Coursework

**By FInlay Nugent**

Contents

**1 - Analysis**

[The problem 3](#_Toc61031178)

[Computational Solution 3](#_Toc61031179)

[Stakeholders 3](#_Toc61031180)

[Analysis of existing solutions 3](#_Toc61031181)

[Data Collection 8](#_Toc61031182)

[Data Analysis 9](#_Toc61031183)

[Success Criteria, Limitations, and Hardware Requirements 15](#_Toc61031184)

[Choice of language 16](#_Toc61031185)

**2 - Design**

[Task breakdown 17](#_Toc61031186)

[Task In Detail 18](#_Toc61031187)

[Introduction to Algorithmic Solution 24](#_Toc61031188)

[Algorithmic solution 27](#_Toc61031189)

[Testing Solution 40](#_Toc61031190)

[Post-Testing Data 45](#_Toc61031191)

**3 – Development**

[Introduction To Development Stage 46](#_Toc61031192)

[Understanding Basics 46](#_Toc61031193)

Analysis

# The problem

Hong Ni, my client, has enjoyed playing first-person competitive shooters since before he can remember. However, he has recently come across a problem. Due to recent technological advancements, the latest first-person shooters have become increasingly more difficult to play on his hardware. He also desires a game where there is a certain level of skill required, and a tiered game design, which would allow the player to truly test their capabilities in a fun yet interesting first-person shooter environment.

The game must have the following: a realistic and satisfying weapon mechanics system, an interesting not massively hardware-intensive graphics layout, and fluid movement systems.

# Computational Solution

Typically, we have only ever seen first-person shooters on computers, or consoles. Therefore, we can see that a computational solution is the best, if not the only one.

The simulation of accurate physics, in terms of movement of the player and the bullets/projectiles, will be efficiently executed with computational methods, especially with “Physic Materials” built into Unity. Doing these computations in any other way would be excessively time consuming. Movement of various game objects, including the player, enemy AI, bosses, and other sprites/objects will be done with other in-built methods in the unity editor.

Keeping track of simple things, such as a score of points that you achieve, will be made very easy to with variables in the code, which can also be easily altered at any time.

There will be several different maps that the player can choose from to play in, and the use of the tools Progrids and ProBuilder, creating entertaining environments will be quick, and painless.

# Stakeholders

There are many stakeholders in this project. Hong Ni will benefit greatly from this project, as his hardware cannot run other, more graphically intensive games that are available.

Besides Mr. Ni, there are several other people who are interested in this project. For instance, several of my friends have been looking for something similar on the market, yet there seems to be a lack of simple first-person shooters. Lots of my friends do enjoy some shooters on the market, however for many they are just simply too much money. This project will be totally open-source and free of charge, so anyone can play it.

# Analysis of existing solutions

This genre of video games has a panoply of already existing solutions, many of which have been massively successful in sales. In order to fully understand the best course of action to produce a viable alternative solution, many other solutions need to be analysed. It is essential that the parts of what makes other shooter games good are used in good taste, and all negative parts be avoided or solved in this project to make a successful product.

**Solution 1: Call of Duty: Modern Warfare**

The first solution that came to mind was the ever-famous series: Call of duty[[1]](#footnote-2). This has been, for many, the first shooter that they have played, so there must be several aspects of the game design which allow the game to be easily accessible, and fun to play.



Courtesy of: “No commentary gaming” <https://www.youtube.com/channel/UCrdI6AcVLxLC7ZFJr19dlDA>

This game has a clear and simple layout, the UI elements are quite minimalist, and the overall scene is small, and not too grand. The pace of the game is quite fast, which allows the user to be entertained. The ability for the user to choose the level of detail the scenery and the weapons are in is absolutely to this game’s advantage; the hardware requirements for this game are not massive, and even relatively old systems can run this game reliably. To quote[[2]](#footnote-3):

*“The minimum requirements are as follows:*

* *OS: Windows 7 64-bit (SP1) or Windows 10 64-bit*
* *Processor: Intel Core i3-4340 or AMD FX-6300*
* *Memory: 8GB*
* *Graphics Card: Nvidia GeForce GTX 670/Nvidia GeForce GTX 1650 or AMD Radeon HD 7950 - DirectX 12.0 compatible system*
* *Storage: 175GB”*

As we can see, the OS that is needed is not a restrictive factor for the user: windows 7 and windows 10 are two of the most widely used operating systems in the world. As for the processor, graphics card, and storage, these are not particularly restrictive either. This will be a very important factor in the design of this project, so the ability to run on relatively old hardware is very important.



Courtesy of: “gamesradar” <https://www.gamesradar.com/call-of-duty-modern-warfare-multiplayer-review/>

In terms of the computational approach, this game is using variables to store a sort of score system, denoted by the +100 on the screen. This will be something that will be replicated in this project, as it can be very rewarding to the player, when playing the game.

This can further be built on by adding leaderboards or upgrades, which Call of Duty also has. The ability to achieve upgrades in this project will also be available, which will hopefully give the user a feeling of progressing, which is often what people enjoy most in shooter games. The use of post-processing effects in this game are used to great effect: they really add a lot to the environment, and make it feel less sterile. Unity has several in-built post-processing effects that will be used to make this project feel more realistic, and potentially make it more pleasing to the eye.

Many mechanical aspects in this game are inspiring, the smooth movement, the fluid animations, the satisfying weapon mechanics. All of these will be seen in this project, all of which will hopefully add up to an enjoyable game.

There are a couple of issues with call of duty, however. Many people seem to not appreciate the sounds of the guns, calling them “tinny” or “boring”. I hope to remedy this in my project with the use of high-fidelity sounds, which hopefully will make the weapons seem more impactful. It is also important to note that this game is as AAA title and has had the work of many hundreds of people over more than a year put into it. So, the variety in the weapons and the multitude of maps will not be seen to such an extent in this project.

**Solution 2: Murder Miners**

Murder Miners[[3]](#footnote-4) is another solution. It is a multiplayer shooter, which primarily focuses on interesting gameplay mechanics and a variety of skill-based weapons. What this solution has over Call of Duty is that it is made by a relatively small company, so the final product will be more similar to this project. The look and feel of the game are quite unique, and while it doesn’t have the final AAA title polish like Call of Duty, it does have a characteristically simplistic design.

The landscapes are quite blocky, which means that they are quite easy to develop. This is one aspect I wish to implement in my game, as it will make world-building drastically simpler, while still being relatively visually pleasing.

Courtesy of: “Jforce Games” <https://jforcegames.com/>

The pace of the game, like many shooters, is quite fast. This will tie in very well with the fluid animations and smooth movement to make a game that is fun to play. A good movement system is vital to creating a good shooter game.

In terms of graphics, the game seems quite basic. This is because the game design team is not massive, so they did not have a lot of time to create extremely high-detail graphics. However, even though they may seem a little bit dated, the graphics do serve the game well.

Also, there is a large selection of maps in the game. In this project there will be several maps to choose from as well, as the ability to select a map to play on adds a lot of interest into the game, instead of the game becoming very repetitive in the same environment. However, the actual themes of the maps in Murder Miners can be quite varied, some being large city-landscapes, and others being alien planets. In this project, however, I intend on making all the maps to a similar sort of theme, which will make the game seem consistent.

The creators of this game, however, used their own in-house game engine to create this game. While this means that they had a greater ability to customize their game to make it as unique and interesting as possible, creating an entire game engine is very time-consuming, and will not be done for this project due to time constraints. The abilities of Unity are vast, and certainly contain everything that would be needed to make this project.

In terms of system requirements, this game is quite unique. Due to the lack of extremely detailed graphics and lightweight game engine, the requirements are very relaxed. From the Steam link previously referenced, it says:

*“The minimum requirements are as follows:*

* *OS: Windows*
* *Network: Broadband Internet connection*
* *Memory: 1 GB RAM*
* *Graphics Card: ATI Radeon X1300-X1950, nVidia Geforce 6 et 7*
* *Storage: 250MB required”*

These requirements are very low, and as the game came out in July 2014, even low performance machines from 6 years ago would still contend quite well with this game. This basically means if the computer has a graphics card it will be able to run this game. For this project, it is essential that even low performance systems can run it. With the use of simple level design and the not extremely high-quality graphics that this game has, it is likely the project will run very well.

**Solution 3: Hotdogs, Horseshoes and Hand grenades (VR)**

H3VR[[4]](#footnote-5) is a game that is supposed to model guns as accurately as possible. Every single bullet, magazine, and attachment are faithful to their real-life counterparts to an extremely high degree of accuracy. This game has been my main inspiration to create a first-person shooter, as I have played this game for a long time.

The one difference between H3VR and this project is the fact that H3VR is in VR. I am not intending to build this project in VR, as VR development is difficult and requires the user to have an expensive headset. On top of that, in order to use a VR headset effectively, you need a relatively high-end machine. I am taking massive inspiration from the game in this project, as I have gotten many hours of enjoyment out of it, although I won’t be using VR in the project.

