TUMBLE LADS: DEVELOPING AN OBSTACLE COURSE GAME WITH NEWTONIAN PHYSICS USING UNITY GAME ENGINE

A dissertation of a development of an obstacle course game

JULY 11, 2022

Student Name – Evan Smith
Student Number – C2108326
Supervisor – Dr Frank Langbein

Acknowledgments

I'm extremely grateful to my supervisor, Dr Frank C. Langbein for his patience and feedback throughout the duration of this project. Lastly, I would also thank my family, especially my parents, brother and sister. They have supported me throughout and kept my spirits and motivation as well as emotional support.

Abstract

The classic problem of collision detection involves the intersection between two objects. This project aims to produce a 2-D prototype that simulates Newtonian physics as well as the inclusion of a user interface within a game engine. The quantitative research focuses on both spatial partitioning and bounding volumes that can be used to address the problem, highlighting both the advantages and disadvantages. Furthermore, a comparison of Unity and Unreal Engine was conducted, with the key findings from the research determining that Quadtree was the best method in spatial partitioning with Unity being the suitable game engine for development. These findings indicate the need for systems to produce collision detection systems in games that contain multiple collisions. These are measured in performance such as CPU, memory, and physics 2D when collisions occur. The results show the prototype is playable however further improvement is necessary for a detailed collision detection that measures its precision and accuracy.

Contents

Table of Figures

Chapter 1 – Introduction

The use of the implementation of physics systems within game engines is a crucial feature in games development. The ability to simulate aspects of Newtonian physics such as collision detection, gravity and velocity in game time are vital to any game. The popularity of Fall Guys, an obstacle course game inspired by Total Wipeout and Takeshi's Castle, arose during the time of the Covid-19 Pandemic. The aim of the project is to develop a game with 2-D graphics as well as dynamic physics simulations within a game engine of choice. The objectives include the development of the game in a 2-D environment as well as generating, storing, managing, and rendering them. Secondly, the prototype must include implementation of Newtonian physics in game context. Thirdly, a literature review into the state of the art of physics and game engines is produced.

During the writing of this dissertation, a literature review is conducted into the state-of-the-art surrounding collision detection that contains two categories, spatial partitioning, and bounding volumes. Unity and Unreal Engine, two game engines commonly used in today's game industry, are compared based on the features that are required for this project, how each of the game engines functions for each feature and selecting the appropriate option based on the findings. Other factors will also be considered, such as constraints relating to this project.

A prototype of the game is developed with the game engine chosen during the literature review, detailing the components of the physics system within the game engine, the code written to implement the features and the tests undertaken to determine the success of the implementation. Finally, the specification is produced, detailing the problem further with a requirement to address it. An evaluation is conducted to determine the functionality of the prototype and the necessity for future developments will be determined based on the improvements required, as well as extensions. The expectations of the findings have been met where the user interface and level have been developed. The physics aspect has been met however the prospect of a grid-based detection system failed to materialize. The level is

developed and playable that incorporates both physics and UI with further optimization. Performance in the aspect of both CPU and memory remained stable only to increase when collision events arise. Lastly, future learning is determined in relation to the insight and knowledge gained during the development of the project.

Chapter 2 – Literature Review

In this section, the literature review determines the state of the art of the field of Newtonian physics that includes collision detection as well as the subfields of octree and quadtree. In addition to the review of the physics aspect in relation to games development, a comparison of both Unity and Unreal Engine will also be examined in terms of their features, programming languages and ease of accessibility. One of these will be chosen with justification provided in the interest of this project.

2.1 Game Engine Requirements

The issues regarding this project are determined by a number of factors, including, rendering and computational intersection. The development of the game involves a substantial amount of object intersection during runtime; therefore, a physics system is required. There are multiple approaches in terms of collision detection, such as spatial portioning, hit boxes in a 2-D environment and bounding volumes. Without a physics system, objects will phase through each other, meaning it will carry on. As the game is an obstacle course game, a collision detection feature is required.

Secondly, the game scene and every update that occurs must be rendered. This means that rendering times in creating objects during gameplay must be rapid. With the previously mentioned problem relating to collision detection, it is important to render the objects once a collision has occurred. In the context of an obstacle course, there could be multiple collisions between different objects, therefore all objects must be rendered without lags. With the problem defined in relation to the project, there are works that are related to this project that solved the issue posed.

2.3 Related Work

Fall Guys: Ultimate Knockout, developed by Mediatonic was released in 2020 on PC, Xbox, and PlayStation consoles. It is a 3D multiplayer battle royale game, in which the player must navigate through a series of multiple objects and mini games to win. At the beginning, 60 players will start and at the end of each round several players who fail to cross the finish line before a specific cut off or those who came last, will be eliminated. The final round will determine the winner whereby one player has to capture a crown at the end of the course, upon obtaining the crown, the player will win the game. The game was inspired by the Japanese game show “Takeshis’s Castle”

Fall Guys was developed using Unity, an open-source game engine that does have a professional version for studios with features that are not accessible in the individual version. Unity uses rigid-body dynamics which is a core component of the physics system within the game engine that allows objects to collide with others instead of phasing through them. When it comes to the collision detection, ragdoll physics are often used that includes knock back that occurs when an object moves at a rapid pace, collides with the player and sends them back.

The main justification of mentioning fall guys is due to the similar nature of this project. Both projects share same ideas and concepts, however, in connection to the problem, Fall Guys utilize the same collision detections as what is expected of this project given how the gameplay runs. With Takeshi’s castle also mentioned, it also bears relevance due to the popularity of obstacle course until 1990 when it ended. Mediatonic took the inspiration of the show in order to release Fall Guys. The connection between the two examples is evident.

2.4 Collision Detection

Physics systems within game engines are tasked to simulate rigid-body components of a game object within the game world. In games development, they are simulating real world elements of physics such as gravity and friction. Collision detection is often described by Montaut [1] et al as a "computational geometry problem", the issue of collision detection is unavoidable; however, the choice remains as to the best algorithm to check for an object intersection. There are two categories of algorithms that are selected, spatial partitioning and bounding boxes. Each of these will be examined as well as the algorithms that are commonly used that apply to both 2D and 3D objects. Pichlmair and Johansen [2] states that "In the case of a 2-D game, collision shapes are usually either circles, triangles, or rectangles. In 3-D games, they are often spheres, boxes, or capsules". Each environment has its own collision shapes within the game engine. However, there are other alternatives to accomplish the goal of collision detection. In this section, two categories of collision detection algorithms are examined.

2.4.1 Spatial Partitioning

Spatial partitioning is procedure which involves a space being divided into two or more subsets. There are numerous algorithms that can be implemented such as K-d trees, R-trees, quadtrees and octrees are used to divide the space. The latter two are the main focus in this category.

2.4.1.1 Quadtrees

Quadtree is a spatial data structure that consists of a node having four children in a two-dimensional environment. Originally proposed by Finkel and Bently in 1974, the tree was originally designed with map analogy as the nodes were labelled "NE, NW, SW, SE". The algorithm functions as traversing through nodes within the tree including subnodes from the current node. The concept of the algorithm is similar to an octree. There are many different types of the algorithm, however the two most common are region based and point based. Region based is a representation of a quadtree that focuses on a collection of blocks of a region. As Samet elaborates, this type of quadtree treats the region as "a union of maximal square blocks (or blocks of any desired shape) that may possibly overlap". This means that although the quadtree is a data structure that utilizes blocks in a partition, the region quadtree is a collection of blocks that are disjointed. Another representation of a

quadtree is known as a point quadtree that involves dividing the region into sub-regions based on the arbitrary point that is classed as the root of the tree. Quadtrees are often used in collision detection, spatial indexing and computer graphics.

There are three types of nodes within the algorithm, the first of which is the point node that is used to represent a point within the box of a 2-D space. Secondly, there is an empty node, this serves as a leaf node to indicate that there is no point within the region that it currently represents. Thirdly, the region node, which often represents a region that can contain four child nodes that can either be a point node or an empty node. The algorithm has two phases - insertion and search. The insertion function consists of recursively determining the best possible child node to store a point within the data structure. There is a condition within this function, if the child node is empty then it will be replaced with a point node that represents the point. When that happens, the insertion is concluded. If the child node is a point node, it is replaced with a region point and it is set as the current node when it is represented as a region node. The search aspect is a Boolean function that focuses on checking if the point exists in the 2-D space. Finding the best child node, a binary condition is presented, if the child node is empty, the value is 0 or false whereas if the child node is a point node, the value is 1 or true. Throughout the process, the tree will deepen, populated with nodes with records stored.

One of the main advantages of using quadtrees is the efficiency of region searching. Kahlon [3] states that the algorithm is very efficient as it can “sparse through the maps very easily and quickly compared to other methods”. This means that the performance of searching through the region and nodes, and querying information is useful for aspects of an application that focuses on spatial data. The main significant disadvantage is the inability to delete nodes during runtime. Finkel and Bentley [4] states that “Very difficult to perform deletions from Quad Trees”, meaning that nodes that are not required remains within the data structure. Also the algorithm cannot merge or reinsert nodes, meaning altering nodes is difficult. In terms of storage, this will require space if the tree deepens with the nodes that can not be deleted remained. The second disadvantage involves pictures. When quadtrees are applied to pictures and a comparison takes place that only involves rotations, it becomes difficult. This is further elaborated by Kahlon [3] where it relates to the “Quadtree depiction of such pictures will be so distinct”.

2.4.1.2 Octree

An octree is described as a three-dimensional binary tree structure within which each octant contains eight children. It is viewed as an extension of quadtrees due to the concept being similar and containing extra branches. Octrees are often used in 3-D games due to the extra dimension. However, it is also used in other applications such as collision detection simulation and rendering. Octree encoding was pioneered by Donald Meagher at Rensselaer Polytechnic Institute in 1980. In the aspect of how each node is represented, Meagher [5] states that “each node represents a region of the universe and has one or more values which defines the region”.

The octree divides a 3D space into 2 X 2 X 2 subspaces where *n* represents the depth of the octree. An octree stores data within nodes or 'leaf' that is recursively generated throughout the process. The first node is known as the root node that represents the entire 3-D object and it generates eight children. The tree is traversed through the nodes as well as generating nodes. There is a condition here, as if it completely describes the region, it is often concluded as a terminal node or a leaf and no more sub regions would be created as a result. Should however the latter occur whereby it does not describe the region, more octants of the current node are created and the process loops until its termination. Its structure is described as simplistic, according to Koh, Jayaraman and Zheng [6], "Due to the regular structure of the octree and its relative simplicity in implementation, it is a popular acceleration structure used in many applications". This means that it is so simple to implement and apply to 3-D objects that it is used in other applications.

There are advantages and disadvantages of applying octrees. One of the advantages that was cited by Meagher, is related to the calculations throughout the recursive loop. Meagher [7] states that partial calculations calculated "are passed to the lower level. Substantial reductions in computation can result." This is due to the recursive element of the loop, to prevent performance from being affected by reducing calculations. The second advantage of applying octrees involves *how* objects are represented within the data structure as Meagher [7] further states "An arbitrary object can be represented to the precision of the smallest cube". The cube is a primitive shape, meaning it is simple, therefore any object that uses this data structure can be simplified. If, however there are more complex shapes, new methods or techniques are not necessarily due to requiring only one set of manipulation and analysis algorithms. The final advantage is related to the hierarchical structure of the data tree. Due to the root node being represented as the entire object, Meagher [7] states that the "Nodes at a level together with the higher nodes completely describe the entire object to the resolution of that level", meaning that data stored at the lower levels can be avoided.

There is a significant disadvantage when using octrees, whereby memory is required due to the processing that takes place. However, the amount required is vast due to the amount of data to store. Wang [8] et al states that “The recursively generating and querying operation makes it very time-consuming”, implying that generating new levels of nodes and query information takes time. When querying information, this depends on the size of the leaf, if it is massive, it becomes time consuming, as Wang stated. However, if the size is smaller, more nodes are generated. The octree will keep expanding deeper, which poses an issue for the memory as it cannot store huge volumes of data. An explanation regarding how the data can become massive is that due to the object and the resolution, the more complex the object and high resolution results in an increase in data. Due to this issue, the algorithm should terminate based on how deep the octree becomes.

2.4.1.3 KD-Trees

KD- Trees was introduced by Bentley [9] in 1975 to address the problem of retrieving data within “a file F which contains a collection of records”. A multi-dimensional binary tree structure, Kd-Trees in practise, is devised by starting at the root node where two children are then created. Traversing the tree overtime, deepens it, much like octrees and quadtrees. With the most type of query being region based, it shares the same principle as quadtrees. In the aspect of storing data, Zhou and Wen [10] states that the data structure is designed for “Storing finite element point sets in k-dimensional space”. The term k-dimensional refers to a numbered dimension that the algorithm is applied to, for example, 2-D dimensional space. In terms of its advantages, the algorithm performs well in terms of efficiency in regard to storage. Another advantage described by Bentley [9] is that the algorithm is “Flexible enough to allow any intersection query. There is no restriction on what the query is when retrieving data. However, there is an issue with the algorithm, removing root nodes at the cost of memory. Deleting root nodes from the tree is possible, however, to do so, is expensive on memory. In contrast to the nature of this project, implementing K-d Trees in a games context requires memory and to delete root nodes throughout the process will have a negative impact on performance.

2.4.1.3 Binary Space Partioning

Binary Space Partioning (BSP) is a hierarchical structure tree that is designed to partition the scene in two, creating two children to store spatial data as well as traversing around the sub-regions and incrementing nodes. Proposed in 1969, Schumacher [11] et al, stated that due to the advances in both computers and circuits, “higher-quality images can be generated”. The study explores the approach to build an image generator in the context of 3-Dimensional computer graphics. In the context of game development, BSP was implemented primarily in first person shooters, one such example is Doom that was released in 1993 where the engine (id tech 1) uses the algorithm in the aspect of using spatial data for a level. In a book written by Sanglard, it explores the theory and practise of the algorithm in Doom, with Sanglard [12] stating that building the tree from the map “is to repeatedly select a line to split the map in two”, thus becoming a recursive process of all subsectors are convex.

