I optimized the difficulty of the game based on the feedback that I got from each of the players.

- Time Management:

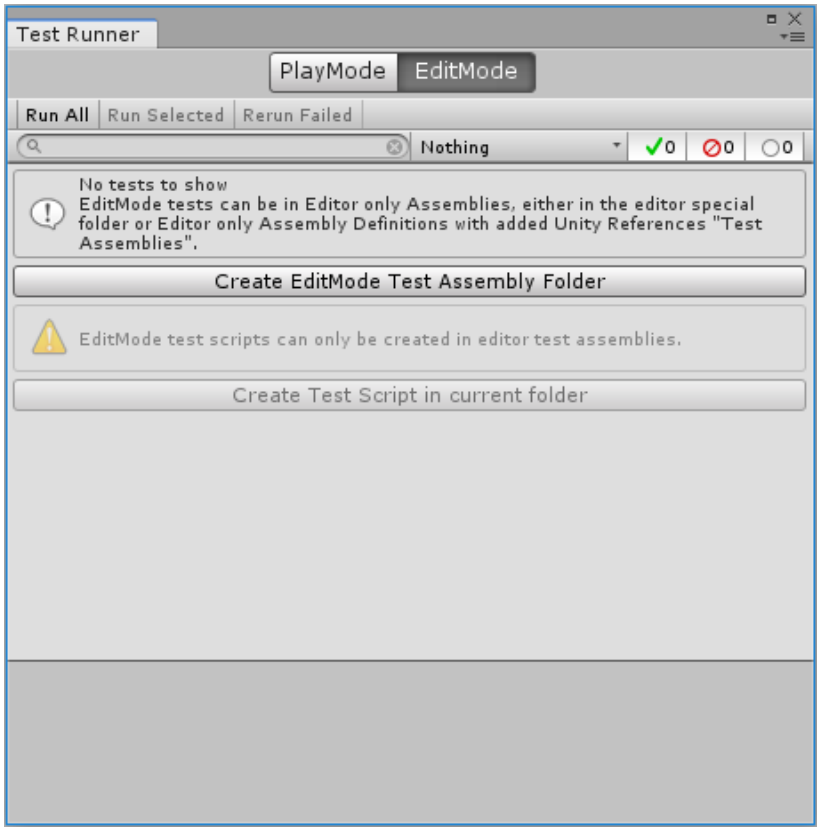
  To make sure that I was still performing well in other modules, I usually divided my days in the halves. I would do my project in the morning and then study the modules and complete assignments in the evening.

Testing

The most common type of testing in game development is Data Driven Development. Test Driven Development is good for testing the logic of your program, but in game design it loses value when we try to make things progress over time. This is because each time we move forward with the development of the game, we're constantly iterating, and the design is changing as the game evolves and progresses. Data Driven Development means that I can still have those tests and rely on those principles used in Test Driven Development but get actual data out of those tests to make my design and game better.

To perform unit and integration tests I used the built-in test runner that is provided by Unity. This tool can be used to test code in both ***Edit*** and ***Play*** mode. I used this test-runner to carry out my integration, unit, and UI tests for my game.

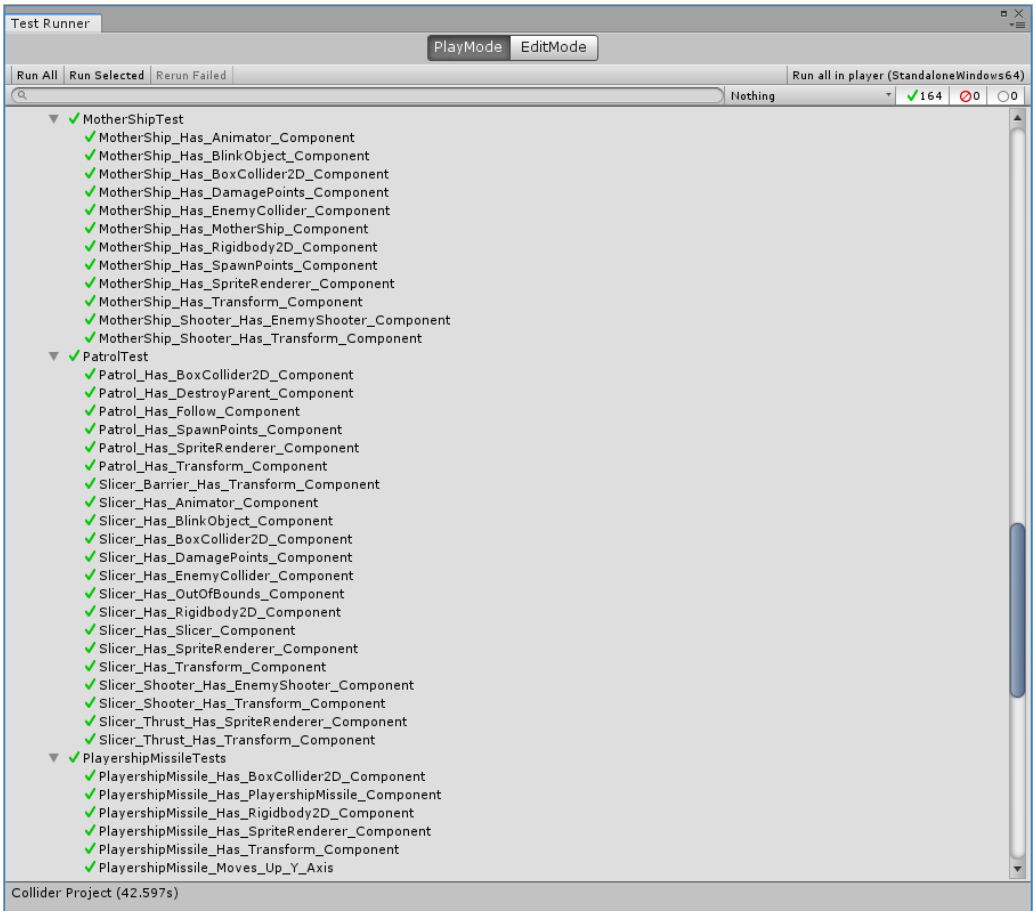
Reference: Data Driven Development [Skip to 7:00]