In terms of the core game features, this game seems to do everything right. The movement is very fluid, the weapons mechanics are basically a recreation of real life, and the graphics are quite satisfying. However, most of the game's mechanics are built for VR, and the fact of the matter is that those mechanics would likely not port well over into normal, flat screen games. So instead of completely copying everything about the game, the only inspirations will be the weapon mechanics and the overall design and layout of the game.



As shown in this picture, the style of the graphics is quite simple, like Murder Miners. I think for a small game developer, the best course of action is to use simple graphics like these.

H3VR was created in Unity as well, so it really showcases that such a solution can be executed to a high degree with the game engine.

Also, the detail to which the guns are actually modelled in this game is not going to be feasible in this project, due to the complexity and the fact that VR allows for a greater level of control/manipulation of handheld objects than a mouse and keyboard. Therefore, the weapon mechanics that we will see in the project will be quite simplified.

# Data Collection

In order to create the best solution possible with the given amount of time, it is important that I find out exactly what needs to be in the solution. There will 2 methods that I will use to collect the data that I need:

**Questionnaire**

This is going to be a small set of important questions based on what the solution will contain and what the stakeholders want from the solution. I will use an online resource to create a questionnaire that is easy to distribute to people, which will allow for a greater sample size, and then hopefully a better picture for what needs to be done for the solution. A common problem with questionnaires is that the fixed answers can be misleading, or not allow the user to give their whole opinion on the matter. To combat this, I will allow for a small box where the people doing the questionnaires can put any further comments they may wish to make.

I will be mainly intending to send this questionnaire to people whom I know as interested in the final project, who may want to play the game, or who are interested in games in general. Once I have received an adequate amount of responses, I will analyse the data, and determine what the majority of the respondents want.

**Interview with the client**

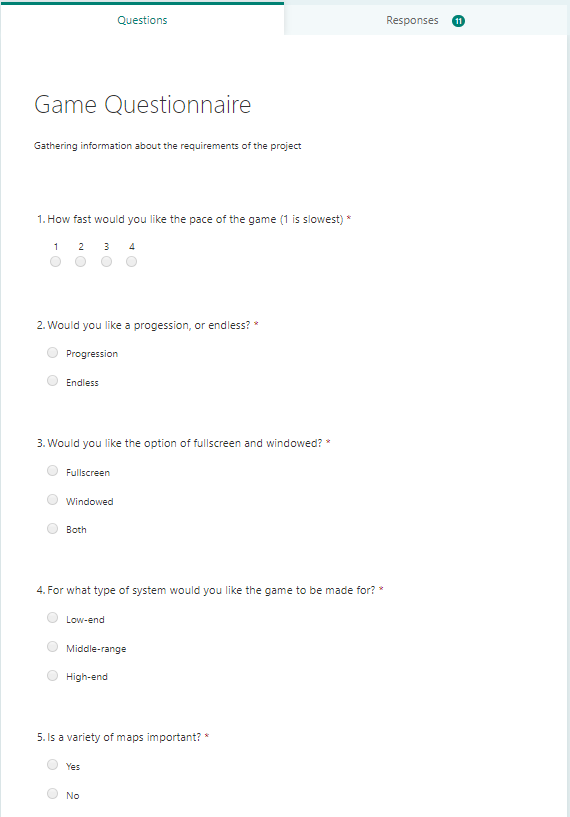
Of course, the main client has the opinion that matters the most in this situation. So therefore, I will take further measures than just a questionnaire to make sure he gets what he wants. This interview will comprise of me asking specific questions based on ideas that might be needed in the game, as well as some insight from the client on what his specific demands are.

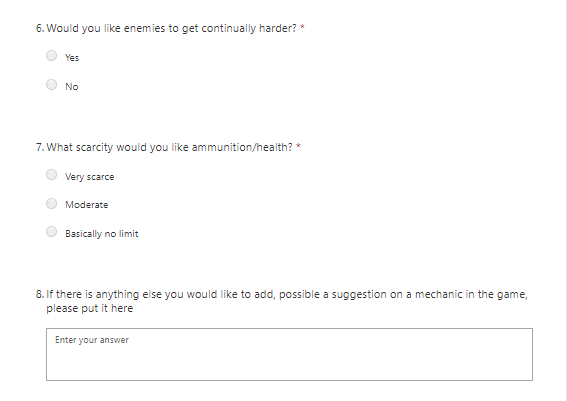
This will be beneficial over the questionnaire because it will let me see exactly what the client wants, and then I can give that opinion a bit more weighting over what is going to happen with the project. Due to it being more personal as well, I think I will be able to extract more information about what he wants for the game than with a questionnaire.

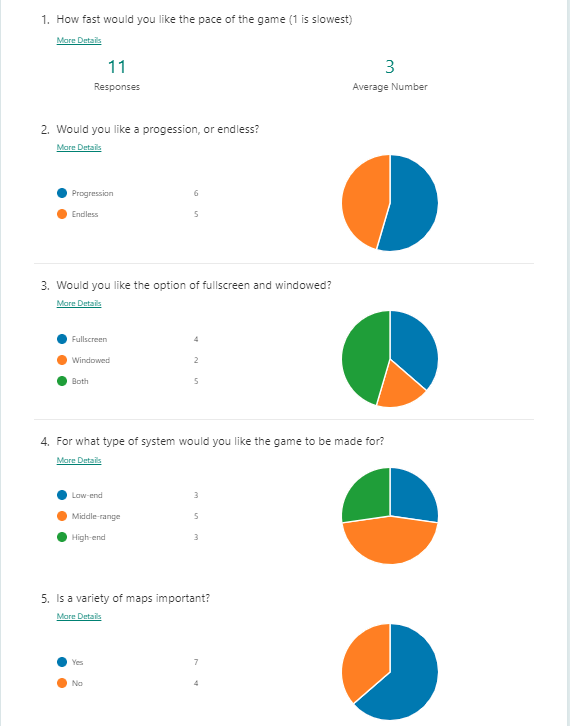
# Data Analysis

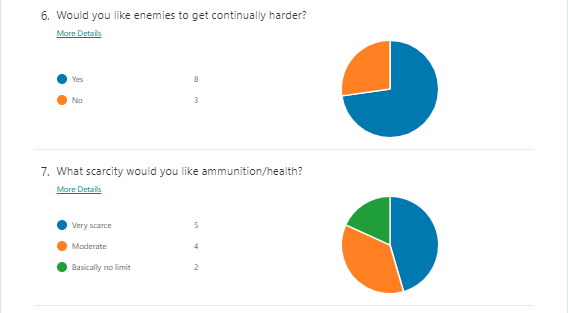
**Questionnaire Introduction**

I used Microsoft’s Forms to create an online questionnaire that was easy to distribute among people. It was very easy to setup and use, and I managed to create several different questions that are relevant to the project. I tried to only use questions that would benefit the final product and give a better picture of what the stakeholders want from the project.

*Here is the setup of the questionnaire:*



*Here are the results of the questionnaire:*



*Note: While there were several responses to the last question, none of them were particularly constructive or contained any useful ideas to be added into the game*

**Questionnaire Analysis**

Although there was some disagreement about some of the questions in the population, there do seem to be many trends in the data.

**Question 1**

The general result of this is question shows that people want the game to be relatively fast, but not extremely fast. I think this is more in line with a game like H3VR, which is relatively fast paced, but is not also excessively fast.

**Question 2**

For this question, there is nearly a 50-50 split between whether the game should have a system of progression, which I find quite strange. For me, a game needs some level of progression to be able to be interesting. As we will see later, the main client Mr. Ni agrees with me in that respect.

**Question 3**

This question was about the user’s preference for a windowed and full screen option. While many voted for just full screen, a lot also voted for the option for both. As this is not particularly difficult to implement with the Unity engine, I have decided that it is best just to add the option anyway.

**Question 4**

When asked about how intensive they want the game to be, many people went for the mid to low range. This means that there is a considerable amount of people who would like an accessible solution to this problem, and therefore a simpler graphics/physics system in the game will be appreciated. While performance is key, quite a few people have mid-high range computers, so this does give us leeway in the amount of effects / other performance intensive aspects in the game.

**Question 5**

The answer to this question was also quite split. Many people would like some variety, while others aren’t massively worried about the types of maps there are available. I think that this means a few maps of similar style, as aforementioned, is the best course of action.

**Question 6**

The result for this question is clear: people want the challenge of the game to increase as it goes on.

**Question 7**

This question has shown that people want scarcity of things to be a factor in the game, rather than it being an infinite slaughter. This project will need fine tuning to make sure that the level of scarcity is bearable, and it needs to be certain that even if you completely run out of ammunition, there is still the ability to fight the enemies (possibly with a melee weapon).