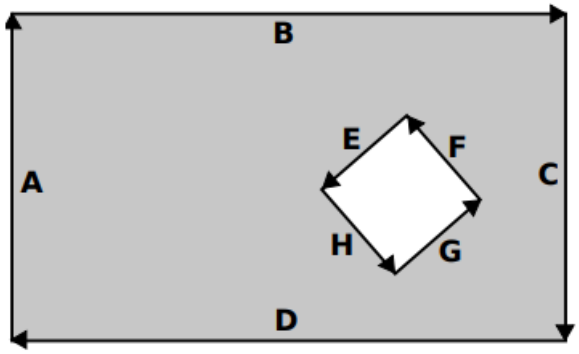

Figure 1 Application of BSP in Doom

One of the advantages relates to the performance of the algorithm. Su [13] et al states that “many algorithms with BSP tree exhibit better performance than those without BSP tree”, meaning that any algorithm that utilizes the data structure will perform better than algorithms that do not use it. One of the disadvantages of using the algorithm is due to the depth of the tree. Much like the previous algorithms discussed, over the time, the tree will grow, with more nodes that contains data gathered. Should the tree grow too big, storage and memory are affected. In the games context, this is not beneficial and will present issues as there will be many object intersections to simulate. The second disadvantage relates to the complexity of the structure. There are some cases where the tree’s structure can become difficult to implement, specifically in this project where the game is always updating and it is a 2D platform obstacle course, it may become complex.

Traditional octrees are a useful spatial data tree that can be used in 3-D geometry within game engines. The advantages of this relate to its simplistic hierarchal structure as well as reduced calculations upon the lower levels, but the issue with its traditional approach remains one of the storage of data depending on the complexity of the object as well its resolution. Another type of octree, a linear tree, was later proposed to address the issues. Per Wang [8] et al, stated that nodes within linear octrees are “generated fast and it doesn’t need to change tree greatly when a certain node divides into more small sub-cubes”. The use of linear octrees improves upon the traditional approach by fast node generation times as well as how querying is also shortened. Another improvement is how It stores data only in leaf nodes; therefore data is reduced overall. Both quadtrees and octrees are very useful data structures, however, in the context of games development, Kd-Trees would be unsuitable due to the amount of memory that the algorithm can cost, particularly with deleting root nodes while the game is constantly updating leading to negative impact on performance. BSP trees are very useful in rendering particularly in first person shooters such as Doom with the way it handles spatial data in levels however, it is not beneficial to use them in other aspects of game development such as game development due to the complexity of the tree.

2.4.3 Bounding Volumes

A bounding box is where an object is contained within a bounding volume that detects an intersection with another object. The use of bounding volumes gives the advantage of accelerating collision queries during runtime. There are many different types of bounding volumes that are used in the collision detection scenario, with each type specific to a different object. The three most common types are bounding spheres, axis-aligned bounding box (AABB) and the oriented bounding box.

2.4.3.1 Bounding Sphere

The bounding sphere is the simplest type as Melero, Aguilera and Feito [14] states, it is "very simple to compute and straightforward to determine whether two spheres collide". Bounding spheres can be applied to a spherical object such as a ball that can bounce off between walls. The detection of the collision refers to the equation utilizing the distance between the centre of the radius and the sphere of the object. Despite the simplicity of the structure, the detection accuracy is poor due to the size of the object, that being the smallest sphere.

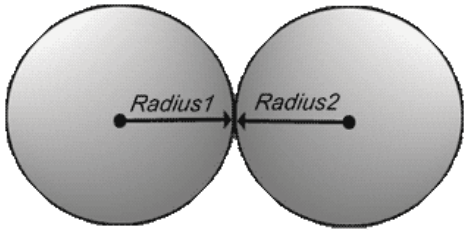

Figure 2 Bounding Sphere Intersection

2.4.3.2 Axis-Aligned Bounding Box

The AABB is another simple bounding volume, a 2-D rectangle, the structure of the volume is simple however, according Gan and Dong [15], in the interest of detection accuracy, it "is higher than that of sphere". This means that the accuracy of detecting an intersection between two objects is significantly higher than using a bounding sphere, due to the size of the object.

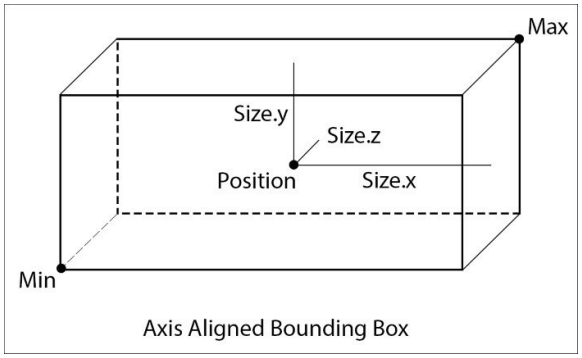

Figure 3: AABB Diagram

The detection equation utilizes the minimum and maximum of X,Y,Z where they are the coordinates of the centre of the box.

$$R\ area = \{x, y\} |\ minx\ \leq x\ \leq maxx, miny\ \leq y\ \leq maxy\}$$

Figure 4: Equation of calculating the area of AABB

The only disadvantage of using this type of bounding volume is that the angle cannot be changed rather it has to be recalculated, so this is not suitable for collision detections where the object's orientation is different.

2.4.3.3 Oriented Bounding Box

Finally the oriented bounding box (OBB), a 2-D rectangle that was introduced as an improvement over the AABB with the major difference according to Chaoyang and Fenli [16] being one of "The direction of arbitrariness" meaning that the rotation of the OBB is different so it can cover the surface of any cube object in any direction. This results a tighter fit volume than that of the AABB and again, a higher detection accuracy than both AABB and the bounding sphere.

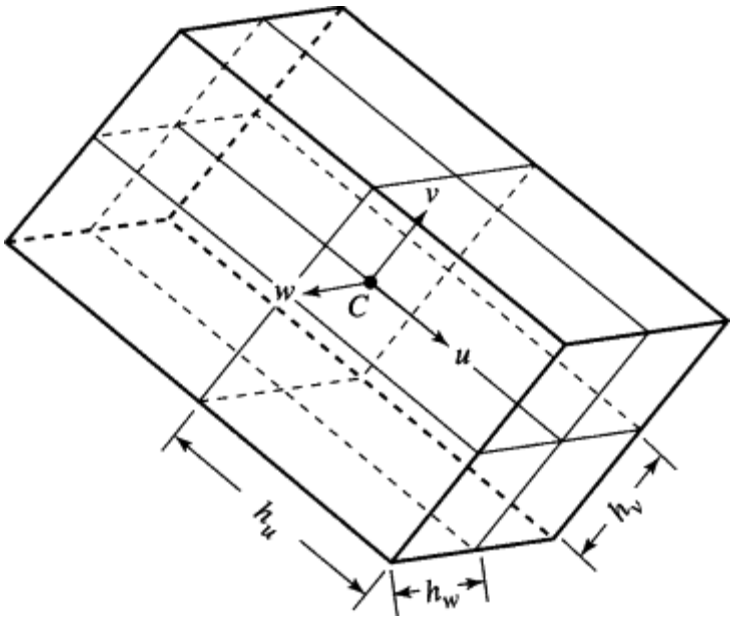

Figure 5: Diagram OBB

2.5 Game Engines, Features and Comparison

When selecting a game engine for this project, there are several features that are required. Firstly, a physics system is needed where the issue of collision detection is considered as objects must not phase through each other during game time. Furthermore, the physics system must implement other aspects of Newtonian physics such as gravity and velocity.

Secondly, the levels must be rendered including any updates to the scene and with many collisions that take place, they must be rendered. The comparison framework involves two of the most popular game engines within the game industry, Unity and Unreal Engine. While there are multiple other game engines such as Gamemaker studio, Clickteam fusion and others, the focus is specifically on Unity and Unreal.

2.5.1 Game Engines

2.5.1.1 Unity

Unity, developed and released in 2005 by Unity technologies is a game engine for game developers to develop and release their projects into the games market. Easily accessible to independent developers and large game companies, Unity allows development for both 3-D and 2-D game projects that can be released on multiple platforms such as desktop (Windows, Linux, OS X), mobile (Android and IOS) and consoles (PS5, Xbox Series X, Nintendo Switch).

2.5.1.1.1 Development

In the aspect of development, Unity allows developers to use either C#, a scripting language that takes elements of C and integrates it to the engine, or JavaScript which is an object-oriented approach, as described. Unity has two integrated development environments (IDE) that developers can utilize with MonoBehaviour and Visual Studio when creating and implementing scripts. Bhosale, Kulkarni and Patankar [17] states that using C# scripts requires attaching "individual behavioural scripts to each game component" within the game scene. This allows the data to be passed back and forth between the object within the game scene and the script applied. In the aspect of debugging, errors would often be listed within the console window that details the error, the location of the line within the script.

2.5.1.1.2 Physics

The physics system within game engines has a role in regards to making Newtonian physics look realistic, according to Salama and Elsayad [18] "mimic gravity, friction, velocity, bounciness, mass and other properties" allowing a realistic environment during gameplay. For the physics system, in the interest of object-oriented projects, there are two different systems that are used, Physx for 3-D environments and Box2D for a 2-D environment. Physx is a physics engine that is developed by Nvidia that renders the physics components faster by using the power of the Graphics Processing Unit (GPU). Originally available on dedicated PhysX cards, it can now be used on GeForce graphics cards that contains CUDA, a programming language that utilizes multithreading to complete tasks. Other games that were developed with Physx are Witcher 3: Wild Hunt, Batman Arkham Knight and Borderlands 2. Box2D is often used in the 2-D environment, developed by Unity technologies, to simulate colliders, physics material and rigid bodies.

2.5.1.2 Unreal Engine

Unreal Engine, which was released by Epic Games in 1998, also allows game developers to create their games and publish them. Projects can be developed in both 2-D and 3-D environments with 3-D being the more focused environment by developers. Common games that have been developed with Unreal Engine, include Fortnite, a battle royal multiplayer third person shooter, Gears of War 3 and Mass Effect 2. Unreal Engine recently released Unreal Engine 5 to the public in April 2022 to support development for next generation consoles, that being PlayStation 5, Xbox Series X as well as platforms such as Windows, Linux and OS X operating systems.

2.5.1.2.1 Development

Developing 2-D and 3-D projects within Unreal, the engine uses C++, an object-oriented programming language that is an extension to C. Unreal Engine users can use two IDE in the form of blueprints and Visual Studio. According to Unreal [19], Blueprint visual scripting is a scripting system that does not require extensive knowledge of C++ and using “Node-based interface to create gameplay elements”. The system uses an object-oriented approach as it uses classes and objects and it allows the developer to connect events, functions and data variables with each other in order to develop behaviour during gameplay. Blueprints is used to develop levels that incorporate elements such as checkpoints, level-up systems as well other aspects of the game such as HUD’s, player characters. Developers can use the console to produce a C++ code alternative to the Blueprint system where Visual Studio IDE is used to produce and debug errors that occur during execution.

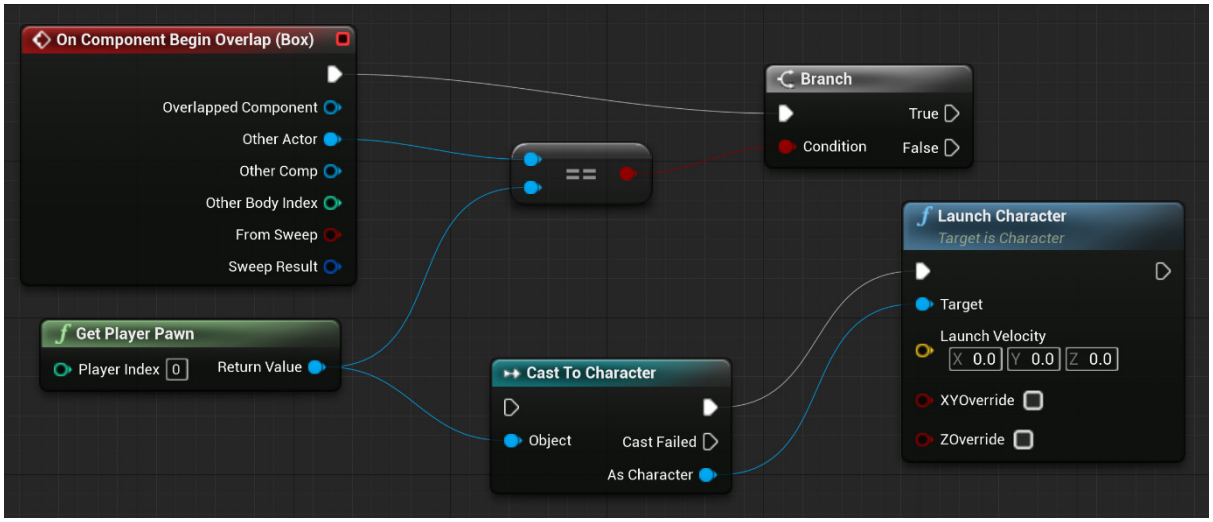

Figure 6: Unreal Engine Blueprint

2.5.1.2.2 Physics

In line with Unity's 3-D integrated physics engine, PhysX is the default collision detection system that is used to simulate physics systems such as collision detections when they occur. With regards to collision detection, both objects would need a "Physicsbody" and "WorldDynamic" type that contains a number of different responses. The two main responses are Trace, that includes a camera within the game scene and visibility as well as object responses that simulate object intersections. With one object's response of world dynamic being set to block and the other of PhysicsBody also set to block, this would create a scenario where both objects would collide rather than phase through. This is the most common example of physics within Unreal Engine.

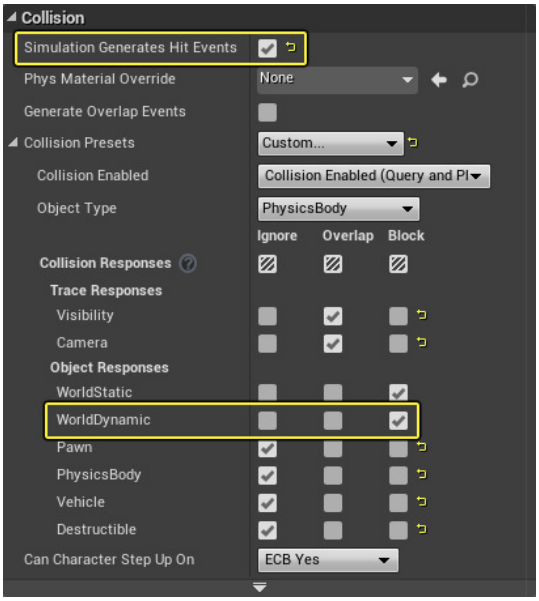

Figure 7:Object Collision Property

2.5.1.3 Comparison of Game Engines

With both Unity and Unreal Engine, the most developed engines for game development, a comparison must be made between the two to determine suitability for this project as well as addressing the problem. Three studies have been used and examined to determine the advantages and disadvantages of each engine that is commonly used in today's industry.