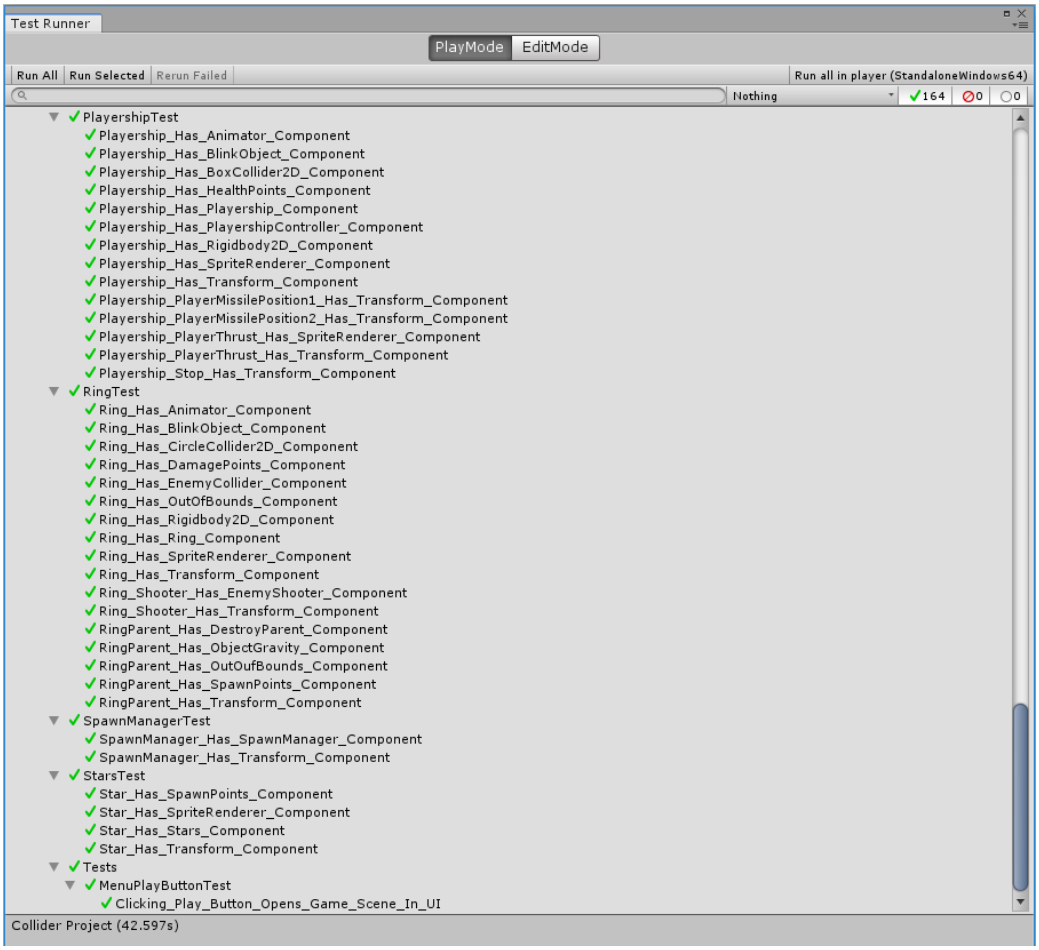


To perform integration and unit tests I used the open-source libraries NUnit and NSubstitute to execute my tests and to add assertions to them. I combined these two open source tools with a closed-source dependency injector called ***IUnityService***, which is not part of the Unity API. This can be found in the ***src/Code*** folder of my GitLab repo. To carry out UI testing I used Unity's Automation Testing Tool that is provided on the Asset store and an open-source library called ***Puppetry***.

Unit Testing

Part of Data Driven Development in Unity is to perform tests to validate the behaviour of the objects in a game when they are attached to their components. For unit tests, I wrote tests to validate that the correct components were attached to each game object at run time and that all the components attached perform as they are expected to. Below are some of the unit tests that I wrote.


Test Runner
PlayMode
EditMode
Run All
Run Selected
Rerun Failed
Run all in player (StandaloneWindows64)
Nothing
164
0
0
MotherShipTest
MotherShip_Has_Animator_Component
MotherShip_Has_BlinkObject_Component
MotherShip_Has_BoxCollider2D_Component
MotherShip_Has_DamagePoints_Component
MotherShip_Has_EnemyCollider_Component
MotherShip_Has_MotherShip_Component
MotherShip_Has_Rigidbody2D_Component
MotherShip_Has_SpawnPoints_Component
MotherShip_Has_SpriteRenderer_Component
MotherShip_Has_Transform_Component
MotherShip_Shooter_Has_EnemyShooter_Component
MotherShip_Shooter_Has_Transform_Component
PatrolTest
Patrol_Has_BoxCollider2D_Component
Patrol_Has_DestroyParent_Component
Patrol_Has_Follow_Component
Patrol_Has_SpawnPoints_Component
Patrol_Has_SpriteRenderer_Component
Patrol_Has_Transform_Component
Slicer_Barrier_Has_Transform_Component
Slicer_Has_Animator_Component
Slicer_Has_BlinkObject_Component
Slicer_Has_BoxCollider2D_Component
Slicer_Has_DamagePoints_Component
Slicer_Has_EnemyCollider_Component
Slicer_Has_OutOfBounds_Component
Slicer_Has_Rigidbody2D_Component
Slicer_Has_Slicer_Component
Slicer_Has_SpriteRenderer_Component
Slicer_Has_Transform_Component
Slicer_Shooter_Has_EnemyShooter_Component
Slicer_Shooter_Has_Transform_Component
Slicer_Thrust_Has_SpriteRenderer_Component
Slicer_Thrust_Has_Transform_Component
PlayershipMissileTests
PlayershipMissile_Has_BoxCollider2D_Component
PlayershipMissile_Has_PlayershipMissile_Component
PlayershipMissile_Has_Rigidbody2D_Component
PlayershipMissile_Has_SpriteRenderer_Component
PlayershipMissile_Has_Transform_Component
PlayershipMissile_Moves_Up_Y_Axis
Collider Project (42.597s)