**Summary**

So, in summary, I believe this questionnaire has revealed many things:

1.The game needs to be relatively fast

2. Some system of progression

3. Windowed and full screen options

4. The game should perform well for a low to medium performance computer

5. A few well engineered maps

6. Enemies get harder over time

7. Quite scarce resources

**Interview Introduction**

The purpose of the interview with the main client is to identify which parts of the game are more important than others. With the help of having a real time conversation with Mr. Ni, I will be able to identify which parts of the solution are required and which are optional. Below is a copy of the conversation I had with Mr. Ni:

|  |  |
| --- | --- |
|  | Started: 19:41 24th March 2020 |
| Me | Hello Hong! I was wondering if you could answer a few questions about the first-person shooter game that I am creating |
| Mr. Ni | Sure, what would you like to ask? |
| Me | I would just like to get to know more about what you want from the game. In terms of the questionnaire that I got you to do, can you remind me of what you answered for the questions? |
| Mr. Ni | ...[Reflected in the questionnaire summary] |
| Me | Ok, that’s good. So, in terms of the difficulty, how hard do you want the game to be? |
| Mr. Ni | I’m not sure… I would like a game that I can relax while playing, but at the same time I occasionally want a challenge when I play games. |
| Me | How about a system where you can change the difficulty of the game? |
| Mr. Ni | Yeah that would be good. I would like it to be quite a large area of difficulties, where the easiest is very easy, and the hardest is very hard [sic]. |
| Me | Sure. Also, I remember you telling me a while ago that your graphics card broke, would you like the game to be accommodating to that? |
| Mr. Ni | A few months ago my computer randomly started to crash out completely when playing intensive games. This was really frustrating, but it was also really annoying how the games weren’t running as well as they used to. So after this happened a few times, and after looking into the system specs and doing diagnostics, I came to the conclusion that my graphics card exploded. I’ve recently also become destitute as well, so I most likely won’t be able to buy another good graphics card for a while. |
| Me | Do you have another graphics card to use in the meantime? |
| Mr. Ni | Yeah, but it’s just a GTX 650, which is basically unusable with a lot of the games I used to play. So for me it’s pretty important that the game can be ran on really old hardware. |
| Me | In terms of the performance of the game, I think when using the Unity engine, making sure that the physics and graphics aren’t extremely demanding will be quite easy. I will make sure that it will run on your computer. |
| Mr. Ni | That would be great! |
| Me | I assume you have a little bit of space left on your hard drive though? |
| Mr. Ni | Oh yeah, as long as its not over 300 GB I will be able to keep it on my computer. |
| Me | It won’t be too large, most likely under 500 MB, so you should be fine. |
| Mr. Ni | Ok that’s fine, but what about RAM? Do you think the game will need a lot of a RAM? I’ve only got like 8 GB. |
| Me | Don’t worry about RAM, this project will be quite lightweight, so you probably won’t need any more than 2 GB to play it. |
| Mr. Ni | Nice. |
| Me | You also said that you would like some sort of progression, would you elaborate on what type of progression you would like to see? |
| Mr. Ni | Well, for me, I don’t need a huge amount of progression. As long as I can roughly tell the game is actually progression, it’s fine. I guess for ways of showing it, maybe add some different weapons. As you go along the game you can unlock these new weapons, which are a lot better than previous ones, and maybe at the end one really interesting weapon, like a laser gun or a rocket launcher or something! |
| Me | Ok, I appreciate your passion. Well I think that’s most of what I wanted to ask you really. Is there anything else you want to tell me before the end of the interview? |
| Mr. Ni | No not really. Thank you for making this game for me, I haven’t been able to play any first person shooters for months since my graphics card blew up. |
| Me | No problem, talk to you soon. |
| Mr. Ni | Bye! |
|  |  |

**Interview Analysis**

The interview didn’t add a massive amount to the list of things that need to be added, although it did add some extra bits of information that may be helpful, these include:

1. A way of changing the difficulty
2. Preferably quite small in terms of file size
3. A concrete progression system, with new guns being given to the player as they progress. Also, an ‘endgame’ sort of weapon like a laser gun/rocket gun right at the end of the game.

# Success Criteria, Limitations, and Hardware Requirements

**Criteria**

*Essential:*

1. *A progression in terms of difficulty*
2. *Small File sizes*
3. *Some sort of progression to better and better weapons*
4. *Simple graphics design*
5. *Interesting and fun weapon characteristics*
6. *A few good maps of similar design*
7. *The ability to change the difficulty on the enemies*
8. *Feeling of scarcity in resources*
9. *Ability to choose full screen/windowed*

*Desirable:*

1. *Ability to save the game*
2. *Items around the world to pick up and use*
3. *Ability to play at any resolution*

**System Requirements**

* OS: Windows (Unity currently only supports windows)
* Graphics card: GTX 650 or equivalent (To render graphics)
* Storage: Roughly 500 MB (To store game)
* Memory: At least 4 GB of RAM (To run game)
* CPU: Any i3 or i5 or equivalent

**Software Requirements**

There are no software requirements.

**Limitations**

There will likely be no multiplayer as multiplayer will be difficult to integrate into the game in the given amount of time, and multiplayer slightly defeats the purpose of the game as a simple shooter.

# Choice of language

**C# And Unity**

I have chosen Unity as the game engine of choice for my project. There are several reasons why I have chosen unity over other similar game engines. One of the main reasons is that I have a bit of experience with unity as a program, as I have spent quite a while getting to know the components of it. Also, Unity allows you to create detailed, 3D play spaces which may not be possible with other engines that mainly deal with 2D spaces.

In terms of language, I had few options. C sharp is a great language, however, it is by far the most widely used language with Unity, as most of the scripts you will make with be in the language. Also, I have had quite a bit of prior knowledge of the language, and often use it for other projects.

design and decomposition

# Task breakdown

Here is a summary of the tasks that will need to be completed in order to finish the project.

* **Options Menu**
  + **Video Settings** allow for tuning the games performance to the machine.
  + **Audio Settings** to tune the volume of the music/game to fit the user.
  + **Game Settings** to help choose difficulty, toggle off blood, etc.
  + **JSON Record** so that options are kept after game is closed.
* **Level Selection Menu** to let the user choose which level they want to play.
  + **Progression Mechanic** so on the first time, only the first level is available.
  + **Level Profiles** such as Icons/Names for ease of identification.
* **Player Movement**
  + **Intuitive movement system** using Unity’s CharacterController.
  + **Gravity** that acts in a realistic manner.
  + **Death Animation** to signal end of the game.
* **Weapon Mechanics**
  + **Reloading and Magazines** to add realism to the weapons.
  + **Shooting** which varies in properties based on type of weapon.
  + **Animations** for all mechanics that cannot be interrupted, using Unity’s Animation component.
  + **Bullet Mechanics**
    - **Blood** if desired by player (in game settings).
    - **Damage model** to manage health / dying of the player / enemies.
  + **Weapons** 
    - **Knife** for close quarters, and never goes away
    - **Pistol** weak starting weapon with plentiful ammo.
    - **Rifle** fully automatic end-game weapon with limited ammo.
    - **Shotgun** hard-hitting end-game weapon with high damage, low ammo.
* **Weapon UI**
  + **Active Weapon** to visually represent what weapon is active at a given time.
  + **Bullet Count** to see how many bullets are left in the magazine in the weapon.
  + **Magazine Count** shows the magazines left, represented by the number of bullets left in them.
* **Enemy AI**
  + **Search area** so the target can acquire the enemy.
  + **Movement methods** so the enemy can traverse to the player in a smart manner.
  + **Attack mode** for when the enemy gets within a certain range of the player.
  + **Patrol** so some of the enemies move about the map in an aleatoric manner.
* **Levels**
  + **Lighting** made for good performance, with a progressing colour scheme.
  + **Levels of Detail** to make the maps appear alive, and interesting.
  + **Common theme** to ground the game, and make a coherent progression.
* **Sound**
  + **Weapon Sounds** such as shooting, bullet impact, casing collision, reloading, drawing/holstering.
  + **Player Sounds** such as walking, running, jumping, dying, being hit.
  + **Music** that can be toggled separately for user preference (in settings).
* **Escape Menu** 
  + **Time Stopping** so that the game doesn’t carry on when it is active.
  + **Inbuilt Options Menu** to change the options mid-game.
  + **Ability to Exit to Menu** for ease of navigation of the game.