Study 1

In a comparison study that was conducted by Vohera [20] et al, Unity and Unreal engine were compared in terms of features such as physics engine, network/multiplayer as well as documentation, difficulty level and OS support. They found that for beginners, Unity is the best option as Vohera [20] states that there is "Very well-written documentation, several courses, and ready-made templates" to learn from and how each of the engine's features worked. However, the graphics aspect in comparison to Unreal engine is very poor and that if a project required huge and vastly complicated worlds, the engine would not be suitable. They found that the use of Unreal Engine it is better suited to developers who have advanced knowledge and skills. However, the disadvantage of using Unreal Engine is the requirement of higher hardware due to the graphics aspect of the engine. Overall, from this study, it appears that Unity is suitable for projects that are not complex and do not require impressive graphics as well as containing a vast amount of documentation and assets.

Conversely, Unreal engine is suitable for advanced users and contains a superior graphics pipeline.

Study 2

The second comparison study was conducted by Sharif who found that in the interest of physics, both Unreal and Unity are considered the best options, due to the physics engine Physx that was developed by Nvidia. In relation to AI, Unity and Unreal Engine are also considered the best for AI development. In the aspect of scripting, Unreal Engine is considered the superior of the two, Sharif and Ameen [21] claims that Unreal along with Godot “have many scripting languages that can work with them”. The programming language used in Unreal Engine is C++ and Python. With regards to both dimensions (3D and 2D) and development features, they found that Unreal Engine is suitable however the engine does focus more on 3D development. Finally, in terms of user accessibility, both Unreal and Unity are the most powerful and can be accessed with documentation and resources. Overall, it appears Unreal Engine is the best game engine in this regard, mainly due to the technological advantages in the graphics aspect.

Study 3

In the third and final study, Christopoulou explores both Unreal and Unity in depth. Both engines can integrate necessary tools however Unreal has a more complex user interface whereas in Unity, only a single window is used. Unity developers are required to have knowledge of C# or JavaScript in order to use the engine whereas Unreal Engine developers can use the Blueprint visual scripting system that can assist them in creating logic as well as objects and classes. With the use of resources such as tutorials and assets, both engines provide developers with many tutorials, with Unity being in the form of text based and Unreal within video. Unreal Engine however have paid tutorials, so developers who do not have the funds to access these cannot do so. Unity has better asset stores, with many of them free and can integrated into a single project, however Unreal requires payment in order to use them. Finally, Unity has fewer hardware requirements than Unreal Engine does, a similar comparison that was found in the first study by Vohera. Overall, Christopoulou and Xinogalos [22] states that Unity is "more suitable for beginners" due to a simpler UI, vast amounts of tutorials and resources in the form of assets that do not require high end hardware however C# or JavaScript is required. Unreal is more suitable for experienced developers where a steep learning curve is expected. It requires good hardware due to the graphics output.

Game Engine Selection

With each comparative study examined, conclusions were drawn at the end of each study, to determine all of the pros and cons of each game engine in the interest of this project. In an ideal world, Unreal engine would be used due to the technological advantages that it has over Unity however the decision was made to choose the Unity game engine over Unreal Engine due to a number of factors that is now examined.

Experience

The decision behind the selection of Unity as the game engine of choice is made based on several factors that are relevant due to the technical skills of the author as well as the project's scope. With regard to the technical skills of the author, experience was gained from using Unity Engine from a previous undergraduate course that involved a module of developing 3-D games using Unity. C# was learned which is a scripting language that is integrated into the game engine to utilize the game components such as rigid body and capsule colliders in scripts. In the aspect of Unreal Engine, the author has also gained experience with C++, an object oriented paradigm that is used for implementation within the engine. Due to the lack of experience with Unreal Engine 4/5, there would be a significant amount of learning with two of the studies highlighting a learning curve, in which the knowledge required would not be accessible due to the timescale of this project.

Easy to use

Unity is easy to learn whereas Unreal Engine would require more expertise and experience in order to create prototypes. It is often recommended by several comparison studies that Unity is a good choice to learn game development before moving on to Unreal Engine. There is a sufficient amount of documentation in the form of Cookbook - a Unity manual that can be accessed online only, providing resources and an asset store to use to implement a prototype. As cited by the studies analysed, Unreal Engine requires a steep learning curve which is not beneficial for a short development cycle.

Time

The third factor is the timescale for this project, this constraint impacts the project as time available for development of the prototype is very significantly short. Therefore, learning to use game engines other than Unity is not possible as it would take too long to learn how another game engine works, including debugging scripts during runtime and any other new features of it.

After examining each study conducted, Unity is the best suitable game engine for this project as highlighted by the three main factors cited in relation to time, accessibility and previous experience with the engine.

The problem now defined as well as work that is connected to this project has been established in the form of Fall Guys. With collision detection considered a classical problem, the literature review provides a clear state of the art of the physics as well as its subtopics, spatial partitioning, and bounding volumes. Spatial partitioning in the interest of games development is now used for collision detection and can be used within game engines as well as bounding volumes. Unity and Unreal engine were both examined in terms of their development features and ease of accessibility with the former being chosen for reasons stated by the author. However, the literature review was very limited due to the low amount of sources in relation to the nature of this project. Although the original sources were used, very little in the five year constraint was found. It can be determined that this decision was both good and bad, mostly for providing a more up to date state of the art at the expense of few resources. Development and testing can now proceed.

Chapter 3 – Specification

As stated, the literature review has examined the current state of collision detection as well as its subtopics including spatial partitioning and bounding volumes. The decision was made to develop a prototype using the latest version of Unity. However, there are several aspects to consider before applying this approach to the problem. In this section, a specification is made to address the problem that is the subject of this study.

Problem

The problem that is posed within the literature review is the implementation of a collision detection system within the game context that can detect object intersections during gameplay. As highlighted in the research, there are two categories of collision detection in the form of spatial partitioning and bounding volumes. With Quadtrees and Octrees being the most common algorithms within the field, Quadtrees would be suitable for a 2D game as Octrees is more suited for 3D games. There are also further questions about how it can be implemented within the Unity engine as it will require a few additional elements in order for the implementation to function as intended. For example, what data types can be used and which data structure is best to use for both performance and efficiency? The literature review also establishes that the Unity engine is suitable for this project and that the prototype has to be developed within a 2-D environment. This includes rendering the level as well as simulating the physics aspect. Furthermore, a UI must be implemented, this includes a main menu and a pause menu that involves freezing the gameplay until a button is pressed. Finally, a level must be constructed that demonstrates both the physics and UI elements. Due to the time constraints, one level will be developed that contains two or more obstacles. With the problem now formulated, the requirements can now be identified to address the problem stated.

Requirements

With the problem above described in detail, the requirements can be formulated to achieve a solution within the prototype that must be developed. With Unity being the engine of choice, a prototype must be developed in a 2-D environment that contains physics and UI as per the aim of the project.

Physics

As the main element of the gameplay, the physics system implemented would have to represent an accurate simulation of the laws of real-life physics. For example, when an obstacle collides with the player, the player will be forced to move backwards due to being hit. Other aspects such as gravity and velocity will also have to be implemented. The physics within Unity relies on two main components, a collider, and a rigid body. Colliders are components that can be in a form of a shape that grounds the object it's applied to. Colliders are used to determine when an object collides with another in a script. The rigid body is another component that simulates physics within the physics engine in Unity. The rigid body contains a number of different elements such as mass and freeze rotation. This is so that the object does not rotate when moving or jumping or when it's in collision detection mode, where two modes are present, continuous and dynamic. These two components are important for every object as they are responsible for the physics calculations to take place as well as not phasing through other objects. To determine the best possible approach of collision detection, Quadtrees will need to be investigated further to determine the best approach to be implemented.

UI

A part of the aim is to also develop a user interface so that the user can interact when running the application. The main menu is the first screen they will see and in the majority of computer games this is where they can adjust their settings. This includes key bindings and graphics settings as well as start new game files. The main menu will allow the player to choose a level that can be played or to exit the game. The second aspect of the UI is the pause screen. In most games, pressing the escape key will freeze gameplay and draw a UI over it that contains widgets in the form of buttons. The pause menu will be basic, and the player can pause if they need a break and later resume the game. Finally, the third UI screen is when the level is completed, and here the level scene will change, and a UI must be present for the player to return to the main menu. In the case of Fall Guys, a UI will appear to determine the next level. The design of the UI must be simple and straightforward as well as assisting in smooth navigation between screens, this can be done using buttons where for example, when a level is selected and the player presses the button, the level will then be loaded.

Level

A level will be developed, where the player will be required to reach a finish line while evading obstacles. The level must be playable where the character can move and jump as well as respawning when knocked off the stage. Two obstacles will be present that are inspired by Total Wipeout and Takeshi's Castle. The level must also include elements such as checkpoints for when the player is knocked off the level, and they must respawn at the checkpoint they passed. The level must be rendered without issue to not affect certain parts where collisions are involved. For example, game lag happens when a frame is not loaded and it moves onto the next, it breaks the immersion of the player. When the level is fully developed and tested to ensure that issues that did not appear during playtime also do not appear within the executable file, the level can be included in the finished executable file alongside the main menu and the finished UI.

Chapter 4 – Design and Implementation

With the specification now formed, the prototype can be designed and developed to meet the requirements made. In this section, the development of the prototype will be explored by looking at the implementation of collision detection between objects as well as other aspects such as user interface, level design and coding practises. Furthermore, testing will also be carried out simultaneously alongside implementation to ensure that the game functions as expected.

Overview of Unity

Within Unity, there are many components that are required, the foremost of which is the physics system, as Box2D is the default system for simulating physics in 2D environments. This includes two important elements, colliders, which may be box, circle or capsule and rigid body. This can be used in reference to collision detection when implementing scripts. The second component is the user interface, this is where the user will interact with the canvas which contains buttons and toggle buttons. This serves the purpose of transitioning between menus and gameplay. The third component is rendering, with a lot of objects within obstacle courses and collisions occurring, it is crucial to render them as smoothly as possible without lag. Finally, the scripts are produced with C# that takes the components stated above and uses them to create the gameplay.

Physics

Box2D is the default physics system when developing alpha and beta builds for a game project. To prevent objects phasing through one and another, there are two main elements that can be used. Colliders come in a few different shapes depending on the object. The collider serves to bound everything within the object, the bounding line can be altered depending upon the developer's preference. However, it can also be used with the rigid body which is useful to work together. The rigid body connects the object to the physics engine, with further usage of its position that is given to the object. The final component of the system is a physics material, applying it to the object that contains two attributes, friction, and bounciness.

User Interface

The user interface in deeper detail, requires two parts, a canvas, and a panel. The canvas, a parent object, serves a purpose of covering the screen so that the UI can be rendered. The panel is a component that is included along with the canvas, tasked to group UI controls that can be accessed by the user. Some of the controls associated with the panel are buttons which are commonly used in games to transition between gameplay and the user interface.

Rendering

When the level is loaded, all objects within the game scene are rendered. There are multiple pipelines that can be used for game development. Examples include Universal Render Pipeline (URP), High Definition Render Pipeline (HDRP) and Scriptable Render Pipeline (SRP). The pipeline used is built in that the game engine provides, with options to configure how the game can be rendered with different paths in addition to command buffers and call-backs for further development.

Scripting

The scripts within Unity assist in connecting the components in the engine together. For example, it connects to the physics engine when the player object collides with another object as well as using the audio system to produce a sound when the player gets knocked back. The variables created within the scripts enable access to the components in order to use them. These are often applied to game objects within the scene, regardless of what components are attached. Interaction between the different components often occur through scripts and act as a bridge between them. Each component can be viewed as a small cog in a bigger machine, wherein one component is connected to the other.

Each of the components discussed are essential within the process of software design for systems to work effectively within an obstacle course game.

4.1 Software Design

The first stage of the process is to determine how the level, UI and obstacles are designed. The ideas for these were drawn on paper as an initial method of development.

4.1.1 Level Design

The level design consists of the player using platforms to navigate from start to finish. The first level serves as a straightforward level that does not contain complex sections that will challenge the player. The level does not contain high quality graphics but rather simple objects that can be later improved upon beyond the deadline for this project. The majority of the level consists of both up and down slopes and small breathe sections in between obstacles and checkpoints, so that when the player does get knocked off, they can respawn. The level requires all components specified above.

4.1.2 UI

The user interface design is very straightforward and basic, consisting of buttons and titles on the screen with the former being the main component for navigation during the game. With all UI elements, all components are centred to remain consistent and not confuse the player's vision. The main menu consists of multiple elements such as buttons and titles, there are two buttons, one to select a level, the second to exit the game. When a level is selected, it is pinned on the top left rather than the middle, thus allowing for the possibility of adding more levels to extend the game in the future. The finished screen is again, very simple and straightforward, only a button to take the player back to the main menu.

4.1.3 Obstacles

Ideas for multiple obstacles were designed on paper with each involving a strategy for the player to apply to succeed. For the purpose of this project, only two obstacles were chosen that are widely recognised in Total Wipeout and Takeshi's Castle. The first of these is the boulders, a popular obstacle that requires the player to be concise in their timing of movement, as failure to do so will result in them being knocked back and possibly off the level. In reference to the physics element, the boulders are usually moved down a slope that will head towards the player when they move up. In the show, there are small spaces where the contestant can move to avoid them, before moving up the slope again. The aim is to

replicate it however it must be designed so that the space between the start and the end are reasonable.

Figure 8: Takeshi's Castle Boulders Course

The second obstacle is the bouncing balls, one that is popular within Total Wipeout. The aim is for the player to reach the other side without falling off into the water. There are four balls in total that when jumped upon, it can be challenging to control movement. This is the final obstacle before the finish line, so the player will have to remain careful as to where they land. The concept is very simple however, the implementation aspect will consist of altering how bouncy the balls should be in order for it to be passable.

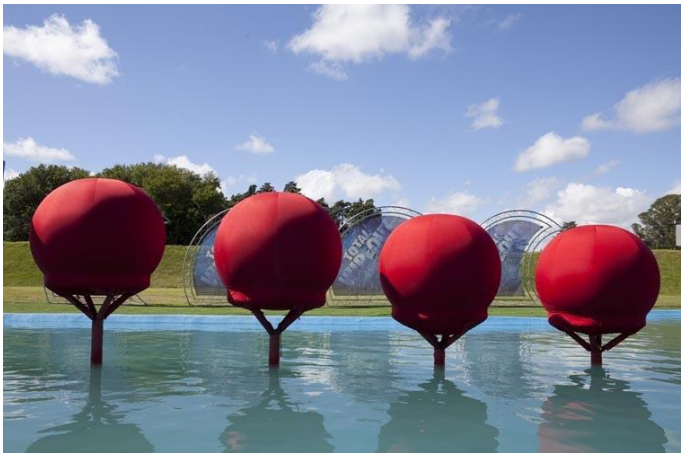

Figure 9: Balls from Total Wipeout

The design of the prototype remains straightforward due to the time constraints where the groundwork is designed that can be extended in future work. The graphics can also be enhanced to add detail however for the nature of this project, the graphics is suitable as the implementation of physics remains a high priority.