Test Runner
PlayMode
EditMode
Run All
Run Selected
Rerun Failed
Run all in player (StandaloneWindows64)
Nothing
164
0
0
PlayershipTest
Playership_Has_Animator_Component
Playership_Has_BlinkObject_Component
Playership_Has_BoxCollider2D_Component
Playership_Has_HealthPoints_Component
Playership_Has_Playership_Component
Playership_Has_PlayershipController_Component
Playership_Has_Rigidbody2D_Component
Playership_Has_SpriteRenderer_Component
Playership_Has_Transform_Component
Playership_PlayerMissilePosition1_Has_Transform_Component
Playership_PlayerMissilePosition2_Has_Transform_Component
Playership_PlayerThrust_Has_SpriteRenderer_Component
Playership_PlayerThrust_Has_Transform_Component
Playership_Stop_Has_Transform_Component
RingTest
Ring_Has_Animator_Component
Ring_Has_BlinkObject_Component
Ring_Has_CircleCollider2D_Component
Ring_Has_DamagePoints_Component
Ring_Has_EnemyCollider_Component
Ring_Has_OutOfBounds_Component
Ring_Has_Rigidbody2D_Component
Ring_Has_Ring_Component
Ring_Has_SpriteRenderer_Component
Ring_Has_Transform_Component
Ring_Shooter_Has_EnemyShooter_Component
Ring_Shooter_Has_Transform_Component
RingParent_Has_DestroyParent_Component
RingParent_Has_ObjectGravity_Component
RingParent_Has_OutOufBounds_Component
RingParent_Has_SpawnPoints_Component
RingParent_Has_Transform_Component
SpawnManagerTest
SpawnManager_Has_SpawnManager_Component
SpawnManager_Has_Transform_Component
StarsTest
Star_Has_SpawnPoints_Component
Star_Has_SpriteRenderer_Component
Star_Has_Stars_Component
Star_Has_Transform_Component
Tests
MenuPlayButtonTest
Clicking_Play_Button_Opens_Game_Scene_In_UI
Collider Project (42.597s)

Integration Testing

For integration testing, I tested the behaviour of the game objects, i.e. I wrote tests to check the behaviour of the player and the enemies.

I began by writing tests to validate user input, i.e. to test that the player can move the player ship on both the x and y axis. To be able to write tests for the player input, I used the ***Humble Object Pattern***.

The humble object pattern is when we extract all the logic from the hard-to-test component into a component that is testable via synchronous tests. This component implements a service interface consisting of methods that expose all the logic of the untestable components with the only difference being that they are accessible via synchronous method calls. As a result, the ***Humble Object*** becomes a very thin adapter layer that contains very little code. Basically, this "Humble Object" will become a wrapper around your testable objects.

Reference: Humble Object Pattern

Why is this pattern used; the problem I am facing lies with Unity's static ***Input*** and ***Time*** classes which are used to get input from the user to move the player.

```
private void Update()
{
    FireMissiles();

    float xAxis = Input.GetAxis("Horizontal");
    float yAxis = Input.GetAxis("Vertical");

    Vector2 velocity = VectorComputation(xAxis, yAxis);
    MoveInDirection(velocity);
}
```

```
private Vector2 VectorComputation(float x, float y)
{
    Vector2 velocity = new Vector2(x, y).normalized;
    return velocity;
}
```

```
private void MoveInDirection(Vector2 velocity)
{
    this.playerPosition += velocity * this.playerSpeed * Time.deltaTime;
```

Static classes are the bane of automated tests because there is no way to control their output. I must deal with these static classes if I ever want my tests to pass. Player movement (the code in the above images) is implemented in my ***Playership*** class. To write tests for player input, I had to extract the logic from my *VectorComputation* and *MoveInDirection* functions into its own class called ***HumbleMovement***. In this class I then combined these two functions into one testable function called *ComputeMovement*.

```
public Vector2 ComputeMovement(float h, float v, float _deltaTime)
{
    return new Vector2(h, v).normalized * this.speed * _deltaTime;
}
```

Then I needed a way to assert that the players movement is indeed being affected by the users input because at this point I still had Unity's static ***Input*** and ***Time*** classes to deal with. I had to be able to override the values that they return so that I can have consistent and predictable tests. To do this I had to use ***Dependency Injection***.

Dependency Injection is a technique whereby one object supplies the dependencies of another object. A ***dependency*** is an object that can be used as a service. An ***injection*** is the passing of a dependency to a dependant object that will use it. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Reference: Dependency Injection

To implement dependency injection, I wrapped those static classes into one service class called *UnityService*, which is not part of the Unity API, and then implemented player movement in a new class, which I called *PlayershipController*. UnityService uses an interface called *IUnityService* to expose the methods and properties used to implement player movement.

```
public interface IUnityService
{
    float GetDeltaTime();
    float GetTime();
    float GetAxisRaw(string axisName);
    Vector3 GetForward();
}
```

Next, I made player movement depend on this service class instead. Then I injected the service into the player at runtime.

```
private void Update()
{
    this.playerPosition += humbleMovement.ComputeMovement(
        unityService.GetAxisRaw("Horizontal"),
        unityService.GetAxisRaw("Vertical"),
        unityService.GetDeltaTime());
}
```

To write the test I used ***NSubstitute*** to create a mock version of IUnityService. For example:

```
[UnityTest]
public IEnumerator Player_Can_Move_Right_On_Horizonal_Input()
{
    var player = playerShip.GetComponent<PlayershipController>();
    player.playerSpeed = 1;

    var unityService = Substitute.For<IUnityService>();
    unityService.GetAxisRaw("Horizontal").Returns(1);
    unityService.GetDeltaTime().Returns(1);
    player.unityService = unityService;

    yield return new WaitForSeconds(3);

    Assert.AreEqual(1, player.playerPosition.normalized.x, 0.1f);
}
```

Since the x-axis is normalized, meaning it has a maximum length of 1 (from -1 to 1). If the player moves right they should be at position 1 on the x axis (with -1 and 0 being left and middle respectively) meaning that the test should pass. I added 3 more tests to check if the player can also move left on horizontal input and up and down on vertical input from the user.