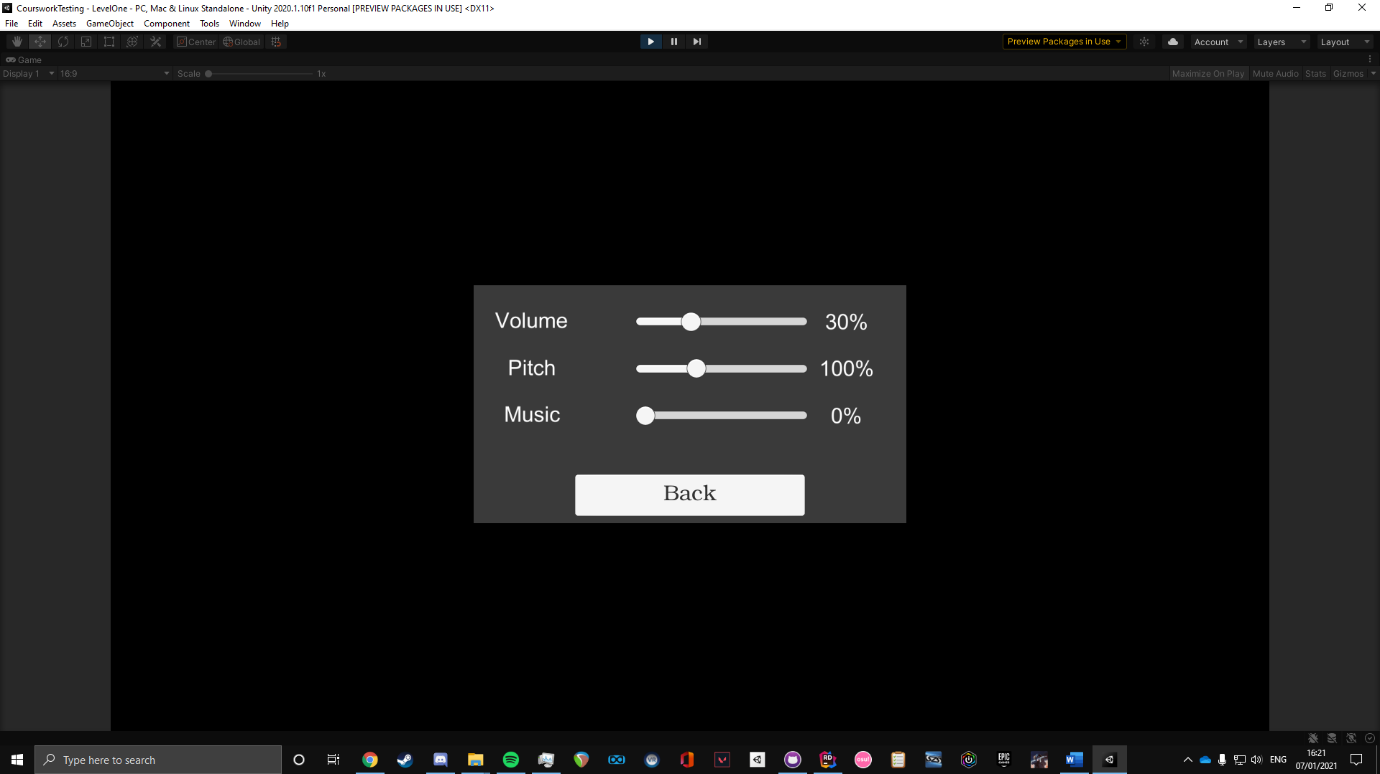
# Task In Detail

**Options Menu**

To create a menu that is comprehensive, multiple panels will need to be created. One for each of the sub-levels listed above. These will contain unique options in each, and along with a smooth way to move between them, will allow the player to easily understand how to optimize their game. I will be using the UI elements built-in to Unity, as they provide great features for creating UI, such as interactivity, and proper scaling to fit the screen. The options menu will consist of a main menu, which allows the user to choose between the setting sub-levels, as well as quit and start the game again.

For each sub-level, there will be a hidden panel, which appears once the button is clicked, and disappears when the user hits “back”. In addition, I will need to make it so that the game stops when the options menu is up, and the way this will be achieved is by setting the frame time to zero. However, this does not stop instantaneous non-physics calculations occurring (such as shooting a gun), so there will need to be code to further solidify this, which will essentially ignore all other inputs the user gives at the time.

In order for the buttons to function, there needs to be method in a script that is executed when they are pressed/changed. So, for instance, a slider to choose if there is blood or not, will need to execute a method that updates the GameSettingsProfile class that is in use by the game, so the rest of the program knows if there should be blood or not. GameSettingsProfile is a class that I created to be able to save settings in a JSON file format. Every time a sub-level is executed, the new settings need to be saved to the JSON file associated with that sub-level, which is also called through a function when the back button is pressed on each panel. For the video and audio settings, there will need to be specific AudioSettingsProfile and VideoSettingsProfile created so that the settings for those sub-levels can also be saved.



***This is the audio settings panel. Each slider is interactable and affects the respective audio element.***

**Level Selection Menu**

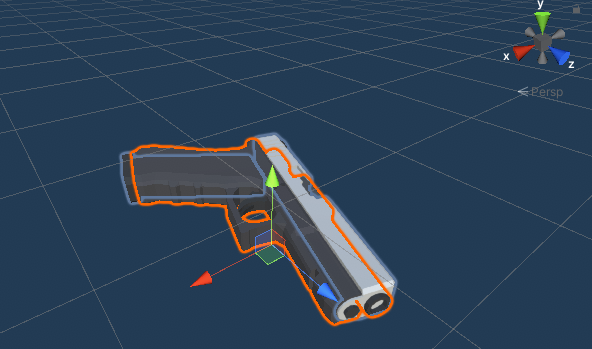
The level selection menu will contain a small animation, as it will be a new scene. This will end up in there being a selection menu appearing. It is important to the progression of the game that levels are locked based on what you have achieved. First, there will be a tutorial. This will be very useful to people who aren’t very familiar with video games, and will go through the basics of the game. Next, level one appears. A button in activated in the level selection screen that allows you to play this level. Once completed, it will activate the button for level two, and so on.

**Add picture of finished level Selection Room**

**Player Movement**

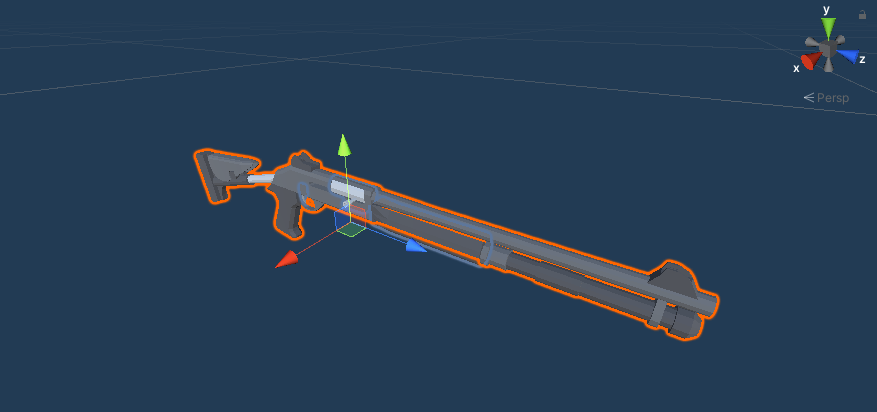
Player movement is made a lot easier with the use of Unity’s CharacterController component. This allows for a simplified version of movement, which doesn’t involve forces, inertia, drag, friction, or velocities. I will be using this to move the player, as I have done lots of work on using different methods of moving the player, such as using the RigidBody component. RigidBody movement is ultimately much more comprehensive, yet due to time limitations it was too difficult to get fully functional. The CharacterController is also used by the enemies for their movement, which will be detailed further under the enemy AI section. I have found through experimentation that the CharacterController is overall more fluid, and feels better for this type of game.

However, due to the fact that I wasn’t using the RigidBody, the player needed a source of gravity (as it is not implemented in the CharacterController). I will implement this via code on the player.

**Weapon Mechanics**

The weapon mechanics are arguably the most important part of this project, as it will likely be what people find the most fun part. This is where the bulk of the work is going to go. A system is used for the magazines, where each magazine is an object of type Magazine stored in a list for each weapon. These can be swapped, and reloaded by the player. This will be accompanied by an animation.

The animations will be created with use of the in-built Animator, which allows for simple animations to be created, and triggered through code. When R is pressed, a function which finds the largest magazine available is called, and when it finds it, it runs the animations while changing the current loaded mag, and places the old one back into the inventory.

4 weapons will be made available: Knife, Pistol, Rifle, Shotgun. The 3 guns will all derive from the same base class, called GenericWeaponBehaviour, which will contain most of the weapons implementation, such as Generating random bullet rotation, generating casings, generating bullets, triggering animations etc. The shotgun is a bit different in the fact that it is a shotgun, so instead of one precise bullet, it produces many smaller but less accurate bullets. This is built into the base class, and checks whether the weapon is a shotgun, and if so, produces the bullets in the manner expected.

The weapons will be using free assets take from the unity store. This fit well into the aesthetic of the game, as you can see from the pictures below.