4.2 Development

Once the design is completed, the development can begin, referring to the design preparation and realising it within Unity. Each component contains a section that details how it has been coded and what each part does within gameplay. Finally, an overview of how the system operates, what is used and what is given to Unity is documented

Physics

As explained earlier in the design and overview of Unity, the implementation of the physics aspect of the prototype revolves around the movement and collision of player object and several other objects, ranging from obstacles such as boulders and bouncing balls to checkpoints and finish lines. In relation to the overall solution, the ability to simulate Newtonian physics as realistically as possible is paramount.

Player

The player object is a simple capsule where the design is similar to Fall Guys. The object has several components that are provided, including Capsule Collider, Rigidbody2D and Player Controller Script. The capsule collider covers the shape of the object in an event of a collision with another object. The Rigidbody2D is applied as well as the script that is responsible for the movement and collisions. The tag is set to ‘player’ which can be used as reference in other scripts.

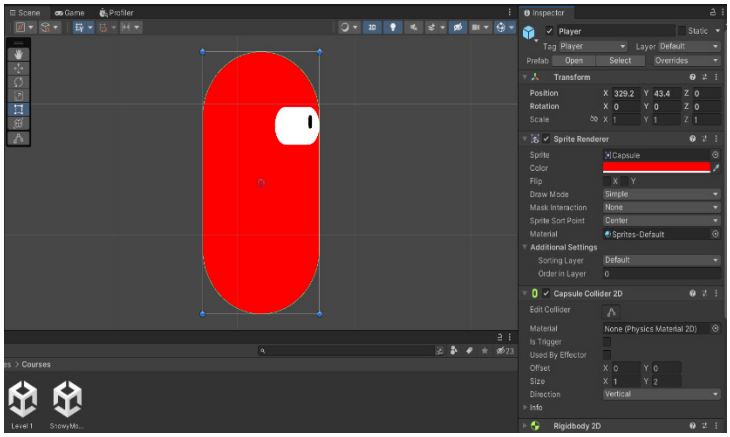

Figure 11: Creation of the Player Object

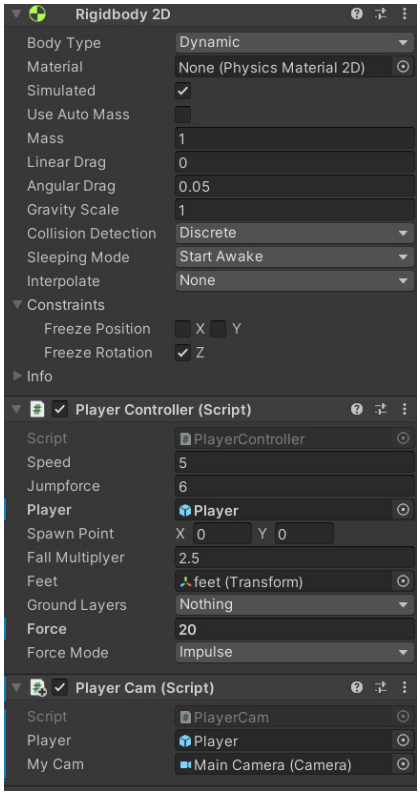

Figure 10: Components of the Player

```
    // Update is called once per frame
    void FixedUpdate()
    {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///The horizontal input stores the axis of horizontal in Input
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        horizontalInput = Input.GetAxis("Horizontal");
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///If the Key pressed is A
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        if (Input.GetKey(KeyCode.A))
        {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Rotate the player to face left
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            rbPlayer.transform.localRotation = Quaternion.Euler(0, 180, 0);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Set the velocity vector to a new vector that contains the horizontal input multiplied by the speed minus 1 and the velocity of y
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            rbPlayer.velocity = new Vector2(horizontalInput * speed - 1, rbPlayer.velocity.y);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Freeze the rotation so that it only faces left
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

            rbPlayer.constraints = RigidbodyConstraints2D.FreezeRotation;
        }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///If the Key pressed is D
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        if (Input.GetKey(KeyCode.D))
        {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Rotate the player to face right
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            rbPlayer.transform.localRotation = Quaternion.Euler(0, 0, 0);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Set the velocity vector to a new vector that moves right
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            rbPlayer.velocity = new Vector2(horizontalInput * speed + 1, rbPlayer.velocity.y);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Freeze the rotation so that it only faces right
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            rbPlayer.constraints = RigidbodyConstraints2D.FreezeRotation;
        }
    }
```

Figure 12: Player Movement Code

The player controller script is responsible for movement whenever a key is pressed. Player movement is coded in the interest of horizontal and vertical movement. Horizontal movement consists of moving left and right, the keys being A and D. When the A key is pressed, the player will face left, rotating it 180 degrees, and update the velocity vector of the rigid body component so that the player is moving left. Finally, the constraints of the rigid body involve freezing rotation so that the player does not fall to the ground. When D is pressed, the code is reversed, with the player facing and moving right while the constraints remain the same.

```
    void Jump()
    {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Create a movement vector to store the x component of the velocity of the rigidbody and its jumpforce
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        Vector2 movement = new Vector2(rbPlayer.velocity.x, jumpforce);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///set the velocity of the rigidbody to the movement vector
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        rbPlayer.velocity = movement;
    }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///A function determining if the player is grounded
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    public bool isGrounded()
    {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Create a Collider2D variable that checks if the player is grounded.
///This uses overlapCircle which is a method that takes two parameters, the feet of the player(the position) and the ground layers
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        Collider2D groundCheck = Physics2D.OverlapCircle(feet.position, groundLayers);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///if the ground check variable is not null
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        if(groundCheck != null)
        {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///The player is grounded
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            return true;
        }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///The player is not grounded
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        return false;
    }
```

Figure 13: Player Jump Code

When it comes to jumping, there are two functions that are required for this. The first function is a jump that focuses on the velocity of the player's rigid body and the movement vector that attains the x of the velocity component of the rigid body and the jump force which can be any integer value. The velocity vector is set to the movement vector where the player will move along the Y axis. The grounded function determines if the player is not in the air, and he is on the ground. The function is a Boolean, meaning that a true or false value is returned at the end.

```
public void OnCollisionEnter2D(Collision2D collision)
{



            if (collision.gameObject.CompareTag("boulder"))
            {
                //get the contact point
                ContactPoint2D contactPoint = collision.GetContact(0);
                //Store the player position
                Vector2 playerPos = transform.position;
                //calculate the difference between the point of contact and the player position
                Vector2 dir = contactPoint.point - playerPos;

                //normalize the vector
                dir = -dir.normalized;
                //get the Rigidbody's velocity and set it to the new vector
                GetComponent<Rigidbody2D>().velocity = new Vector2(0, 0);
                GetComponent<Rigidbody2D>().inertia = 0;
                //Add the force to the rigidbody so the player is knockedback
                GetComponent<Rigidbody2D>().AddForce(dir * force, forceMode);
                rbPlayer.constraints = RigidbodyConstraints2D.None;


            }
}
```

Figure 14: Boulder Collision and Knockback Code

On collision enter is where an intersection between two objects occurs. In this case a boulder is a common obstacle within the game, using a collision parameter from Collision2D, a check is made if the game object colliding is a boulder. If it is true, the knockback has to be implemented, that means the player will be moved backwards to represent the real-world physics. We start by getting the contact point of where the collision happened while creating two vector variables, one that stores the player position and the other a direction vector that determines where the player will be moved when the collision happened. The player position contains the position of the transform vector of the player. The direction vector is normalized before it can be used in the calculations. Using the rigid body component of the player, we get two elements, the velocity vector and inertia. Finally, we get one more element from the component, AddForce to which we use the direction vector, the force which is a float and its force Mode. This line of code is important as it sends the player backwards.

This method only focuses when the object that contains a collider component has intersected with another object that allows the player to move past it. This method can be used for pickup items, checkpoints that are placed through the course and other objects that do not require colliding and not phasing past it.

```
public void OnTriggerEnter2D(Collider2D collider)
{
    if(collider.gameObject.tag == "Player")
    {
        highlight.SetActive(true);



        Debug.Log("Player has collided With Tile: " + collider.gameObject.name);
      // Debug.Log("Player Position X :" + playerPosX);
    //   Debug.Log("Player Position Y :" + playerPosY);
    }
```

Figure 15: OnTriggerEnter2D collision code

In the case of checkpoints, this method takes in an argument collider that is connected to a Collider2D class. Using this parameter we can determine if the collider has intersected with another game object whose tag is ‘player’. This code enables any action to be followed through when an object connects to another object that only has a tag called ‘player’.

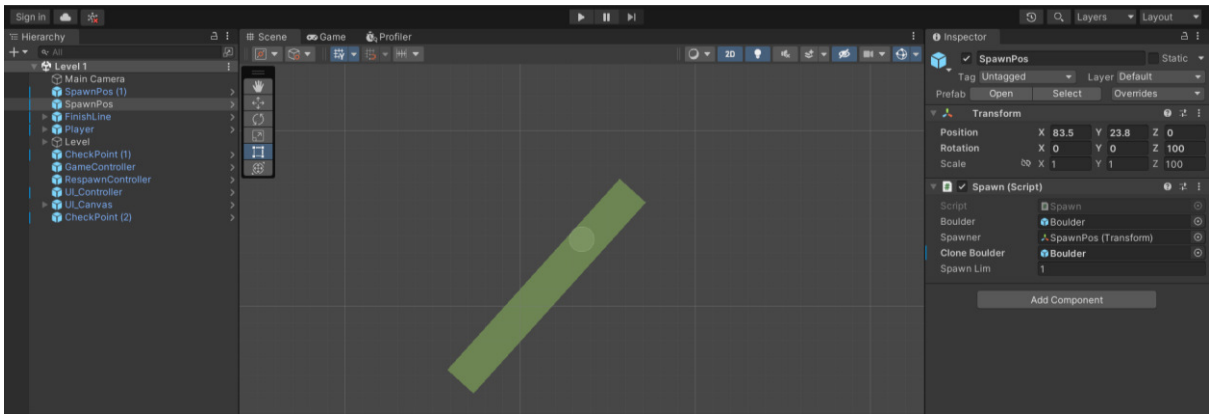

Figure 16: Creation of spawn objects

The spawn object allows boulders to spawn on a continuous loop throughout the duration of gameplay. As seen in Takeshi’s Castle, boulders continuously appear until the contestant is either knocked out or gets past the obstacle. The concept is relatively easy to understand,

as an empty game object is created that does not require a sprite but a script, one of which requires three important components: The prefab of a boulder, the transform of the "spawnPos" object and a clone of the prefab.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Declare variables required
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    public GameObject boulder; //The original Prefab
    public Transform spawner;


    GameObject mMyClone;
    public GameObject cloneBoulder; //clone of clone
    public int spawnLim;

    float mytimer = 0f;
    // Start is called before the first frame update
    void Start()
    {
        //objSpawn();
    }

    void Awake()
    {


    }

    // Update is called once per frame
    void Update()
    {
        mytimer -= Time.deltaTime;