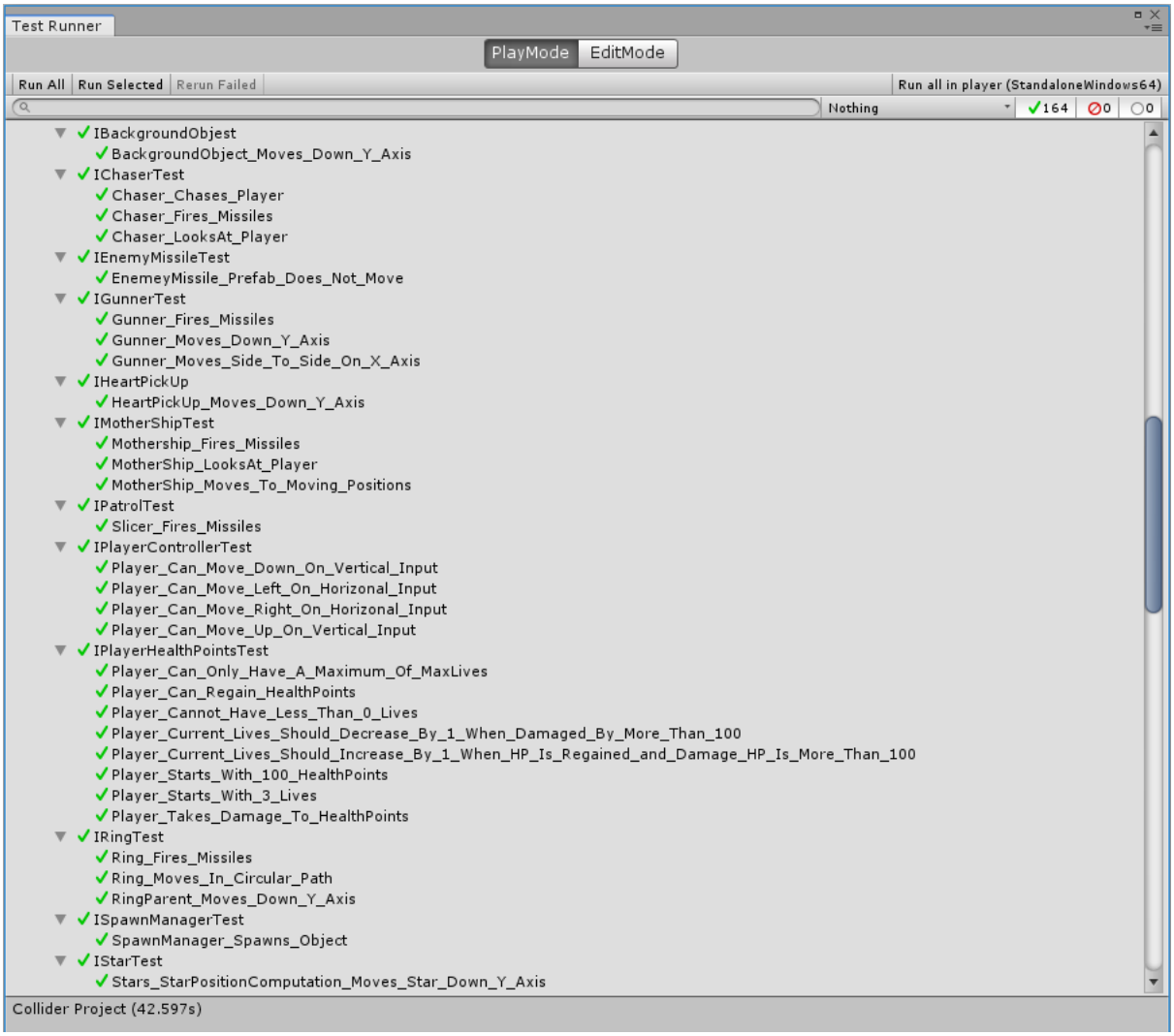
I also wrote tests for the players health, i.e. to check if the player starts off with the correct number of lives and health points and that the players lives and health points never become less than 0 (important test because if it ever did, then the player would never die).

For the enemies, I wrote tests to validate their behaviour, for example do they move or chase the player, and do they shoot missiles, etc. As aforementioned, these are the types of tests implemented in Data Driven Development.

```
[UnityTest]
public IEnumerator Chaser_Fires_Missiles()
{
    var shooterPosition = enemy.transform.GetChild(2).transform.position;

    yield return new WaitForSeconds(enemy.transform.GetChild(2).GetComponent<EnemyShooter>().startTime);

    var missilePosition = GameObject.FindGameObjectWithTag("EnemyMissileTag").transform.position;

    Assert.AreNotEqual(shooterPosition, missilePosition);
}
```

Above is an example to test if the *chaser* enemy ship fires a missile. Each enemy game object has a *shooter* object attached to them as a child object which is responsible for firing missiles. If a missile is fired, then the position of the missile should not be the same as the position of its *shooter*.

I also added unit and integration tests for my *SpawnManager* component.



User Interface Testing

To perform user interface testing I used Unity's Automation Testing tool that was provided in the asset store and an open-source library called **Puppetry**.

References:

[Unity UI Automation Testing Tool]

[Puppetry]

User interfacing testing is very important in game development. It involves assessing the UI based on important variables such as visibility, consistency, and usability. So, I wanted to have user interface tests for each of these important variables.