The rifle will be similar in functionality to the pistol, except for the fact that it is fully automatic. This will be achieved using Input.GetKey instead of Input.GetKeyDown, which will allow the firing method to be done while holding down the shoot button.

As for the knife, this will not derive from the same base class as it is too different from the other weapons. Instead, a new script will be made for it where it handles the use of the knife, such as the animation of swinging, dealing damage, and checking if the player hit anything. It will cast a ray from the camera of the player outwards, checking if the player can see anything. If there is an enemy within a set distance, it will inflict damage upon the enemy. If there is nothing, no damage will occur and a sound that corresponds with that will play. If any piece of environment is close enough, no damage will happen, and a sound will play that represents hitting the environment (a clang).

**Weapon UI**

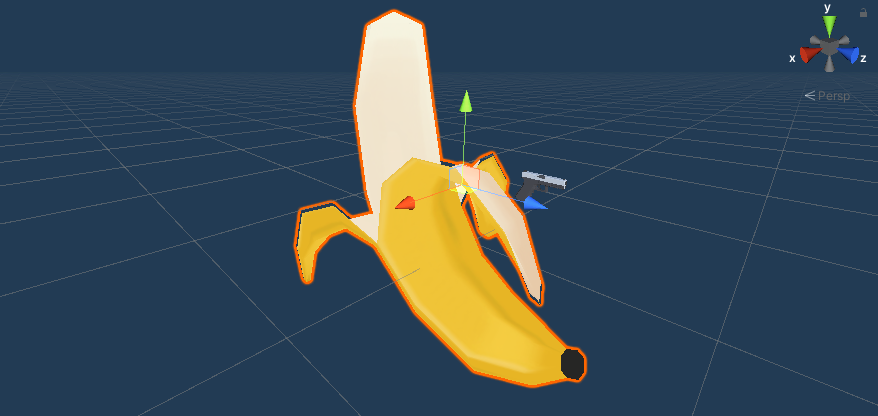
An important aspect of the game is playability; it is meant to be a relatively fast paced, single-player shooter. In order to increase the players’ knowledge of the game, an intuitive UI will be needed. During play time of the game, there will be two panels shown to the player. The first one will hold useful information about the weapon, the amount of ammunition in the current magazine, the ammunition in the other magazines, and the players current health. All of this functionality is contained within scripts of other GameObjects, such as the weapon the player is using, so no code is needed specifically on the UI.

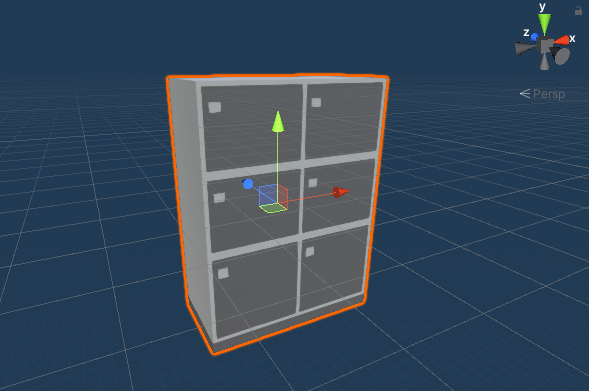
**Enemy AI**

In order to produce a solution to this problem that is fun to play, a good enemy AI system is essential. The way the enemy AI works is through the use of the in-built component called CharacterController, which is also what allows the player movement. There are many aspects about the enemy AI that will be needed. This includes the ability to search for the player. This will be achieved using code to see if the player is near the enemy, and within a certain field of view that the enemy has. When the player is detected, the enemy should act accordingly.

There should be a level of randomness with the way the enemies react when they see a player. First, they should not be able to immediately attack the enemy, there should be a sort of ‘reaction time’. This will be induced via a method where a random amount of time is waited before any more actions are taken by the enemy. Next, the enemy should start to shoot at the player, until the player is dead, or they have run out of ammo. A level of inaccuracy should be given to the enemy, so that they are not always hitting the player. This should be able to be modified through the game difficulty setting, which the player can change.

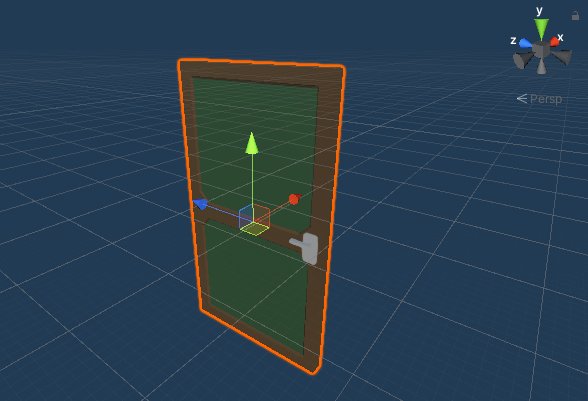
To create another layer of depth, it has been decided to allow some enemies to patrol. They have a set of a few waypoints that they can access and can move to. After a random amount of time, they will move to another one. They will disregard this behavior once they see the player, of course. However, not all of the enemies can behave in this manner, so there will be an ability to toggle this on and off for certain enemies.

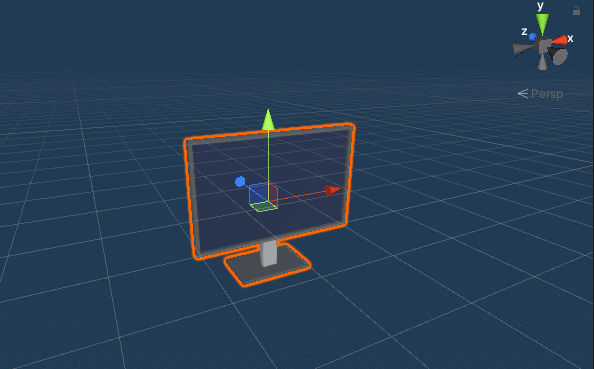
****



**Levels**

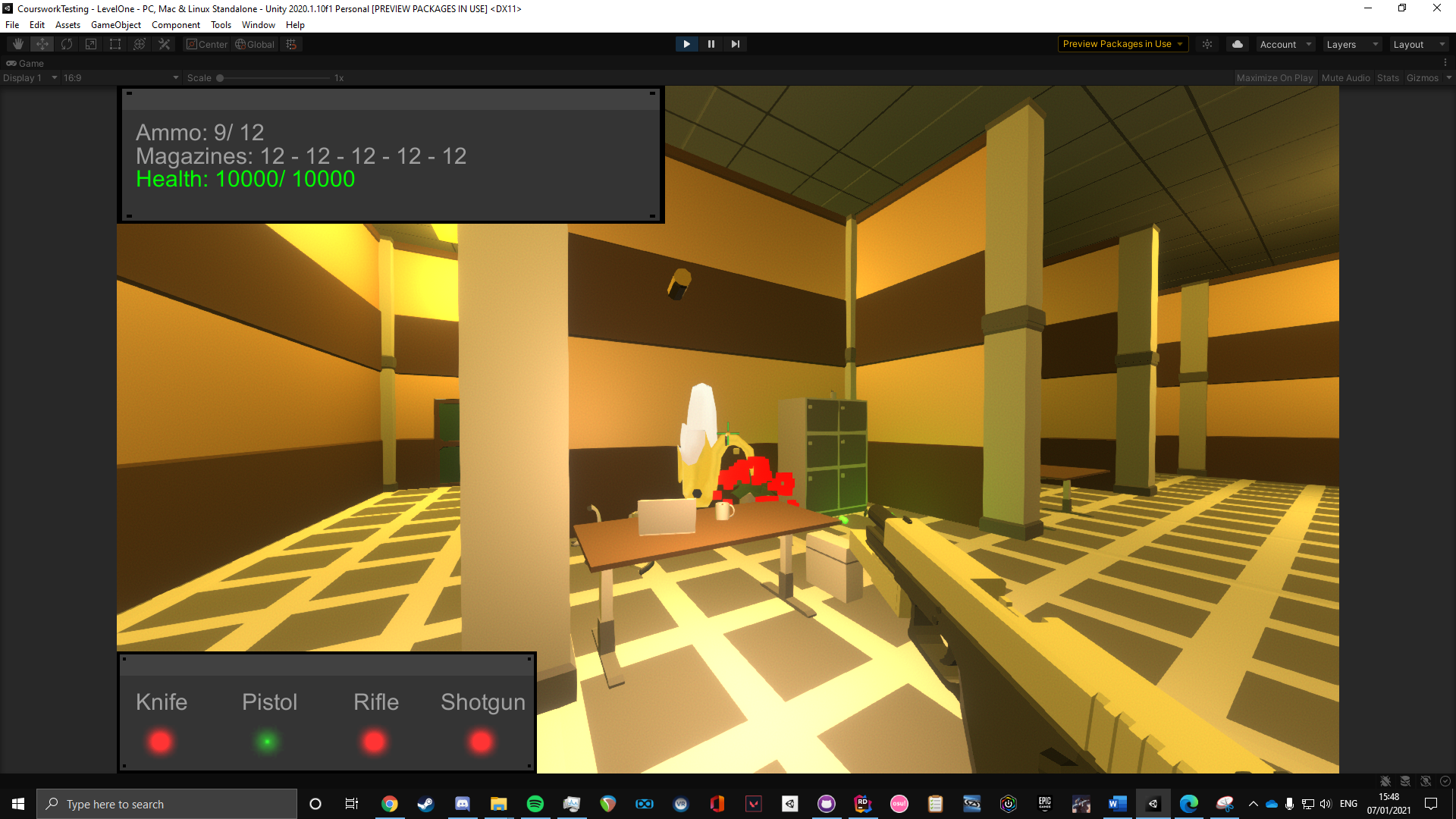
The level system will be immensely important, as it will add a high level of ‘replayability’ into the game, which is often what people find to be what lacks from many games of this genre. The levels will be produced, as according to the specification, in a generally similar theme, with some level of progression of difficulty. This will be noticeable by the colour of the lights, as the difficulty increases, the colours of the lights will change.