        if(mytimer <= 0f)
        {
            ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            ///Create a variable that will create a bolder at the position of the spawner object and the rotation
            ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            mMyClone = Instantiate(boulder, spawner.position, transform.rotation);
 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Set the timer to 6 seconds
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            mytimer = 6;
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Destroy the clone at every 6 seconds since created
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            Destroy(mMyClone, 6);
        }
```

Figure 17: Creating boulders

During the design of the level, one of the most common obstacles that is featured on Takeshi's Castle is the slope that contestants must climb while boulders roll down towards them to knock them off balance. Inspired by this, the first obstacle the player will encounter is the boulder and in order to succeed they need to time their movements carefully to not get hit and fall down the slope. The spawn script is used to achieve a loop that continuously creates a clone of the boulder prefab. The implementation is done in the update function however this can be created in a function that can be called. A "Mytimer" variable is important as it will be used to measure how long the boulder is created. To determine if a boulder is spawned, an 'if' statement is used to ascertain if the timer variable is less than or

equal to 0. If it is, the clone variable will instantiate a boulder object from its prefab, creating it at the position of the spawner object which is empty, and rotate it at its transform. The destroy function destroys the game object during gameplay, this can commonly be used when a pickup item is collected by the player therefore it needs to be removed. The destroy function requires two arguments, the game object which in this scenario is the clone variable and the time that it should be destroyed, that being 6 seconds. The boulder will be destroyed every 6 seconds, fulfilling the loop that remains until the level is completed.

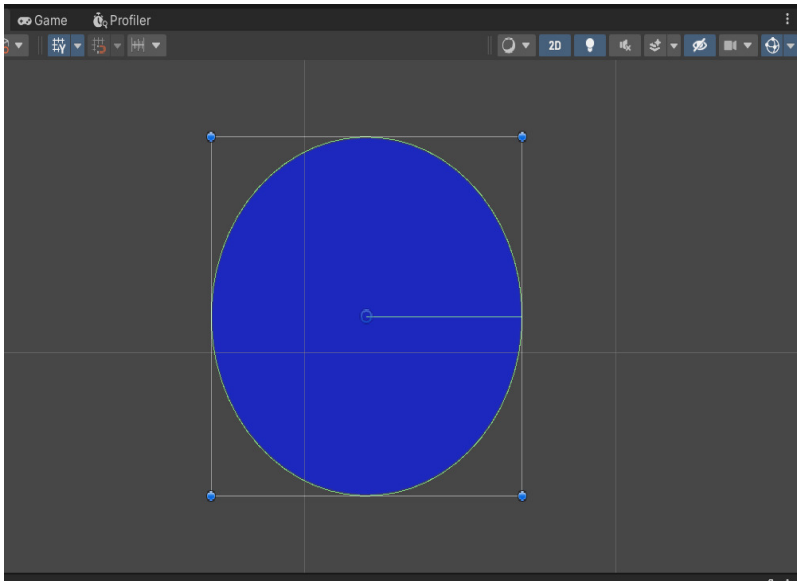

Figure 19: Creation of Checkpoints

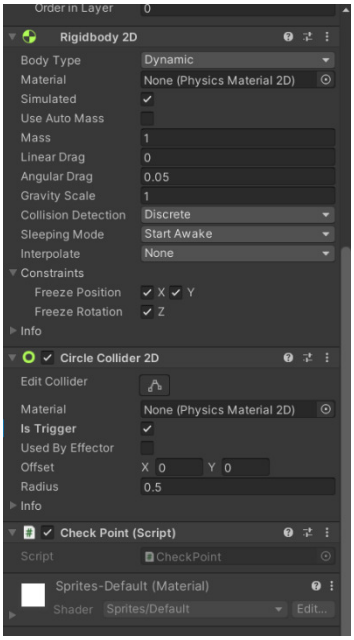

Figure 18: Components of checkpoint

The checkpoints serve that the player can pass through and be able to respawn at the position of the checkpoint instead of the beginning of the level. This contains the RigidBody2D and a circle collider 2D component. This object does not contain a script as the code is developed in the player script.

Rendering

Following from the rendering component of Unity in addition to the design, the camera is the only component that is related. This is mostly due to the built-in rendering pipeline with the rendering paths that can be accessed in the camera. The design of the camera revolves continuously following the player wherever they move.

The camera is one of the important components of the game as it gives the player a view of the player within the game itself, as well as any objects that come into view. The tag is set to 'Main Camera' by default so that it can be referenced in other scripts that require it. The position of the camera in the Z axis is moved back by -10 so that it can view the player and not get too close.

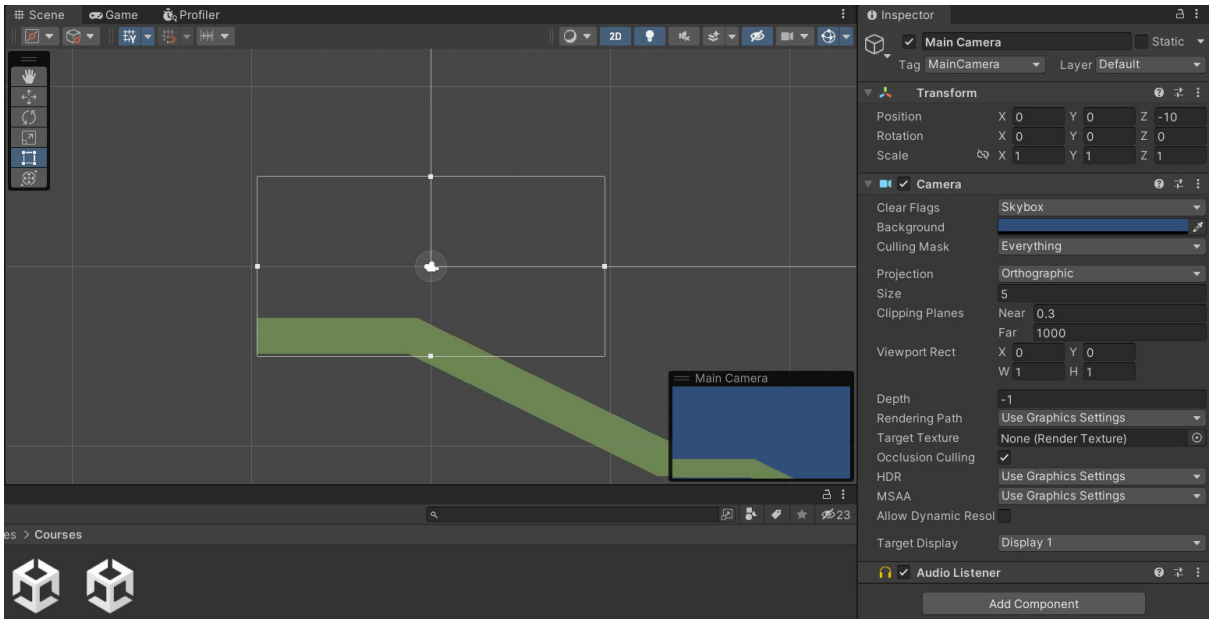

Figure 20: Camera Components

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

public class PlayerCam : MonoBehaviour
{
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Declare variables required
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    public GameObject Player;
    public Camera myCam;

    // Update is called once per frame
    void Update()
    {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Set the position of the camera
///Create a new vector3 that focuses on the x,y and z position of the player while keeping a short distance.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        myCam.transform.position = new Vector3(Player.transform.position.x, Player.transform.position.y, Player.transform.position.z - 10);
    }
}
```

Figure 21: Camera Implementation

The camera script is designed to have the camera follow the player throughout the level. It requires two main variables, the player and the camera. As a result of the camera following the player, it needs to be updated continuously with a new vector value created for its position. In terms of distance, the z axis of the vector can be subtracted depending on how far the camera can be from the player.

User Interface

Following from the design of the user interface, the implementation focuses on the functionality of the canvas, panel and its elements. Transitions between gameplay and UI are handled with mouse input, clicking on buttons for change to occur.

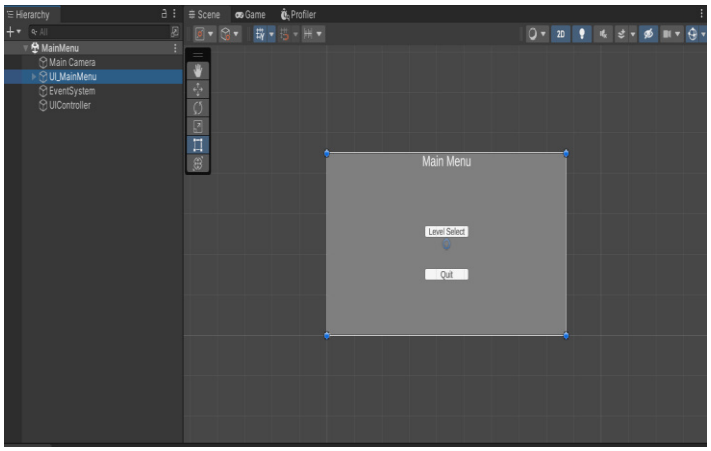

Figure 22: Creating Main Menu

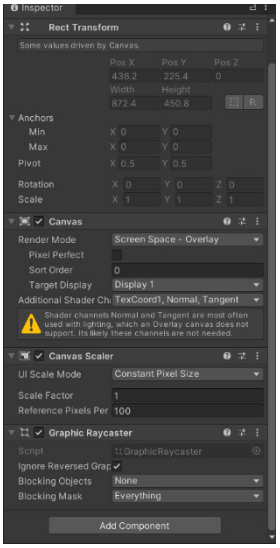

Figure 23: Canvas Components

The main menu's principal component is a canvas, which allows all of the elements of the UI to be rendered and to be stored. An option of how it can be rendered is presented to the user. The second component is a panel which is the next main component that groups all of the objects such as buttons, texts and images. There are two buttons, one to select a level, the other to exit the application.

The main menu script deals with the first scene when the executable of the project is running. Whenever a game is launched, the main menu is the first screen that the player will see, and they are faced with the decision whether to load the game where they left off or to change a part of their settings, such as audio volume or sensitivity of the controls. The script requires several game object variables that connect to the components of the UI such as buttons, panels and text. These are part of fields that are serialized.

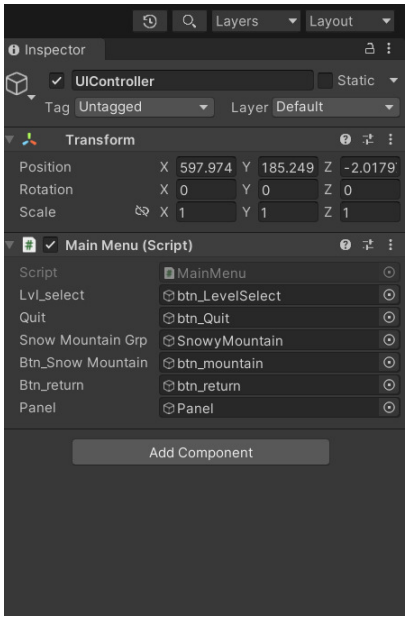

Figure 24: UI Controller Component

```
public class MainMenu : MonoBehaviour
{
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Create serialize fields for the UI components to be used in this script
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    [SerializeField]
    public GameObject lvl_select;

    [SerializeField]
    public GameObject quit;

    [SerializeField]
    public GameObject snowMountainGrp;

    [SerializeField]
    public GameObject btn_SnowMountain;

    [SerializeField]
    public GameObject btn_return;

    [SerializeField]
    public GameObject Panel;


    // Start is called before the first frame update
    void Start()
    {
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///For every button created, create an instance of the button that gets the component of the button
///Add a listener to the button that requires a function
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        Button btn = quit.GetComponent<Button>();
        btn.onClick.AddListener(Quit);

        Button btnlvl = lvl_select.GetComponent<Button>();
        btnlvl.onClick.AddListener(LevelSelect);

        Button btnReturn = btn_return.GetComponent<Button>();
        btnReturn.onClick.AddListener(Back);
```

Figure 25: Main Menu Code

During the start function, the buttons are set up by creating an instance of the button class, getting the button component of the UI button. The instance variable requires a listener to

listen for a function that will run when a button is clicked. The “AddListener” function requires a parameter in the form of a function.

```
////////////////////////////////////////////////////////////////////////////////////////////////////
///Quitting the application function
////////////////////////////////////////////////////////////////////////////////////////////////////
    void Quit()
    {
////////////////////////////////////////////////////////////////////////////////////////////////////
///Log that the player is quitting and quit the application
////////////////////////////////////////////////////////////////////////////////////////////////////
        Debug.Log("Player is Quitting");
        Application.Quit();
    }
////////////////////////////////////////////////////////////////////////////////////////////////////
///Function for moving back to main menu
////////////////////////////////////////////////////////////////////////////////////////////////////
    void Back()
    {
////////////////////////////////////////////////////////////////////////////////////////////////////
///Show the level select component of UI
////////////////////////////////////////////////////////////////////////////////////////////////////
        lvl_select.SetActive(true);
////////////////////////////////////////////////////////////////////////////////////////////////////
///Show the quit button
////////////////////////////////////////////////////////////////////////////////////////////////////
        quit.SetActive(true);
////////////////////////////////////////////////////////////////////////////////////////////////////
///Hide the snow mountain group in the UI
////////////////////////////////////////////////////////////////////////////////////////////////////
        snowMountainGrp.SetActive(false);
////////////////////////////////////////////////////////////////////////////////////////////////////
///Hide the level select button
////////////////////////////////////////////////////////////////////////////////////////////////////
        btn_return.SetActive(false);
    }
////////////////////////////////////////////////////////////////////////////////////////////////////
///Load the Snowy Mountain Level
////////////////////////////////////////////////////////////////////////////////////////////////////
    void LoadSnowMtn()
    {
////////////////////////////////////////////////////////////////////////////////////////////////////
///Change scene to the first level
////////////////////////////////////////////////////////////////////////////////////////////////////
        SceneManager.LoadScene("Level 1");
    }


}
```

Figure 26: Code for Qutting the game

The majority of the functions associated with the script consist of showing and hiding UI components using the SetActive function. The function requires a Boolean, true or false. True shows the UI components whereas false hides it. For example, when the back button is clicked on in the level select window, it needs to hide the components when level select is present and show the components associated with the main menu. Apart from SetActive, there are two other functions that are used. The Application.Quit() is a system.generic library function that when the executable is running, clicking the quit button will allow the user to quit the application. Finally, in terms of loading levels, an extra library is required in the form of Scene Management, which provides a number of functions, one of them being load scene that requires a string parameter where the name of the scene is used.