- Visibility:

  To carry out tests for this variable, I ensured that whenever a scene was loaded, its respective components were present and that components expected to be active in the hierarchy, were active (i.e. when the player opens the game scene, the in-game menu should not be active until they click the escape button).

```
/// <summary>
/// This Test is to ensure that when the Menu scene is loaded, that all the Menu components that are expected to be active
/// in the hierarchy are indeed active and components that are expected to be inactive are indeed inactive.
/// </summary>
[Test]
public void OpenScene_MenuSceneComponentsExpectedToBeActiveAreActive()
{
    GameObject mainCamera = new GameObject().FindByUPath("/Main Camera");
    GameObject canvas = new GameObject().FindByUPath("/Canvas");
    GameObject background = new GameObject().FindByUPath("/Canvas//Background");
    GameObject rawImage = new GameObject().FindByUPath("/Canvas//Background//RawImage");
    GameObject videoPlayer = new GameObject().FindByUPath("/Canvas//Background//Video Player");
    GameObject menu = new GameObject().FindByUPath("/Canvas//Background//Menu");
    GameObject ranking = new GameObject().FindByUPath("/Canvas//Background//Ranking");
    GameObject controlsMenu = new GameObject().FindByUPath("/Canvas//Background//ControlsMenu");

    mainCamera.Should(Be.Present);
    mainCamera.Should(Be.OnScreen);
    mainCamera.Should(Be.ActiveInHierarchy);

    canvas.Should(Be.Present);
    canvas.Should(Be.OnScreen);
    canvas.Should(Be.ActiveInHierarchy);
```

```
[Test]
public void OpenScene_GameSceneComponentsExpectedToBeActiveAreActive()
{
    GameObject mainCamera = new GameObject().FindByUPath("/Main Camera");
    GameObject spawnManager = new GameObject().FindByUPath("/SpawnManager");
    GameObject playerShipMove = new GameObject().FindByUPath("/PlayershipMove");
    GameObject background = new GameObject().FindByUPath("/Background");
    GameObject starCreatorMove = new GameObject().FindByUPath("/StarCreatorMove");
    GameObject canvas = new GameObject().FindByUPath("/Canvas");
    GameObject eventSystem = new GameObject().FindByUPath("/EventSystem");
    GameObject soundManager = new GameObject().FindByUPath("/SoundManager");
    GameObject gameManager = new GameObject().FindByUPath("/GameManager");
    GameObject movingPosition = new GameObject().FindByUPath("/MovingPosition");

    mainCamera.Should(Be.Present);
    mainCamera.Should(Be.OnScreen);
    mainCamera.Should(Be.ActiveInHierarchy);

    spawnManager.Should(Be.Present);
    spawnManager.Should(Be.OnScreen);
    spawnManager.Should(Be.ActiveInHierarchy);
```

- Consistency:

  For this variable, I wrote tests to ensure that all the buttons that were present in each scene were interactive and display the correct text.

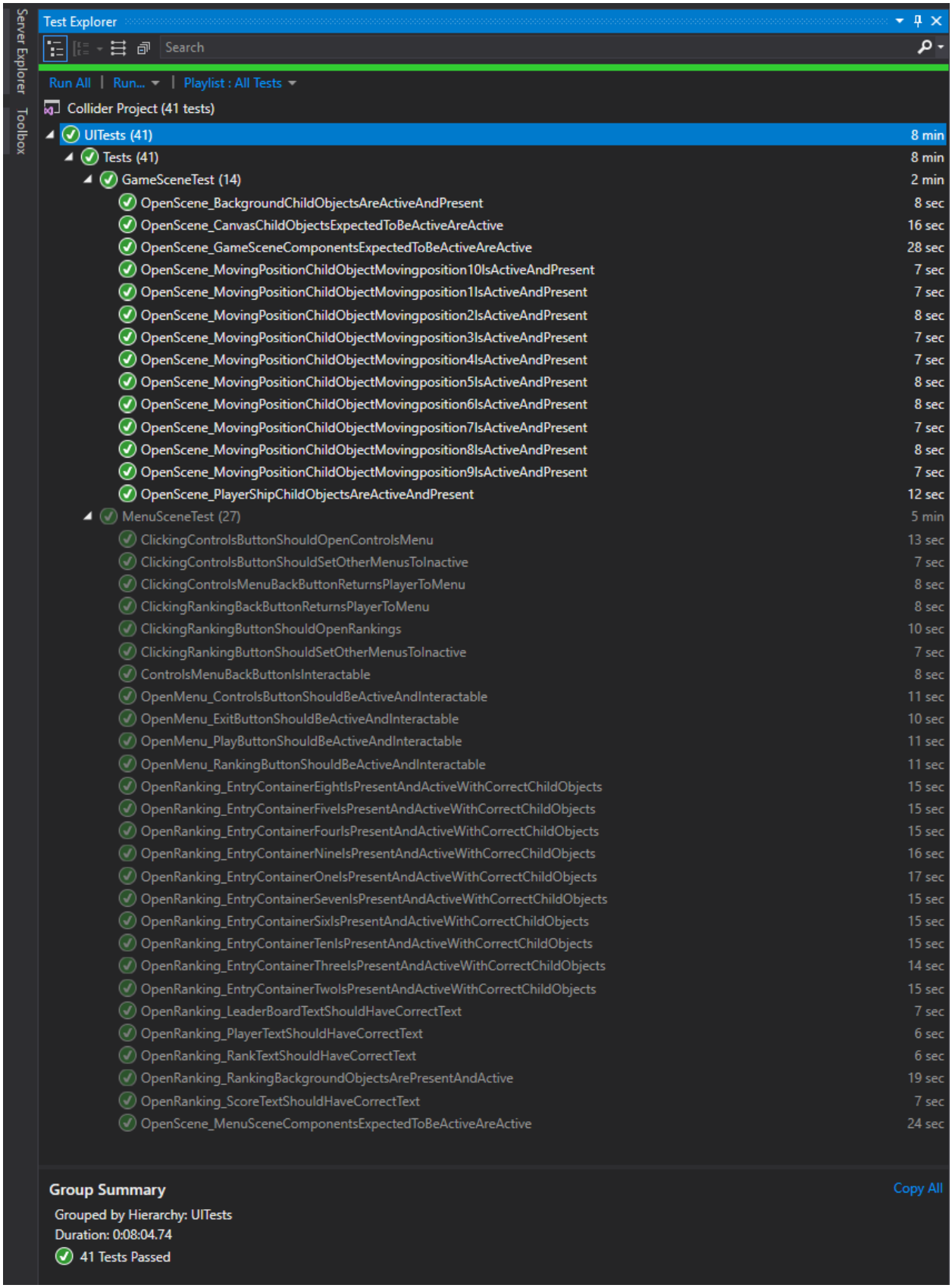
```
/// <summary>
/// The tests below ensure that the menu buttons are active in the Hierarchy
/// and that each button is interacactable and displays the correct text.
/// </summary>
[Test]
public void OpenMenu_PlayButtonShouldBeActiveAndInteractable()
{
    GameObject playButton = new GameObject().FindByUPath("/Canvas//Background//Menu//play_button");
    GameObject playButtonText = new GameObject().FindByUPath("/Canvas//Background//Menu//play_button//play_text");

    playButton.Should(Be.Present);
    playButton.Should(Be.OnScreen);
    playButton.Should(Have.ComponentWithPropertyAndValue("Button", "m_Interactable", "true"));
    playButton.Should(Be.ActiveInHierarchy);

    playButtonText.Should(Be.Present);
    playButtonText.Should(Be.OnScreen);
    playButtonText.Should(Have.Component("TextMeshProUGUI"));
    string text = playButtonText.GetComponent("TextMeshProUGUI");
    Assert.IsTrue(text.Contains("PLAY"));
}
```