In terms of lighting, I will be creating my own objects, which will act as a simple, low-poly office light, which will be able to be used everywhere in the game, with minimal change depending on the level. These will be placed in the ceiling, like the design of a real office light system. The lighting will be done in real time via rasterization, as baking a lightmap into the missions will take too much processing power for the machine the game is being developed on.

Also, research into baked lightmaps would take a lot of time and would take even more time to get it to appear high quality, as seen in many budget Unity games.

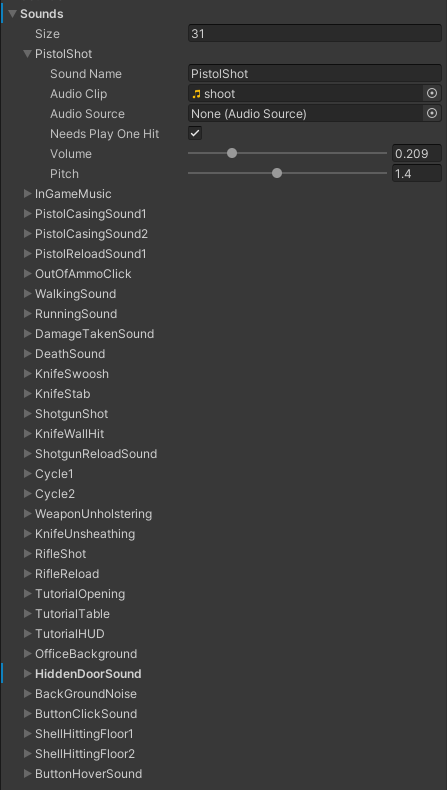
This is where I will be using free Unity assets, the structure of the maps will mostly be part of an asset pack, which is provided by Unity as placeholders. These are low poly assets and go well with the theme of the game.

In order to decrease the demand on the device that it is being played on, there will be minimal levels of extra detail in the maps. This, however, will not be to the detriment of gameplay, as most of the focus of the game will be on the action. Some level of details such as cupboards, office equipment, desks and chairs will be added to make the levels feel more alive.



**Sound**

The sound is quite important, as it really brings immersion into the game. The sound clips that are in the game are taken from various sources, all that are free of copyright. In order to facilitate the playing of sounds in Unity, I created a Sound class, whose objects have a reference to the actual sound file, as well as a default sound source, and a volume and pitch value that allows the developer to customize the audio levels in relation to each other very easily.

***All Sound Clips: (PistolShot is expanded to show the content of each sound object)***

**Escape Menu**

This will be essential to the experience of the game. It will be a hidden panel, which will be activated when the player presses the escape button. Time will also need to be halted, so the game does not carry on when in the menu. The in-built Time class will allow us to achieve this through the Time.timescale property. A fade in / fade out will be added as well to add an extra layer of polish.

# Introduction to Algorithmic Solution

**In order to accomplish this task, it is necessary to understand the structure of the environment. Unity provides many tools to allow this project to be completed efficiently. In each project, there are several elements:**

**MonoBehaviour**

MonoBehaviour is a class that is inherited by all classes and scripts that want to interact with Unity. The MonoBehaviour inheritance allows a class to:

1. Execute a function every frame
2. Execute a function at the start of the game
3. Reference other scripts in the game.

So of course, MonoBehaviour adds a lot of functionality to C# scripts. However, some scripts may not require these behaviours to be effective. We call these scripts Serializable scripts. They are usually meant to be quite lightweight, like a traditional class.

An example of a Serializable class is a settings profile. This would contain various floats, integers, and Booleans, but it would not need any of the functions provided by MonoBehaviour. To increase performance, therefore, these class omit this inheritance.

**GameObjects**

These are essentially all that you see in the game. These allow something to exist in the world space. You can attach certain attributes to them, such as a colour, a shape, an animation. We call these attributes ‘Components’. Most importantly may be the fact you can attach a bespoke script onto any object. This will allow great levels of functionality to be added to the objects. Some examples of GameObjects are the player, the weapons, the enemies, the props in the scenes. Another great thing about GameObjects is the fact that they can have children, i.e. a subordinate GameObject. This is useful as it allows for grouping of multiple GameObjects. For example, I could make all the walls, ceilings, and lights children of a ‘building’ GameObject. This makes it overall easier to work with, as well as providing extra functionality (Such as children inheriting certain components from their parent GameObject).

**Scripts**

Each snippet of code seen in this documentation is, in one way or another, inside a script, which is attached to a GameObject in the scene. These are all generated specifically for this project, so they allow for a great level of control of what goes on in the game. They are written in C#, which may not be the absolute best language for game design, due to its high amounts of overhead, it gets the job done for a smaller project such as this.

**Transforms**

Transforms are a small group of 3 vectors. These vectors include a size vector, a rotation vector, and a position vector. All of these is of the for (x, y, z). A unique transform is attached to each GameObject, which allows us to define exactly those parameters. This allows for dynamic movement of the objects.

**Animation**

Animations are incredibly powerful in Unity. They are used extensively in this project, as they allow for a certain movement of an object to occur at anytime, anywhere. In order to facilitate use of animations, each animated GameObject needs an animation controller. This is manipulated to set up a certain list of parameters, and a certain logic process which runs certain animations on that GameObject at the correct time. For instance, for the pistol, as soon as the pistol enable button is pressed, a “pull up” animation is executed. Then, once it has completed, an idling “breathing” animation is played. If the player fires the gun, a “recoil” type animation is played, and then the pistol goes back to the idle animation.

Animations are very useful to synchronize events in the world space. In each animation, you may put a marker at any point. You can attach a function to this marker, and place it on a frame of the animation. Once this frame is executed, that function will also be executed. This helps us to streamline things, such as a shooting sound starting at the start of a shooting animation.

**UI GameObjects**

UI GameObjects are essential to this project as well. They are used in all types of UI, such as the main menu, the HUD elements, and the options panels. Essentially, they are all GameObjects, but with extra in-built features. They all exist under a Canvas GameObject, which enables the functionality of the buttons and sliders. The fact they are UI elements, they come with the ability to understand when they are being pressed, and the ability to execute a function based on that. This is useful as it allows for movement between the different panels of the options menu, as well as modifying the values of the options.

**RigidBody Component**

Among other, lesser used components, is the RigidBody. This allows us to define a weight to the object. Then, this component will simulate a gravity. So, when it is hit by a projectile (i.e. the player or a bullet), it will move in a realistic manner.

**Collider Component**

This is essential. This allows for the developer to define a certain area in the world space where a GameObject Inhabits. This allows code to be ran in case this area is affected, i.e. if a player walks into a given area. In C# and the following pseudocode, this is denoted with the function called “OnTriggerEnter”.

**Other Components**

There are many components that will be used in this project. Some of these include sound, the CharacterController, and others.

**Prefabs**

These are essentially blueprints for GameObjects, which can be copied into the world at any time using the “Instantiate(prefab)” function, or by dragging them into the environment. This is extremely useful as it allows for global changes to objects of the same time. For example, I could have a prefab of an enemy. This is placed many times in many scenes, but if I wanted to make them a little bit taller, I could simply change the root prefab, and then all of them would show that change.

# Algorithmic solution

**For each part of the decomposed task, there will need to be some level of algorithmic computation behind it. This section overviews the plan for these algorithms in pseudocode.**

**Options**

The options menus are quite a large part of the game, as they allow for a greater customization of the game, which can further fit the players needs. While most games have a comprehensive options menu system, it is often fluent, and easy to understand for new players. This is due to good structuring, such as options being in the most obvious place. This is what I am hoping to attain with the options in this game.