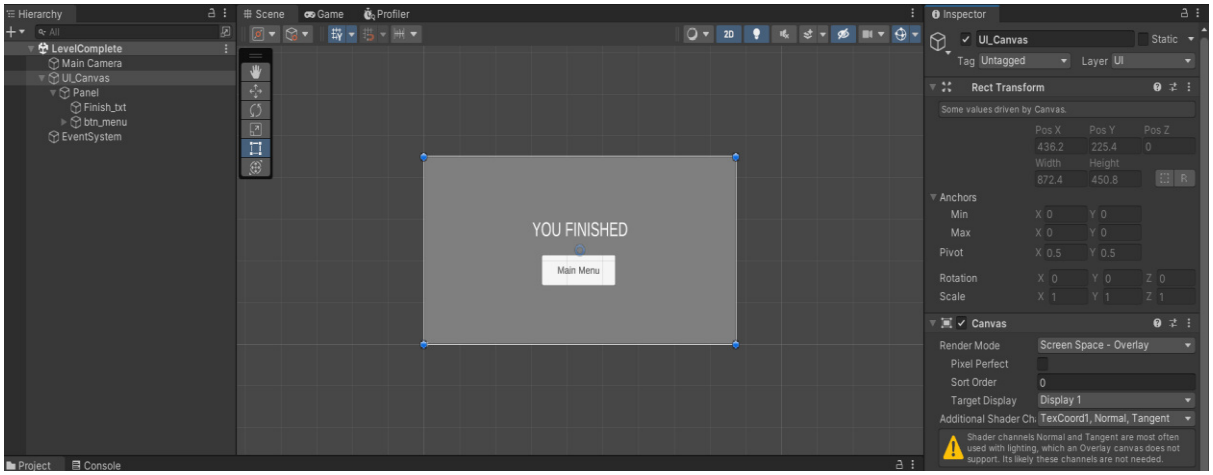

Figure 27: Creating Level Completion Screens

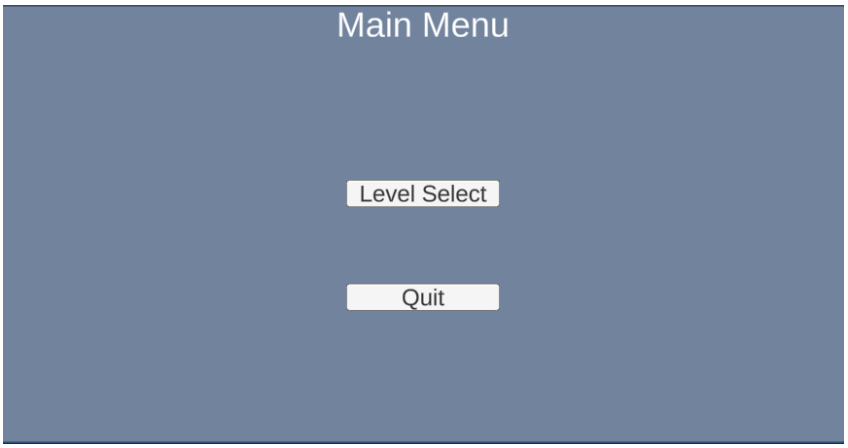

Figure 28: Main Menu

Figure 29: Level Select

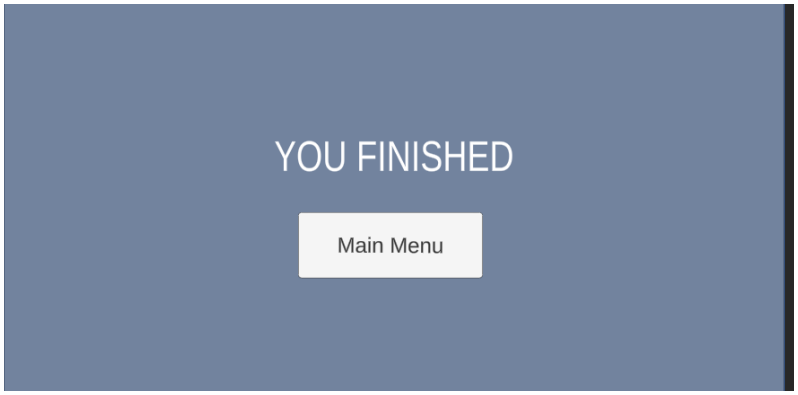

Figure 30: Level Completion Screen

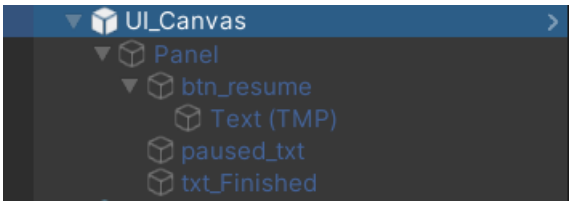

Figure 31: Gameplay UI Objects

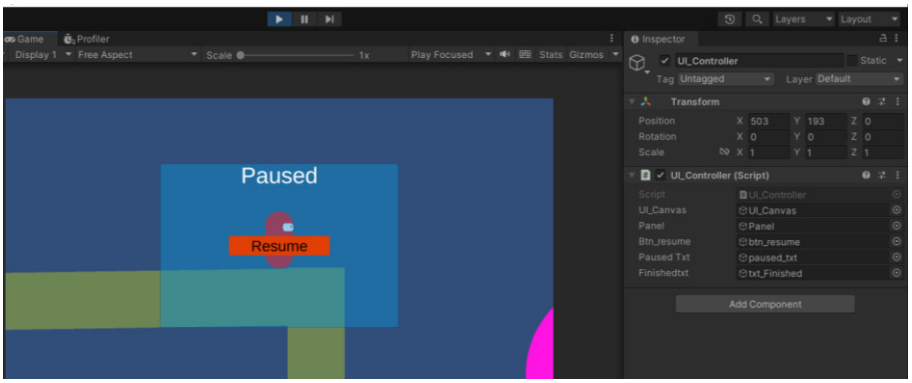

Figure 32: Pause Menu

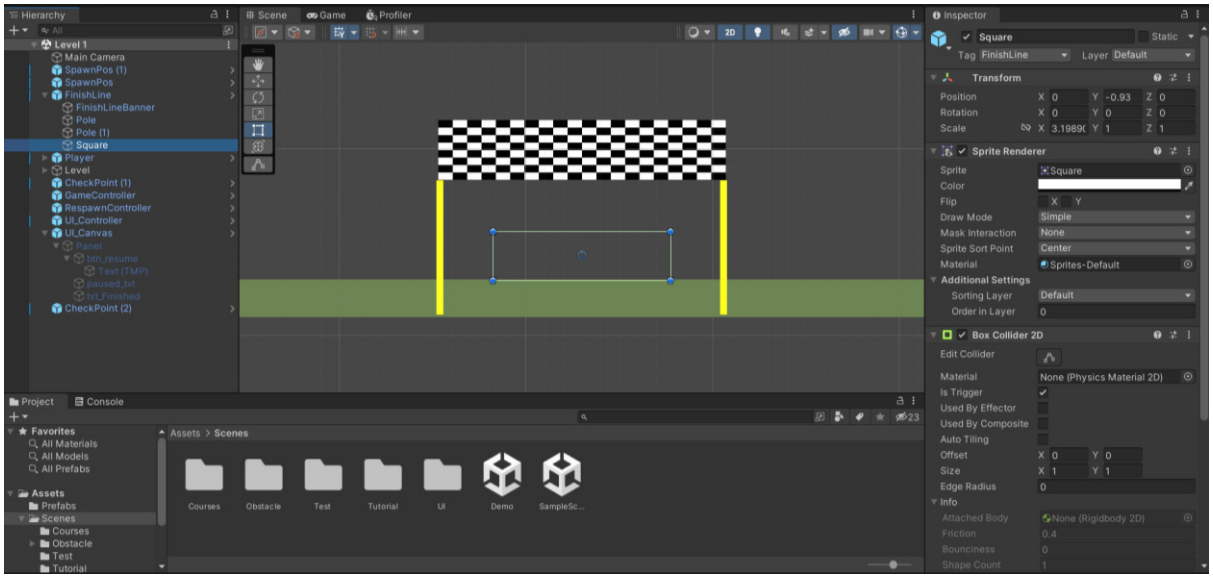

Figure 33: Creating Finish Line

The finish line is the object the player must reach to finish the level. The finish contains an empty square object that contains a box collider 2D component that is needed to transition to the finished level screen.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5
6   public class FinishLine : MonoBehaviour
7   {
8
9       // Start is called before the first frame update
10      void Start()
11      {
12
13      }
14
15      // Update is called once per frame
16      void Update()
17      {
18
19      }
20
21      private void OnTriggerEnter2D(Collider2D collider)
22      {
23  ////////////////////////////////////////////////////////////////////////////////////////////////////////////
24  ///if the collider intersects with the player
25  ////////////////////////////////////////////////////////////////////////////////////////////////////////////
26          if(collider.tag == "Player")
27          {
28  ////////////////////////////////////////////////////////////////////////////////////////////////////////////
29  ///Load the scene that is next in the build list, this is reference to the executable file that builds all levels
30  ////////////////////////////////////////////////////////////////////////////////////////////////////////////
31              SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
32          }
33
34
35
36      }
```

Figure 34: Finish Line Code

The script only requires an OnTriggerEnter2D method that is similarly used in the player script. An ‘if’ statement is crucial to determine the loading of the next scene. If the collider of the tag is the player, using the scene manager, load the next scene within the build list that is used in the executable file that contains all the scenes.

Chapter 5 – Evaluation

Following examination of the design and development of the prototype, it will be evaluated with reference to its areas of both success and failure. In this section, an evaluation is conducted into the performance of the prototype that involves the collision detection, player movement and UI. Demonstrating the functionality of this, a framework is presented, highlighting a clear aim that references the previously specified requirements in addition to the test results, concluding with a brief discussion of these.

Test Area

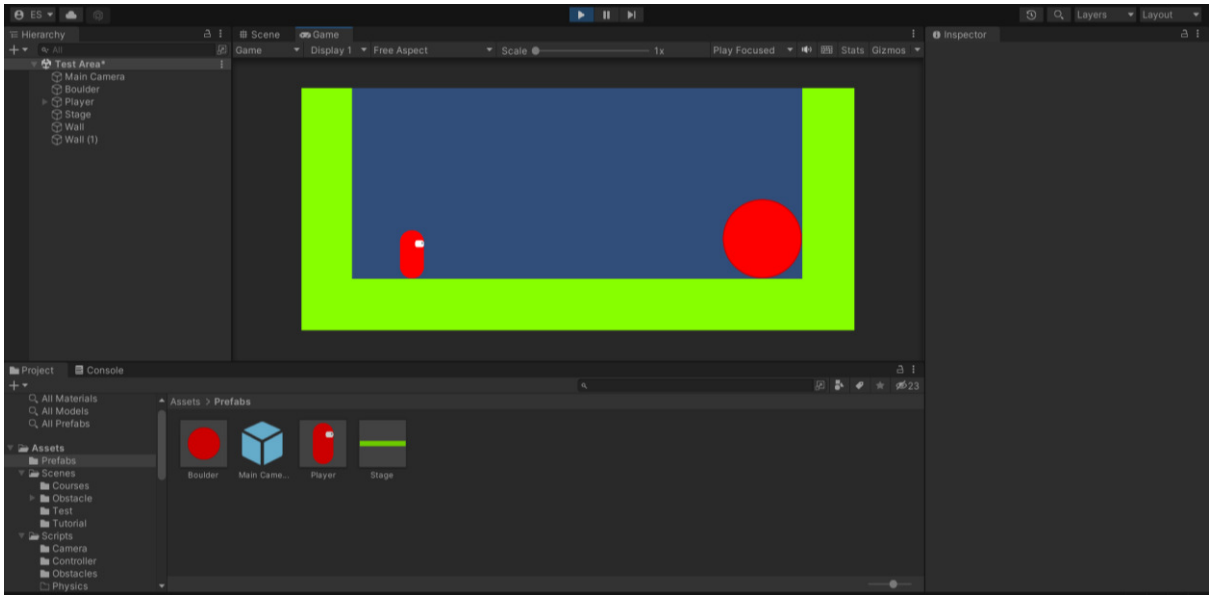

Figure 35: Test Area

During the implementation, a scene named 'Test Area' is created. The nature of this scene is to test the functionality of the scripts that will be used in the main prototype. The benefits of this are to determine if there are errors within the script before it can be applied to the main game and to fix the issue. It can be used to test and further develop each gameplay component before it can be added to the level scene so that it can be added to the scene list when building for an executable file.

Collision Detection

The collision detection phase of the prototype contains two types of methods, OnTriggerEnter2D and OnCollisionEnter2D. Success depends upon whether any object has intersected with the player. To measure this, a useful line of code is used, “Debug.Log();” is a line of code that prints a string to the command line that is logged during runtime. This is useful for debugging lines of command when troubleshooting as well as determining if the code works as expected.

Collision Detection			
Test No	Aim	Success?	Notes
1	The player will collide with the boulder obstacle	Yes	The player has collided with the obstacle
2	When the player has collided with a killzone object, the player would be respawned at checkpoint	Yes	The player has collided with the killzone and is respawned at checkpoint
3	When the player collides with a boulder at a different angle, the knock back code will still function	No	Although the knockback works when both objects are on the same grounds, however when this changes when the boulder collides with the player on a slope, it does not function. It may be possible to examine this further.

Figure 36: Table of Collision Detection Tests

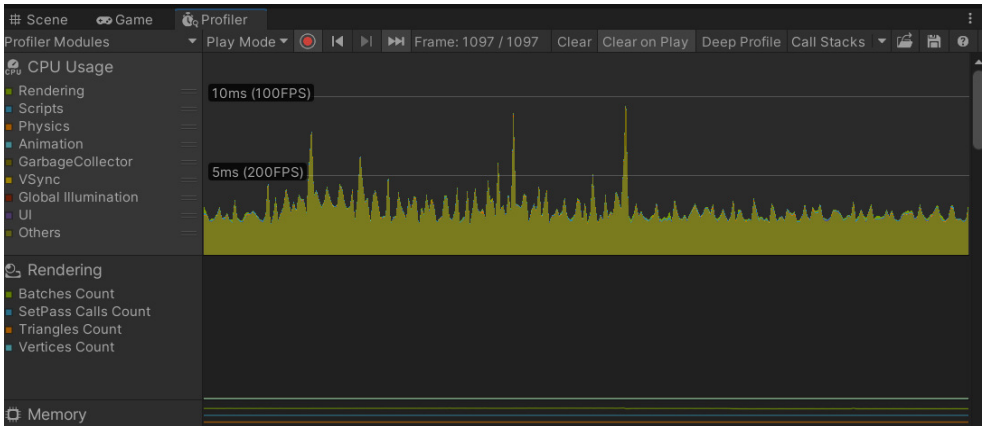

Figure 37: CPU and Rendering Performance Test

The performance of the collision detection tests is captured using the profile system within Unity. This is very useful to determine how the game's performance is measured and whether it needs further optimization. The tests carried out in relation to this are focused on both CPU, memory and rendering. For the single collision detection that occurs, the results from the CPU usage indicates that the spikes when they occur mostly relate to the category of other.

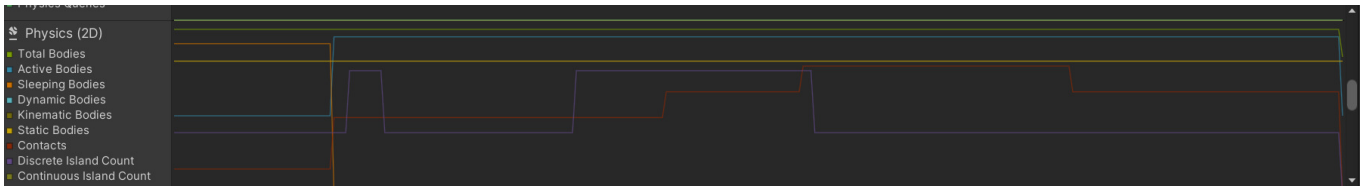

Figure 38:Phyiscs 2D performance test

The aspect of the physics in 2D showed that when a collision is made, the active bodies aspects increase, as the collision detection is dynamic, this is to be expected. Sleeping bodies happen right before the collision occurs, this means that any object that is not moving, is classed as a sleeping object. When a collision occurs, the objects are no longer sleeping.

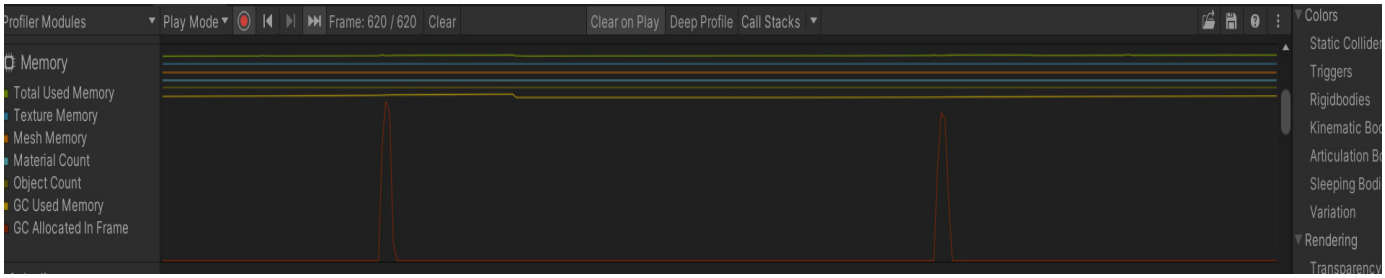

Figure 39: Memory Test

In relation to memory, there is not a huge spike for the memory used overall. The only spike that occurs during the single collision is GC allocated in Frame. GC (Garbage Collection) allocation in frame refers to how much managed memory is used during runtime. The spike only happens when the player is knocked back.

Another test was also conducted into the correctness of the collision detection. These tests are carried out manually to examine the events where the player collides with an obstacle. The importance of the test is to determine the success or failure of the collisions and make note on if more refinement is needed. There are three cases that are standard within the gameplay.

Test 1

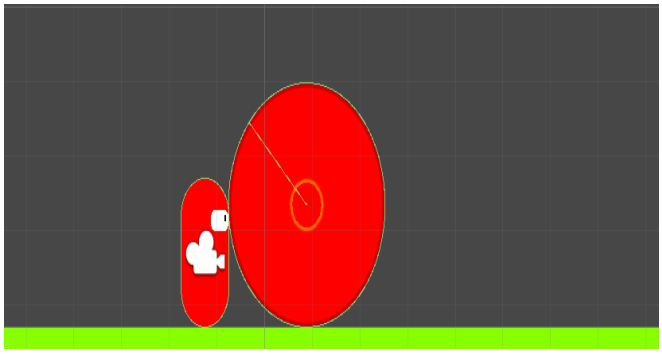

Figure 410: Frame 1 of test 1

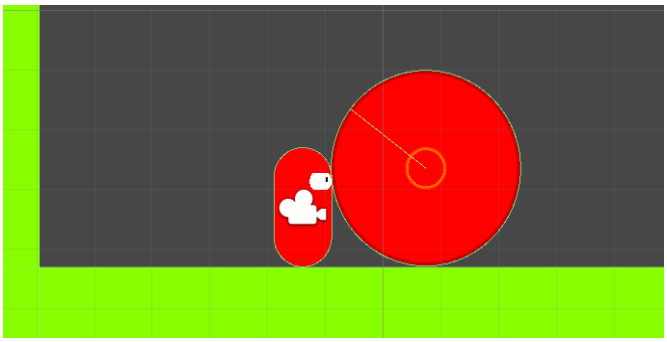

Figure 401: Frame 2 of Test 1

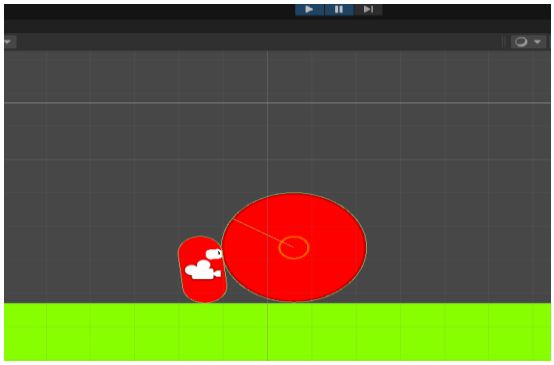

Figure 432: Frame 3 of Test 1

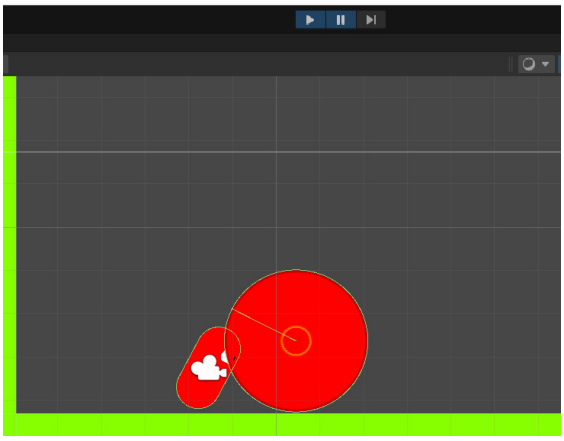

Figure 423: Frame 4 of Test 1

This test focuses when both objects are grounded and the angles are the same. Each image reflects a frame that is recorded. The first frame shows an intersection with both objects edge present. The collider lines are the same as the edge in order to provide a tight fit around the object. The second frame shows the player is moved back slightly from the boulder where third image shows the player's rotation is slightly altered. This is where the knockback effect takes place. The final image however does show a different result. The player is shown to phase into the boulder, this is due to the knockback code where the constraints of rotation is removed. To determine a realistic simulation, alterations to the boulder and the player's knockback code must be made.

Test 2

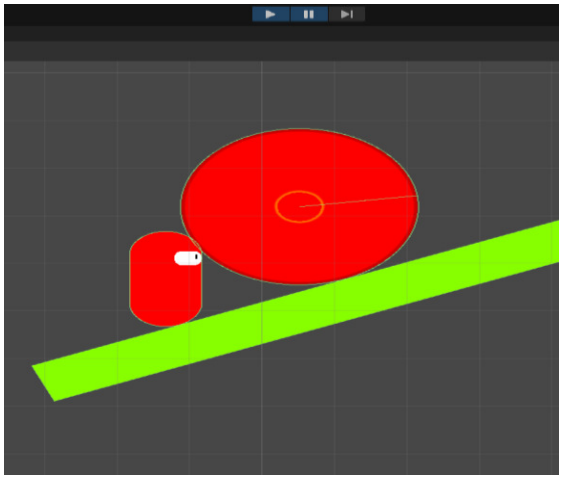

Figure 44 Frame 1 of Test 2

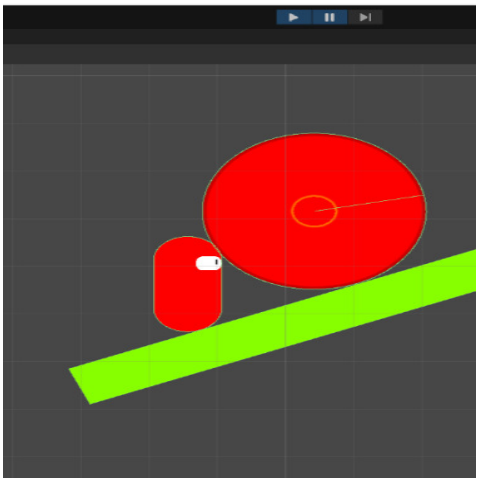

Figure 46: Frame 2 of Test 2

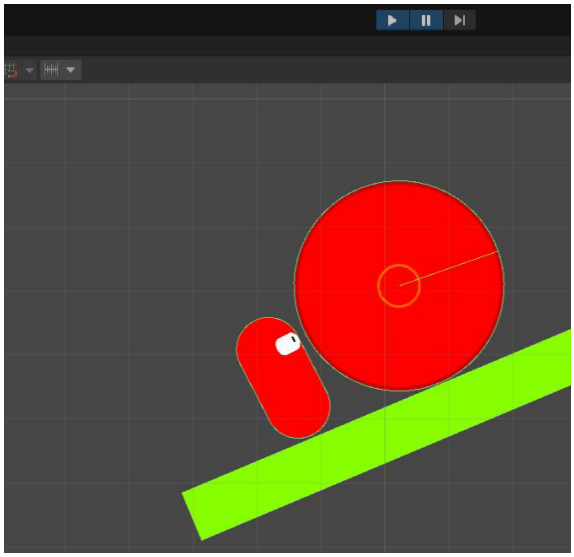

Figure 45: Frame 3 of Test 3

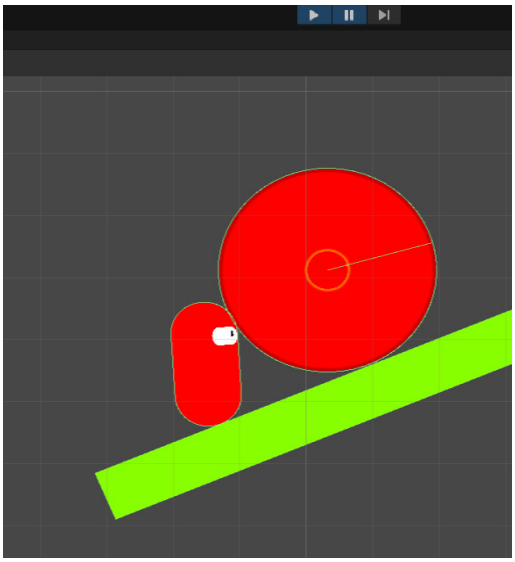

Figure 47: Frame 4 of Test 2

The second test involves a scenario where the player will move up the slope where a boulder moves down. This is the most common scenario within the level developed. As in the first test, the first image shows the objects colliding, the second shows the player moved back by the boulder. The third object shows the knockback code taking effect, rotating the player slightly where the fourth image shows the player being moved back and rotating. This result is more realistic than the first test.