- Usability:

  Finally, I wrote tests to ensure that when buttons were clicked, they performed the right actions. For example, when the player clicks on the controls button in the main menu, the controls sub-menu should be displayed with it its component and child components active in the hierarchy and with the main menu and other sub-menu components set to inactive in the hierarchy.

```
[Test]
public void ClickingControlsButtonShouldSetOtherMenusToInactive()
{
    GameObject controlsButton = new GameObject().FindByUPath("/Canvas//Background//Menu//controls_button");
    controlsButton.Click();

    GameObject menu = new GameObject().FindByUPath("/Canvas//Background//Menu");
    menu.ShouldNot(Be.ActiveInHierarchy);

    GameObject ranking = new GameObject().FindByUPath("/Canvas//Background//Ranking");
    ranking.ShouldNot(Be.ActiveInHierarchy);
}

[Test]
public void ClickingControlsButtonShouldOpenControlsMenu()
{
    GameObject controlsButton = new GameObject().FindByUPath("/Canvas//Background//Menu//controls_button");
    controlsButton.Click();

    GameObject controlsMenu = new GameObject().FindByUPath("/Canvas//Background//ControlsMenu");
    GameObject controlsMenuBackButton = new GameObject().FindByUPath("/Canvas//Background//ControlsMenu//back_button");
    GameObject controlsMenuKeyboardImage = new GameObject().FindByUPath("/Canvas//Background//ControlsMenu//Keyboard");
    GameObject controlsMenuMouseImage = new GameObject().FindByUPath("/Canvas//Background//ControlsMenu//Mouse");

    controlsMenu.Should(Be.ActiveInHierarchy);

    controlsMenuBackButton.Should(Be.Present);
    controlsMenuBackButton.Should(Be.OnScreen);
    controlsMenuBackButton.Should(Be.ActiveInHierarchy);
```

Below are all the user interface tests that I wrote:

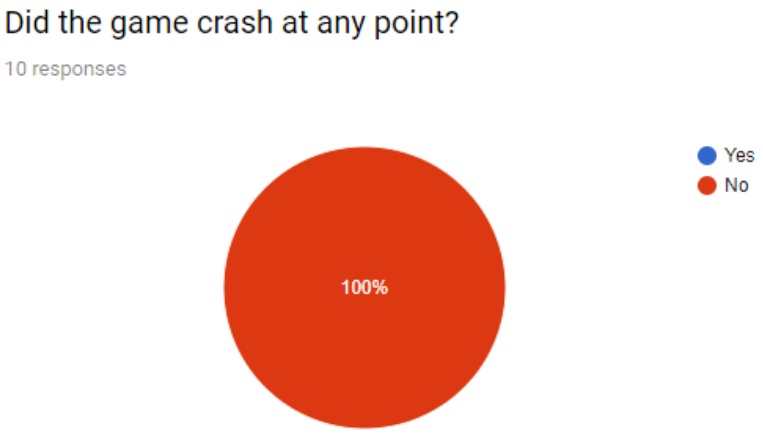


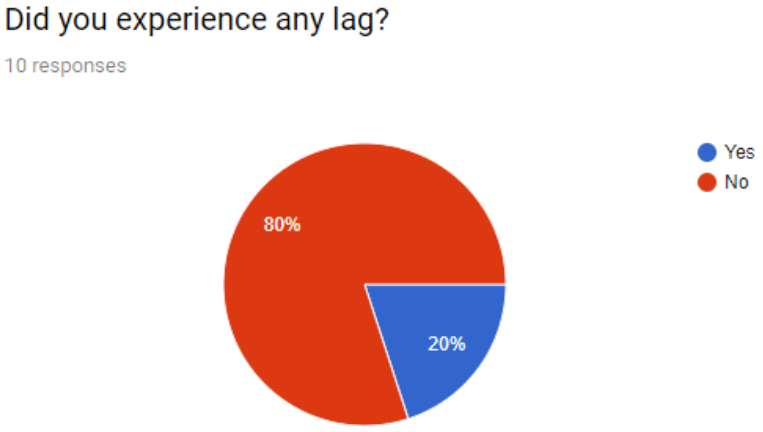
User Acceptance Testing

The last method of testing that I carried out was user acceptance testing. To carry out user acceptance testing I allowed people to play my game and to fill in a questionnaire after they played. For user testing, my main focuses were:

1. Firstly, to make sure the players were comfortable with the user interface.

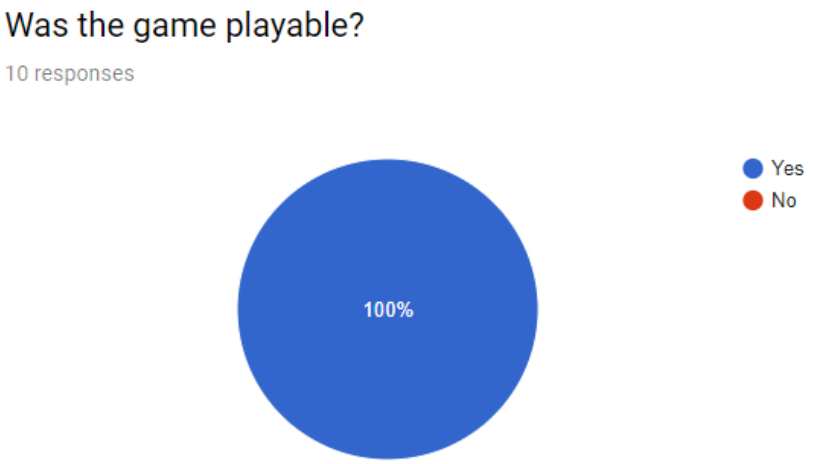
2. Secondly, to ensure that the system was usable, the game did not crash at any point, and that the player did not experience any lag.