In order to make sub-menus in the options menu, I am using a UI canvas, with various panels in it. Each panel has a certain selection of buttons on it which is relevant to what the panel is. The panels include: A main panel, an options selection panel, video settings panel, audio settings panel, and a game settings panel. These are what are needed for just the in-game menu. For the main menu, there are some more buttons that are needed. Of course, we also need all of the options menu available, as well as buttons to select a mission (which is further explain in the next section). In addition, we will need a button to allow the player to close the game.

As there is not much that can vary in the options menu, not much validation is needed. Any references to other parts of the menus should never return a null value, as this is set up in the editor and will never be changed by the user.

I will be using prefabs, which will enable me to create changes that affect all objects in the game. Prefabs are essentially blueprints, which allow me to copy them and place them in different places of the game. If I change the root of the prefab, that change will be made to all of them in the game. The code related to the UI is very simple, with a single function relating to the purpose of each button, as well as a function on certain buttons that causes the current settings to be saved to a SettingsProfile in the settings directory. Also, each time a panel is loaded, the proper settings need to be shown. As they are prefabs, they do come with predetermined values for the sliders (such as music volume being set to 50%), however for the user’s sake this should reflect the current settings. So, each time a panel is loaded, it takes the data from the appropriate directory (stored as a key variable), and then loads that information into the menu.

In terms of the structure of the scripting for the options menus, there will need to be a base class, called BaseMenu, which will include key variables such as all the sub panels. This class includes methods that will be needed by every panel (audio, video, game). This mainly includes the ability to have a “back button”, when pressed it disables the current selected menu, and activate the menu below (for instance pressing back on the video panel will go to the options menu). From this class, each panel will derive a script, and add other functionality specific to the panel. For instance, the audio panel will need a reference to the directory where audio settings are held, as well as a reference to the buttons and sliders in the menu. Once the back button is clicked, a function is called which writes the current audio settings to the directory, and applies them.

The base class layout is shown below:

*class BaseMenu*

*{*

*function ToMainPanel()  
{*

*Disable all panels (as referenced in an array)*

*Enable the main panel*

*}*

*function ToVideoPanel()  
{*

*Disable all panels (as referenced in an array)*

*Enable the video panel*

*}*

*function ToAudioPanel()  
{*

*Disable all panels (as referenced in an array)*

*Enable the audio panel*

*}*

*function ToGamePanel()  
{*

*Disable all panels (as referenced in an array)*

*Enable the game panel*

*}*

*function PlayHoverSound()  
{*

*/\* This is called in the script reference on the buttons \*/*

*Plays hover sound in the audio manager.*

*}*

*function PlayClickSound()  
{*

*/\* This is called in the script reference on the buttons \*/*

*Plays click sound in the audio manager.*

*}*

*}*

***Important Variables here: Array of all panels, AudioManager.***

***Important Validation here: N/A.***

Here is the audio panel script, which derives from the base class:

*class AudioPanel*

*{*

*function OnEnable()   
{*

*InitializeSettings()   
}*

*function IntializeSettings()  
{*

*Load AudioSettingsProfile from the specified directory.*

*Set each slider to the appropriate setting in the file.  
}*

*function UpdateText()  
{*

*Update the tag next to each slider to show the rounded value that*

*The sliders show*

*}*

*}*

***Important Variables here: Array of all panels, AudioManager.***

***Important Validation here: N/A.***

**Level Selection Menu**

Similar to the options, the code for this is relatively simple. For each button, there will be a single function that causes the game to load into the necessary scene. In terms of the level progression, this is saved in a JSON file of type UnlockedLevelData, which stores what levels the player has finished. On entering the scene, this data is retrieved, and the according unlocked levels are shown to the player.

Add level selection pseudocode

**Player Movement**

This is one of the main features which is entirely held within one script in the final product. Comparatively, this is one of the simplest features that is being added. All of the methods in the snippet below are ran each frame, which means that it polls all input data from the player (such as whether they are holding down A or D to move left or right, which is denoted by the horizontal and vertical input axes respectively). This code, in tandem with the CharacterController, should allow the player to move fluently throughout the game.

*Class PlayerMovementScript*

*{*

*function OnEachFrame()   
{*

*UpdatePlayerRotation()*

*UpdatePlayerMovement()*

*UpdatePlayerRunning()   
}*

*function UpdatePlayerRotation()  
{*

*CurrentRotation = 3D Vector, where*

*x = mouseX axis,*

*y = mouseY axis,*

*z = 0*

*Let CurrentRotation only be between -80 deg and 80 deg in the x.*

*PlayerRotation = PlayerRotation + CurrentRotation  
}*

*function UpdatePlayerMovement()  
{*

*Let newMovement = 3D Vector, where*

*x = horizontal input axis,*

*y = vertical input axis,*

*z = 0*

*Move the player by newMovement*

*IF the player is on the ground and is pressing jump:*

*Jump player*

*ENDIF*

*IF the player is not grounded:*

*Add velocity downwards to player.*

*ENDIF  
}*

*function UpdatePlayerRunning()  
{*

*IF player is walking:*

*Speed of player = walking speed*

*ELSE*

*Speed of player = running speed*

*ENDIF*

*}*

*}*

***Important Variables here: Player Transform, Speed of player, Input axis, Player Sensitivity, Walk Speed, Run Speed.***

***Important Validation here: If player doesn’t exist in the space.***

**Weapon Mechanics**

This part of the code gives function to the weapon objects in the game, so it is vital to gameplay. Due to the overlap of weapon mechanics, such as shooting and generating a bullet casing, I have decided to create an Inheritance system, with all weapons deriving from an abstract ‘weapon’ class, which contains most basic functionality.

*abstract class GenericWeaponBehaviour*

*{*

*function OnStart()  
 {*

*UpdateHUD(CurrentLoadedMag)*

*}*

*function ShootingWeaponUpdate()  
 {*

*currentMagAmmo = CurrentLoadedMag.ammo*

*IF currentMagAmmo == 0 AND isShooting:*

*Play outOfAmmoSound*

*ENDIF*

*IF isShooting AND canShoot AND curentMagAmmo != 0:*

*Start shootingAnimation*

*currentMagAmmo—*

*UpdateHUD*

*CurrentLoadedMag.ammo—*

*IF weapon isn’t shotgun:*

*Create bullet at end of gun*

*Randomize rotation of bullet*

*ELSE*

*Foreach (number of shotgun pellets)*

*Create bullet at end of gun*

*Randomize rotation of bullet*

*Randomize bullet offset*

*ENDFOREACH*

*ENDIF*

*WaitForTime(float time)*

*GenerateBulletCasing()*

*ENDIF*

*}*

*IEnumerator WaitForTime(float waitTime)  
 {*

*canShoot = false*

*canReload = false*

*Wait for waitTime THEN*

*canShoot = true*

*canReload = true*

*}*

*function GenerateBulletCasing()  
 {*

*Create casing at side of gun*

*Randomize rotation of casing*

*}*

*function Reload()  
 {*

*Start reloading animation*

*Play shooting sound*

*Find largest available magazine*

*Load magazine into weapon (set as currentLoadedMag)*

*Update HUD*

*}*

*function CheckReload()  
 {*

*IF canReload and is pressing R key:*

*Reload()*

*}*

***Important Variables here: Array of all weapons, Ammunition prefabs, Player Weapon UI, Casing Prefab, Audio Manager, Shooting sound clip, out of ammo sound clip.***

***Important Validation here: Only runs if the weapon is enabled (if weapon is being used).***

As you can see from this class, it is abstract. This means that, while it provides methods, they are never actually called in the class itself. The use of the methods is up to the deriving classes, where the functions ‘ReloadCheck’ and ‘ShootingWeaponUpdate’ will be used.

Here is the one of the deriving classes, this is the pseudocode for the pistol script, which implements the behaviour defined in the GenericWeaponBehaviour. All of the objects referenced in the derived class are overridden here, so that the methods that are called are called in reference to the objects that are for the pistol (bullet is a pistol bullet, front of the gun is the front of the pistol, etc.)

*class PistolBehaviour() Deriving from GenericWeaponBehaviour*

*{*

*/\*Lots of overridden variables from derived class\*/*

*function OnEachFrame()  
 {*

*ShootingWeaponUpdate()*

*CheckReload()*

*}*

*}*

There is a deriving class that is unique to all the weapons in the game. For each of them, as they act differently from each other in certain ways, there is checks in the base class. If the class is defined as a shotgun, it will run the base class as a shotgun. The same will be seen with the rifle.

**Weapon UI**

The weapon UI is meant to be easy to understand for the player. In order to achieve this, I wanted to designed it to be as simple as possible. There will be 2 panels, one contains the information about health, and ammunition. The other panel will show the player what weapons they have at their disposal, as well as the current weapon they are using.

Most of the code relating to this is held in other scripts, such as in the base weapon class. For the weapon class, each time the weapon is enabled (such as when it is the current active weapon), the UI will change to data to do with the currently selected weapon. This will also need to be deactivated once the menu is opened, this is controlled in the menu script also.