Test 3

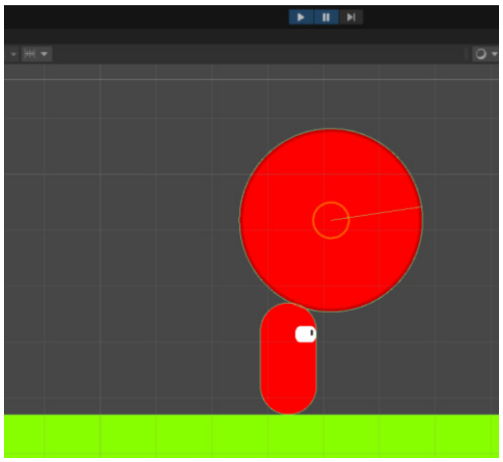

Figure 498: Frame 1 of Test 3

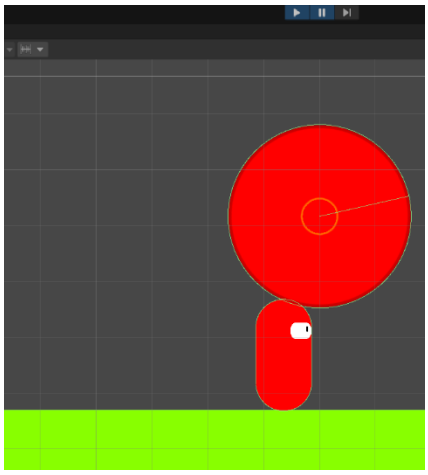

Figure 489: Frame 2 of Test 3

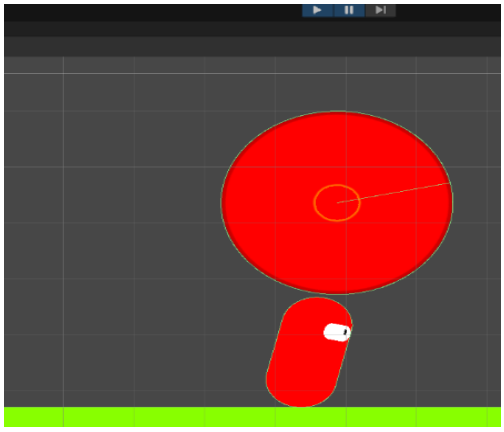

Figure 50: Frame 3 of Test 3

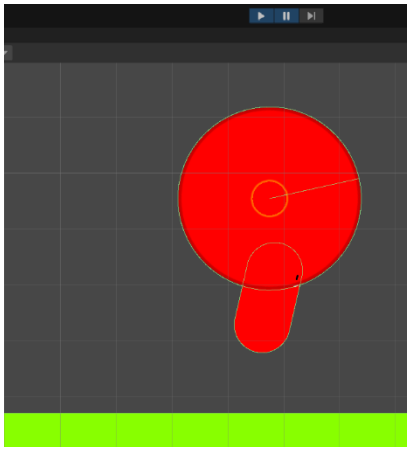

Figure 51: Frame 4 of Test 4

The third and final test shows the boulder colliding with the player from the top. This is one of the least common scenarios however it can occur. The first two images shows an intersection, the third shows the player rotating forward. The final image shows the player phasing into the boulder. This test failed in this regard as the player should not have phased through the object. With all of these tests concluded, more work need to be focused on the knockback as tests 1 and 3 shows that when rotating, the player can phase through the obstacles. Both the code and the prefabs of the obstacles needs to be changed in order for the correctness of the collision detection to be accurate.

Gameplay

Gameplay			
Test No	Aim	Success?	Notes
1	The boulder will be created every 6 seconds	No	The approach of not using a clone of the prefab will cause data loss where it can delete the prefab itself. Error – "Destroying Assets is not Permitted to Avoid Data Loss"
2	The boulder will be created every 6 seconds	Yes	The boulder is created and destroyed within the time window specified. The approach had to be altered due to the failure of the previous test. Clones of the prefab was required.
3	When the player passes the checkpoint, the position of when spawning is saved	Yes	The position is saved correctly.
4	When the player respawns, it respawns at the last checkpoint	Yes	Functions as expected
5	On pressing "A" to move left, the player's rotation will change,	Yes	Functions as expected

	and the player will move left.		
6	On pressing “D” to move right, the player’s rotation will change, and the player will move right.	Yes	Functions as expected
7	On pressing the space bar, the player will jump.	Yes	Functions as expected

Figure 52: Table of Gamplay Tests

UI

The user Interface is tested based on if the UI components such as buttons were functioning as well as scenes that are loaded in the executable file.

Main Menu			
Test No	Aim	Success?	Notes
1	On Level select click, the level select components must be loaded.	Yes	Everything is displayed as expected
2	When the quit button is clicked, the application must be closed down	Yes	The Application is closed
3	On Snowy Mountain button click, the level must be loaded	Yes	The level is loaded, and the player can play the level.
4	On pressing the back button, the Level select components must be hidden and the main menu components were shown	Yes	The level select components were successfully hidden and the main menu components were shown

Figure 53: Table of Main Menu Tests

The tests of the main menu scene and scripts ran without issue, most of the code is mainly hiding and showing components of the UI where the only code that is different is when loading the level and quitting the application when running the executable file. This stage is very straightforward due to the nature of the main menu screen.

Pause Menu			
Test No	Aim	Success?	Notes
1	On clicking escape button, the pause menu should appear.	Yes	Everything is displayed as expected
2	When the quit button is clicked, the gameplay is frozen	Yes	The gameplay is frozen as expected
3	Upon clicking resume button, the pause menu should be hidden	Yes	The pause menu is disappeared as expected.
4	Upon clicking resume button, the gameplay will be unfrozen	Yes	The gameplay is unfrozen, and the player can continue as expected.

Figure 54: Table of Pause Menu Tests

The pause menu in this scenario combines elements of gameplay as well as user interface. When the escape button is pressed, the gameplay using the time element is frozen, with the UI appearing. Although there is a resume button, it can be stated that a quit button should also be present as the user may wish to quit the level and select another one. This requires a little more time than implementing the main menu as gameplay is involved and originally was not freezing all movement within the scene.

Level Finished Menu			
Test No	Aim	Success?	Notes
1	When the player reaches the finish line, the level finished menu should appear	Yes	Collision with the finish line was successful and everything is displayed as expected
2	When the main menu button is clicked, the scene changes to the main menu	Yes	Button click is functional and the scene is changed to main menu,

Figure 55: Table of Level Completion Tests

The level finished menu depends on whether the player has reached the finish line in order for the scene change to take place. If the code of colliding with the finish line is functional, the finished level screen will appear. The only operational purpose of this UI is to take the player back to the main menu. If the game was extended in future work, this will include more details such as a leader board if more players are playing.

There is an issue that will need to be resolved. The player can jump multiple times which breaks the game as most games provide only one jump except for a double jump in cases where skills can be upgraded. There needs to be a limit as to how many times the player can jump.

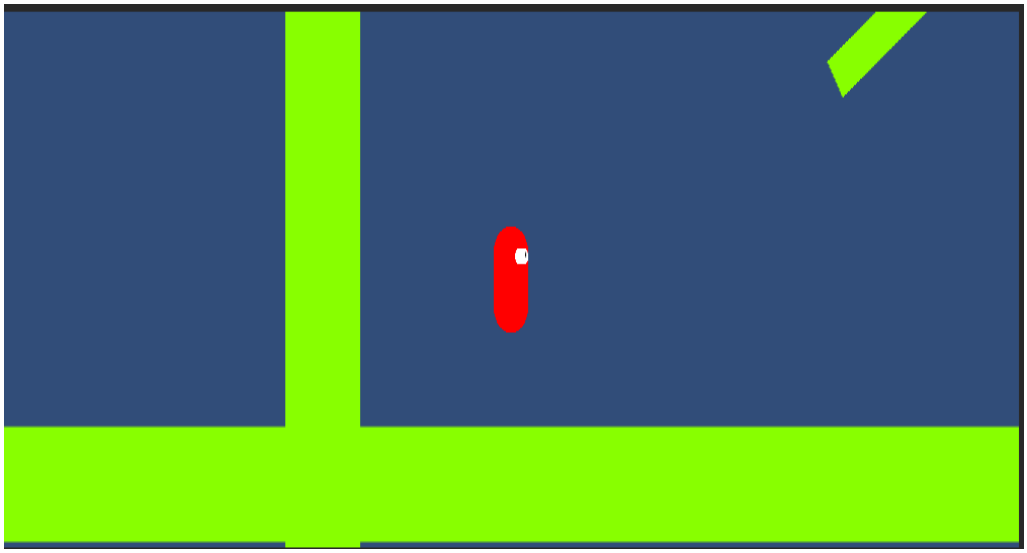

Figure 56: Player Jumping Once

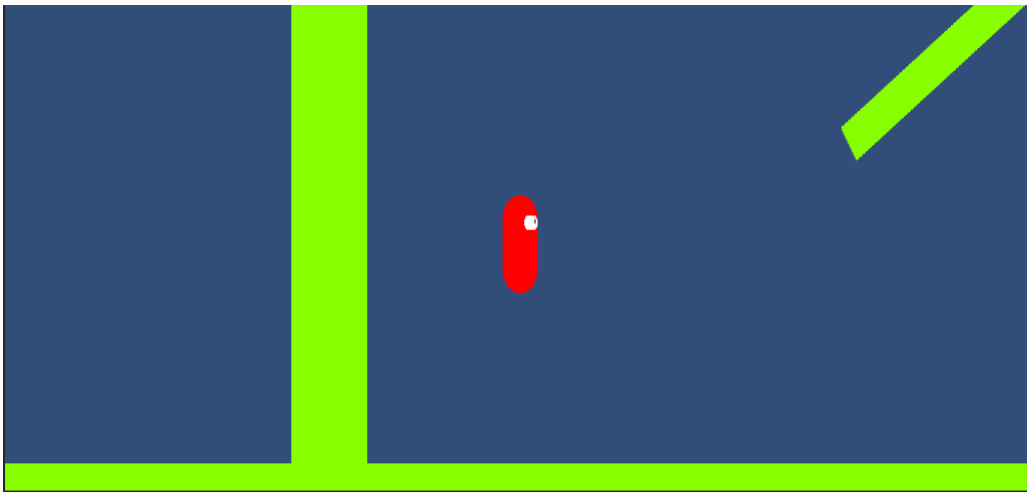

Figure 57: Player Jumping Twice

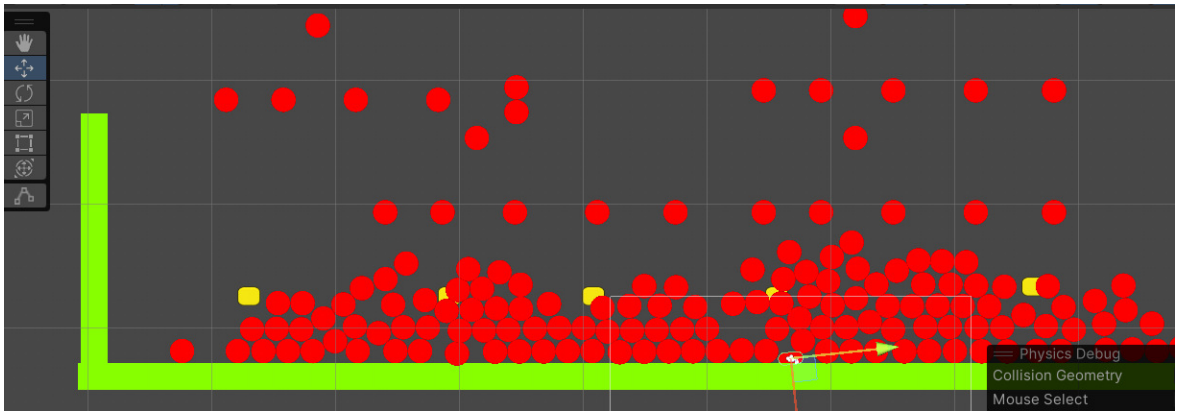

Figure 58: Collision Detection Simulation of 100 boulders

When testing how many collisions can occur at a time, a separate scene was created. This allowed the player object to collide with 100 boulders. Due to the hardware used for development, there is no lag in performance.

Chapter 6 – Future Work

With the evaluation completed and the issues examined, there are several improvements that can be made to further enhance the prototype as well as extending it further to include more replay ability. In this section, issues cited within the evaluation can be addressed and a solution found and furthermore, other components that can be added to the game can be identified.

There were a few aspects of gameplay that could be improved upon for it to be refined. The first is the use of a single jump, as with the current version of the prototype, the player can make multiple jumps which breaks the game as the player can jump their way to the finish line. If this was released as a game, it would not work. The second aspect is a more realistic knockback physics when a collision between a player and an obstacle intersects.

With regard to the collision detection aspect of the game, the scripts use the function of the physics 2D engine of Unity. Due to the level design and the way the gameplay is implemented, it can handle collisions of more than 100 obstacles at a time, however, if the game was to receive more development time and is extended, then more optimal solutions are required. This can be ranged from quadtrees and grid-based detection systems to speed up the process and reduce computation time. In relation to these solutions stated, a substantial amount of time was placed into implementing the quadtree and grid-based detection systems. First, the quadtree can be implemented in languages such as C++ where many examples of it have been published. In the aspect of Unity, very few examples were shown, some were over-engineered and become libraries that can be installed and used which is not suitable for this project.