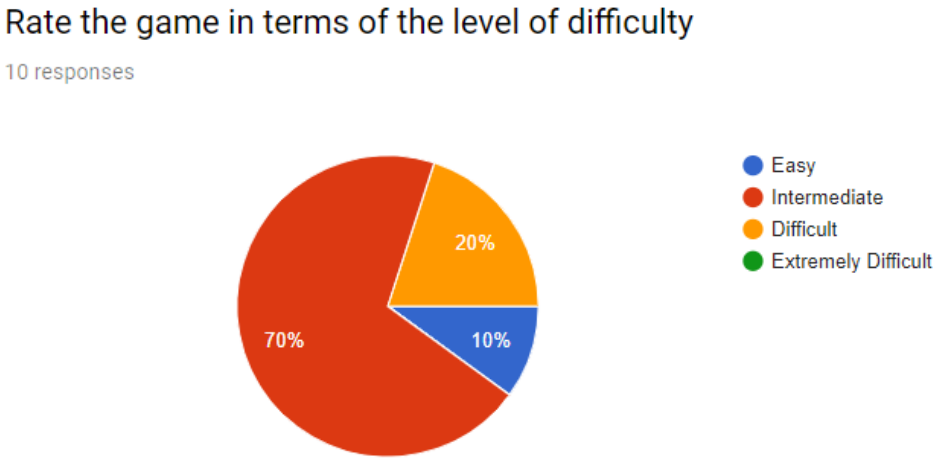




As you can above, a couple of people experienced lag. I took the necessary steps to reduce the lag, which I will discuss in my optimization section of this document.

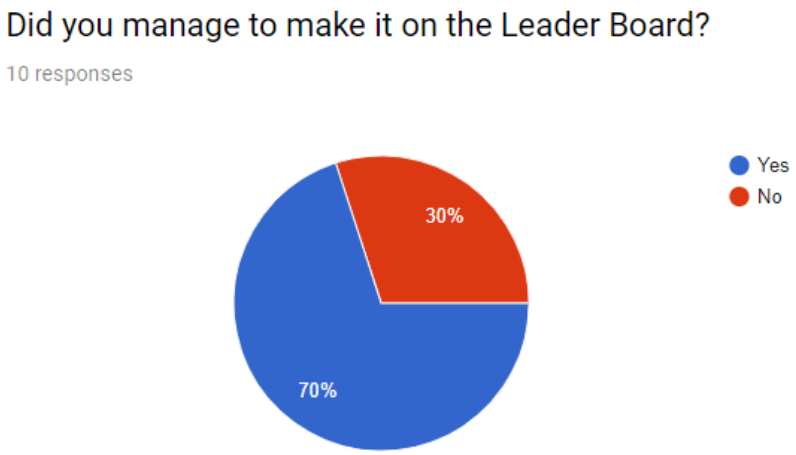
3. Thirdly, to make sure that the game was playable in terms of the difficulty level. I wanted to keep the difficulty level at intermediate, so I asked the players to rate the game in terms of difficulty.

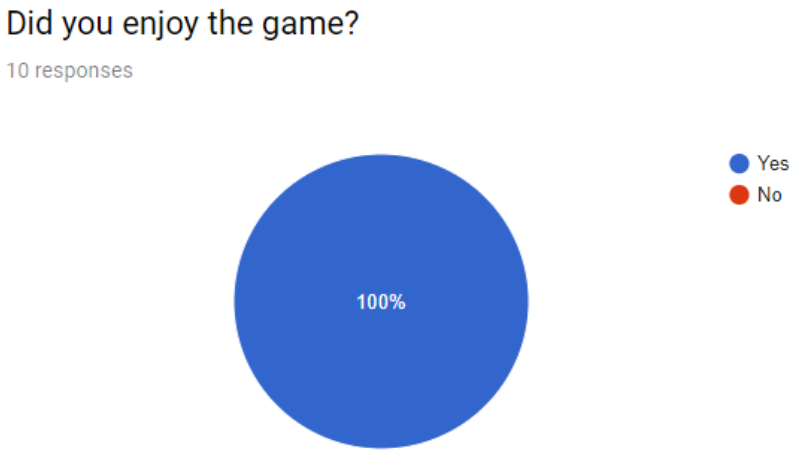




I was satisfied with the results and was happy to see that 70% of the players felt like the game was an intermediate level of difficulty and that all players thought the game was playable.

4. Finally, to see if the users enjoyed the game and made it onto the leader board. As you can see only 30% of the players didn't make it. This was because I set the refresh time in the database too high (to about 15 seconds), which caused some players to think their scores were not posted (basically, if you left the rankings sub-menu before 15 seconds you didn't see your score). So, to fix this I lowered the refresh rate.

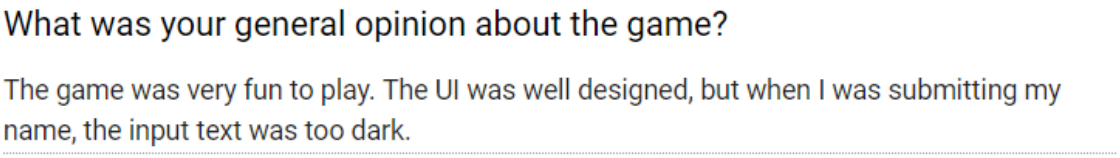




Optimization

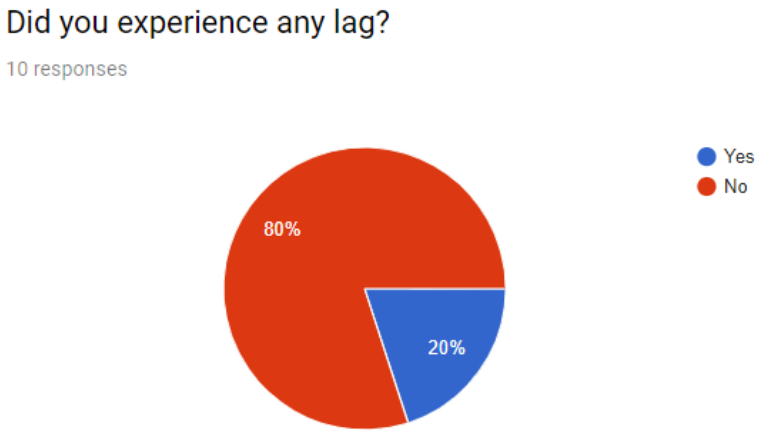
After carrying out user acceptance testing, I made some changes based on the feedback that I got from the players. There was an optional question at the end of my questionnaire that allowed players to give a general opinion on the game if they wanted too.

- The input text when players were submitting their name was too dark:



To address this issue, I just made a simple change to the UI to make input text brighter.

- Some players experienced lag:



- Lag – is a time delay between a player's action and the game's reaction to that input.

So far, I have one way that reduces lag in the game already, which is ***Object Pooling***. This is discussed in [blog 7]. To reduce lag even more, I will be using a technique known as ***Static Batching***, which is done in the Unity UI:

- Static Batching – allows the engine to reduce draw calls for geometry of any size provided that it shares the same material and does not move.

Finally, to reduce lag even more, I implemented player reloading. Rather than letting the player fire missiles whenever they like, the player will be given five missiles. When the player runs out of missiles, the missiles will then be reloaded, and the player will have to wait a few microseconds before they can start to fire again. This works because reducing the number of missiles the player can fire ultimately reduces the number of draw calls being made to the CPU per frame.

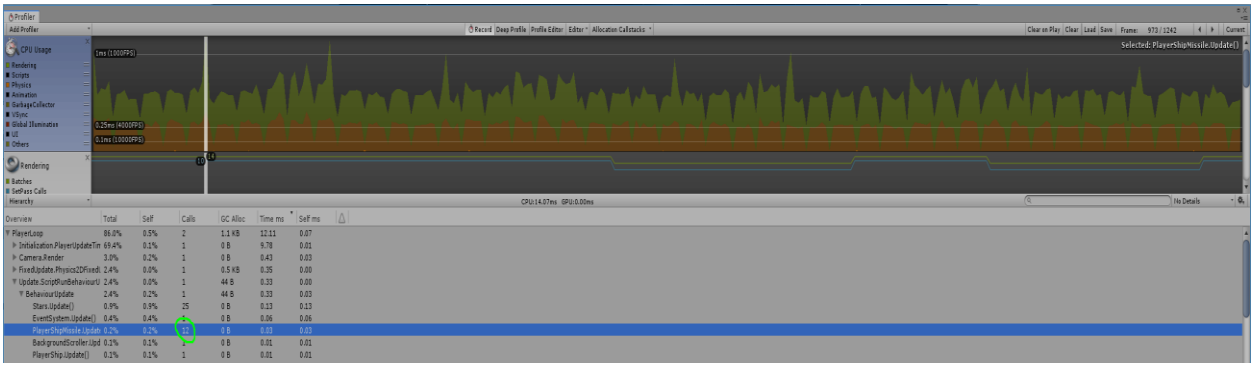
- Draw Call – contains all the information telling the GPU about textures and rendering objects etc. encapsulated as CPU work that prepares drawing resources for the graphics card.

Reference: [Draw Calls]

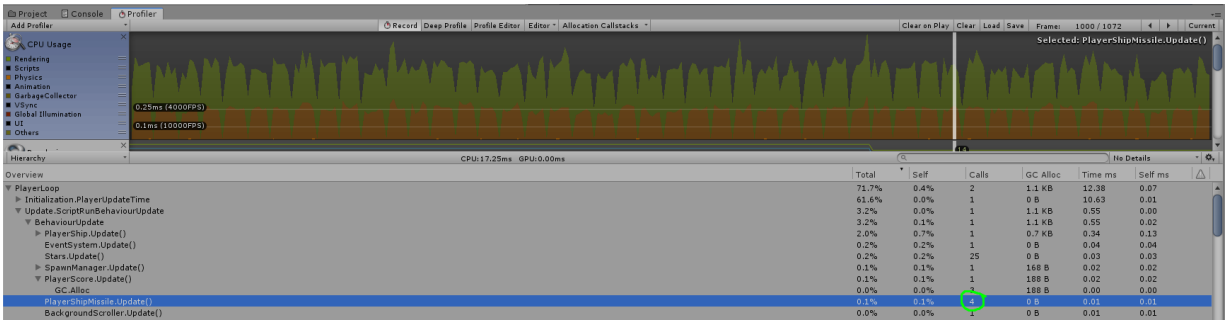
To demonstrate the effects of this, below you can see that the number of draw calls made by the PlayerShipMissile component has been reduced. To look at the performance of my game, I used the ***Unity Profiler***.

- Unity Profiler – gives detailed information about how your game is performing.

Draw calls of player missile before optimization:



Draw calls of player missile after optimization:



As shown above in the second image, I have dropped the number of draw calls made by the CPU to **4** per frame, it used to be **12**. This will reduce the amount of lag significantly.

Results

Overall, I was satisfied with the way my project turned out. I enjoyed all aspects of the project; from designing, thinking of ideas, changing my ideas, and interacting with the users during user acceptance testing to see how I could improve the game. I was also happy to see that players enjoyed the game and that majority of the people felt like it was playable in terms of the level of difficulty. I also feel like I learned about the *Unity Engine* and the physics that I used in my game. As a person who plays games very often, I thought it was very interesting to look at some of the techniques that game developers us to create enjoyable games with high graphics, while still keeping performance at an optimum level.

Future Work

In regard to the future, I plan to make more games and to add more features to this one in order to build up my skillset. As aforementioned, I am also very interested in the profitable side of game development, so I will definitely be looking into monetizing some of the games that I make.

I also really enjoyed the testing phase of my project, so I also plan on going into jobs related to software testing.