When trying to create a simple and straightforward version of a quadtree in the interest of this prototype, the implementation failed due to missing components that were required for the script to function as expected. Secondly, another method was a grid-based detection system, one that is used in strategy games such as Civilisation. This requires the use of a grid and tiles, where the tile contains a box collider that can be used to determine if a player or obstacle collides, meaning the game object is in the tile. The data structure used for creating tiles is used in the form of a dictionary that takes two parameters, the object of the script and a 2D vector of where the tile is generated. In the aspect of the tile script, it requires a data structure purely for game objects that contains a tag 'Obstacles'. A hash set is created for this, used primarily in C# .NET framework, it functions in the same way as a dictionary but with a much faster performance. However due to the time constraints of this project, the remainder of the implementation was not completed. The project will be worked on past the deadline in order to learn from the shortcomings to implement a better physics detection system.

In terms of extending the project, once a more suitable collision detection system can be developed to handle more collisions during gameplay, more AI players can be implemented to give an element of challenge to the player for finishing first. This means that pathfinding algorithms such as A*, a modern version of the original Dijkstra algorithm will have to be implemented. AI can be used if the player plays in solo mode. The second extension is the use of a multiplayer component where, for example, LAN network can be used to connect players locally to a game that can consist of more than one level. This requires further research into developing servers and clients in Unity.

Chapter 7 – Conclusion

With the conclusion of the evaluation, there are a number of key findings that need to be explored. In this section, findings from the conclusion are presented in brief detail.

Firstly, the prototype was implemented that contains the specified requirements from the problem formulation. The results from the evaluation show that the physics implemented represent a simple object intersection which is beneficial for this stage of development, however it requires a fully developed collision system in the form of a grid. The user interface is fully operational and can be used at ease by the user to transition from the panels to gameplay smoothly. The level that contains the obstacles specified which test the physics aspect is developed and playable. Overall, the prototype *is* playable and fully functioning with grounds for further improvement to enhance the gameplay beyond this project.

Chapter 8 – Reflection of learning

Throughout the duration of completing this project, the author learnt a multitude of things. From researching the current literature to developing the prototype itself, there are many lessons to learn from the process. In this section, a number of areas are explored in relation to reflecting upon the work to achieve the aim and objectives of this project.

From conducting the literature review, the author learnt that there were very few papers that examined both the advantages and disadvantages of each algorithm in relation to spatial partitioning. There were also very few papers talking about how spatial partitioning can be used in collision detection. In the last five years, there have been very few literature resources that have been produced which are relevant to this study and currently there is still a lack of appropriate resources in the form of conference papers, textbooks and peer-reviewed journals.

During the implementation of the prototype, research was carried out as to how Quadtrees, which was examined in the literature review, was implemented within the Unity engine. During the research process, there were very few accessible examples of quadtrees being implemented. Although the concepts and ideas of how it can be constructed were noted, there were very few projects that showed how it worked. However, there were some that were developed, in a form of its own library that would not suit the project due to its over engineered aspects.

The final point that was learnt from this process was how to approach aspects of the implementation phase when it does not go well. For example, with a month of development, productivity must not be slowed down when there are other aspects of the prototype to work on. During these phases, morale can become low and therefore this needs to be avoided and methods need to be introduced to rectify this. All of the issues that have been identified will be acknowledged and improved upon.

Chapter 9: References

[1] Q. L. L. V. P. J. S. J. C. Louis Montaut, "Collision Detection Accelerated: An Optimization Perspective," 19 May 2022. [Online]. Available: https://arxiv.org/abs/2205.09663. [Accessed 24 08 2022].

[2] M. J. Martin Pichlmair, "Designing Game Feel: A Survey," *IEEE TRANSACTIONS ON GAMES,* vol. 14, no. 2, pp. 138 - 152, 2022.

[3] A. T. Randeep Kaur Kahlon, "QuadTree Visualizer," *International Journal of Engineering Research & Technology (IJERT),* vol. 11, no. 4, pp. 295-301, 2022.

[4] R. A. F. a. J. L. Bentley, "Quad Trees A Data Structure For Retrieval On Composite Keys," *Acta Informatica ,* vol. 4, pp. 1 - 9, 1974.

[5] D. J. R. Meagher, "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer," Image Processing Laboratory, Valencia, 1980.

[6] P. K. J. J. Z. Naimin Koh, "Truncated octree and its applications," *The Visual Computer,* vol. 38, p. 1167–1179, 2022.

[7] D. Meagher, "Geometric Modelling Using Octree Encoding," *Computer Graphics and Image processing,* vol. 19, no. 2, pp. 129 - 147, 1982.

[8] Y. Z. G. G. Q. J. Y. W. L. H. Wei Wang, "A Hybrid Spatial Indexing Structure of Massive Point Cloud Based on Octree and 3D R*-Tree," *Applied Sciences,* vol. 11, no. 20, pp. 1 - 16, 2021.

[9] J. L. Betley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM,* vol. 18, no. 9, pp. 509 - 517, 1975.

[10] D. W. Jia ZHOU, "Research on Ray Tracing Algorithm and Acceleration Techniques using KD-Tree," in *2021 IEEE 6th International Conference on Intelligent Computing and Signal Processing (ICSP 2021)*, Xi'an, China, 2021.

[11] B. M. G. W. S. R. Schumacher, Study For Applying Computer-Generated Images to Visual Simulation, Virginia: Defense Technical Information Center, 1969.

[12] F. Sanglard, Game Engine Black Book: Doom, California: CreateSpace Independent Publishing Platform, 2018.

[13] N. G. Y. T. X. Z. Yansen Su, “A Non-revisiting genetic algorithm based on a novel binary space parition tree,” *Information Sciences,* vol. 512, pp. 661 - 674, 2020.

[14] Á. A. F. R. F. Francisco Javier Melero, “Fast collision detection between high resolution polygonal models,” *Computers & Graphics,* vol. 83, pp. 97-106, 2019.

[15] Q. D. Baiqiang Gan, “An improved optimal algorithm for collision detection of hybrid hierarchical bounding box,” 01 February 2021. [Online]. Available: https://link.springer.com/article/10.1007/s12065-020-00559-6. [Accessed 2022 08 24].

[16] G. F. Geng Chaoyang, “An Improved Algorithm of the Collision Detection Based on OBB,” in *2018 International Conference on Sensor Network and Computer Engineering (ICSNCE 2018)*, Xi’an, 2018.

[17] S. K. S. N. P. Tejas Bhosale, “2D Platformer Game In Unity Engine,” *International Research Journal of Engineering and Technology (IRJET),* vol. 5, no. 4, pp. 3021 - 3024, 2018.

[18] M. E. Ramiz Salama, “Basic elements and characteristics of game engine,” *Global Journal of Computer Sciences: Theory and Research,* vol. 8, no. 3, pp. 126 - 131, 2018.

[19] Epic Games, “Blueprints Visual Scripting,” Epic Games, [Online]. Available: https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/. [Accessed 24 August 2022].

[20] H. C. D. C. A. D. V. J. Chaitya Vohera, “Game Engine Architecture and Comparative Study of Different Game Engines,” in *International Conference on Computing and Networking Technology (ICCNT)*, Khargpur, 2021.

[21] S. Y. A. Karzan Hussein Sharif, “Game Engines Evaluation for Serious Game Development in Education,” in *2021 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Hvar, 2021.

[22] S. X. Eleftheria Christopoulou, “Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices,” *International Journal of Serious Games,* vol. 4, no. 4, pp. 21 - 35, 2017.

Chapter 8: Appendices