**Enemy AI**

In terms of enemies, they all derive from the same prefab, which means they all act in a similar way, with similar scripts attached. This allows for a greater level of modification later in development. In order to achieve the random patrol mechanic, I have made an empty GameObject in each map, each with a few children which are other GameObjects. These children are essentially just to hold a position on the map where patrols are meant to end. These are chosen at random if a specific enemy is designated as a patrol enemy (See RandomNavMeshLocation method below).

This is separated into two different scripts, both of which are accessed by each enemy. One of these is the EnemyBehaviourScript:

*class EnemyBehaviourScript()*

*{*

*function OnEachFrame()  
 {*

*IF the player exists AND the enemy can see the player:*

*CheckEnemyFieldOfView()*

*ELSE*

*IF this enemy is a patrol enemy AND has no current path:*

*StartRandomPath()*

*ENDIF*

*ENDIF*

*}*

*function TakeDamage(float damage)  
 {*

*Decrease this enemies health count by the function input.*

*IF health is lower than 1 AND the enemy is not already dead:*

*Die()*

*ELSE:*

*Play damaged sound on enemy.*

*ENDIF*

*}*

*function Die()*

*{*

*Play death audio sound.*

*Add one to the number of kills the player has.*

*Destroy this enemy GameObject, after dropping the audio source.*

*}*

*function CanSeePlayer()*

*{*

*Send out a ray towards the player from the enemies eyes.*

*IF the ray hits another object before hitting the player:*

*Mark enemy as blocked from view.*

*ELSE IF the ray hits the player first:*

*Set enemy as being able to see player*

*ENDIF*

*}*

*function CheckFieldOfView()*

*{*

*IF the player doesn’t exist:*

*Return from this method.*

*ENDIF*

*IF the distance from the enemy to the player is bigger than the enemy’s field of vision:*

*Return from this method.*

*ENDIF*

*/\* From this point on in the method, we are assuming the enemy has seen the player\*/*

*Start a countdown till the enemy starts shooting.*

*Set the enemy as having seen the player.*

*Set the enemy’s destination as the vector of the player’s transform.*

*IF the enemy is inside the stopping distance from the player:*

*Stop the enemy moving*

*MakeEnemyFaceTarget()*

*MakeGunBarrelPointToTarget()*

*ShootAtPlayer()*

*ENDIF*

*}*

*function MakeEnemyFaceTarget()*

*{*

*Set the direction the enemy is facing to the player.*

*/\* This is quite complex vector math, so deserves its own method \*/*

*}*

*function MakeGunBarrelFaceTarget()*

*{*

*Set the direction of the barrel of the enemy’s gun towards the player.*

*}*

*function ShootAtPlayer()*

*{*

*Start the shooting script in the enemy’s gun.*

*}*

*function RandomNavMeshLocation()*

*{*

*Return a random child of the waypoint GameObject.*

*}*

*function StartRandomPatrol()*

*{*

*Once this is started, it calls RandomNavMeshLocation() every 10-20 seconds. This allows for the enemy to seem more ‘alive’. This is called once at the start of the life of this enemy, but not again after it sees the player.*

*}*

*}*

***Important Variables here: Enemy Transform, AudioManager, Waypoint Parent, Player Transform, Enemy Weapon, Enemy Weapon Script.***

***Important Validation here: If player still exists.***

But, of course, each enemy has a weapon. This weapon needs a script for functionality. This controls the creation of weapon sounds, bullets, and bullet casings, and is called the EnemyWeaponScript:

*class EnemyWeaponScript()*

*{*

*/\* This has a lot of boilerplate; it is just a simplified version of the player pistol script \*/*

*function Shoot()*

*{*

*Start a timer till next weapon shot.*

*IF the player exists AND the enemy can see the player:*

*IF there are no bullets left:*

*Play Click Sound*

*Return from this method.*

*ENDIF*

*Spawn a bullet in front of gun.*

*Play Shooting Animation.*

*Decrement the bullet quantity left.*

*SpawnBulletCasing()*

*}*

*}*

**Levels**

Creating the levels requires little code. In order to facilitate the creation of levels, I will be using tools given by Unity. In terms of assets, I am using free placeholder assets, which are also from Unity for free. In terms of extra assets, I have created extras such as Lights and post-processing effects to give the levels some extra uniqueness.

In order to add life into the scenes, various objects in the scene move. General objects on the ground, such as cupboards, tables, chairs, cabinets, computers, and others, will have a RigidBody attached to them (as explained in the section previous). Also, in many levels, there will be hidden doors that allow for a sense of progression in the game. These are controlled via a counter of how many enemies have been killed by the player. Once the player has killed enough enemies in a certain area, a hidden door may open to allow the player to proceed. The actual movement is controlled by an Animation.

At the end of a level, a trigger GameObject is placed. This is a certain area, where, once entered, will run a function which ends the game.

*class EndTriggerScript()*

*{*

*IF the object inside the collider is the player:*

*Continue.*

*ELSE*

*Return.*

*ENDIF*

*Fade camera to black.*

*function OnTriggerEnter()*

*{*

*SWITCH level:*

*Case 1:*

*Set mission 1 as finished.*

*Break from switch.*

*Case 2:*

*Set mission 2 as finished.*

*Break from switch.*

*Case 3:*

*Set mission 3 as finished.*

*Break from switch.*

*Etc..*

*ENDSWITCH*

*}*

*}*

**Sound**

The sound class is defined with certain attributes. These contain name, audio clip reference, desired audio source, as well as volume and pitch. These are all used when playing the sound to make sure that it is played properly. Objects of type Sound are stored in an array which exists under the sound manager script in the scene. This is where they are retrieved from when they are played.

The sound manager script, which is placed in every scene, controls what is played at what time, it is defined with the following:

*class SoundManager()*

*{*

*function OnStart()  
 {*

*Load audio settings from file.*

*PlayMusic()*

*}*

*function WriteAudioSettings()  
 {*

*Takes values from settings menu, writes them to file.*

*Applies new values to sound.*

*PlayMusic()*

*}*

*function Play(nameOfSound, audioSource)  
 {*

*Fine sound by its name from array.*

*Set audio source to desired audio source.*

*If it is not already, make sure the audio*

*source contains the audio clip.*

*SetVolumeAndPitch()*

*}*

*function SetVolumeAndPitch()  
 {*

*Sets volume and pitch to predetermined values.*

*Modify the values based on what is on the AudioSettingsProfile.*

*}*

*function PlayMusic()  
 {*

*/\* This is called in the WriteAudioSettings section so when the*

*changes are made to the volume / pitch, they affect the already*

*playing music, instead of waiting until it has finished. \*/*

*Play(musicName, musicSource)*

*}*

*}*

**Escape Menu**

The code for the escape menu is relatively simple, just like most of the UI in the game. There will be 3 main buttons, one to return to the game, one to return to the main menu, and one to enter the options menu. The activation of the escape menu is handled by a different script. The pseudocode for this can be seen here:

*class SoundManager()*

*{*

*\*/ All of this code’s functions are called by the*

*actual buttons in the scene, which has no code \*/*

*function OnStart()  
 {*

*Disable the menu panel.*

*Turn time to 0 speed.*

*PlayMusic()*

*}*

*function LoadMainMenu()  
 {*

*Loads the main menu scene.*

*}*

*function ReturnToGame()  
 {*

*Disables the menu panel.*

*Turn time to full speed.*

*}*

*}*

# Testing Solution

**Testing is a very important part of the development cycle of this project. The main area of testing is through constant testing of various inputs whenever a new feature is added. For instance, when movement was implemented, every single key, include multiple keys together at the same time were tested. However, doing this while coding may have left some holes where no testing occurred. This section is aimed to fill those holes.**

**Testing**

The nature of each testing element is important, these are as follows:

**Normal** ~ A correct input, which should achieve a correct result

**Erroneous ~** An input which shouldn’t do anything.

This would likely need to be actively dealt with by the code.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number | *Area of Test* | *Action* | *Nature* | *Expected Results* | *Outcome + Notes* |
| **1** | **Options Menu** | **Left Click Audio Panel / Game Panel / Video Panel.** | **Normal** | **Respective panel to open.** |  |
| **2** |  | **Pressing escape.** | **Normal** | **Should open the main menu + affect the changes.** |  |
| **3** |  | **Pressing the escape button.** | **Normal** | **Should open the menu.** |  |
| **4** |  | **Hovering over any button.** | **Normal** | **Should play hovering sound.** |  |
| **5** |  | **Clicking any button.** | **Normal** | **Should cause the button to make a clicking sound.** |  |
| **6** |  | **Changing a setting.** | **Normal** | **Should do nothing.** |  |
| **7** |  | **Changing a setting + pressing back.** | **Normal** | **Should affect the appropriate change.** |  |
| **8** |  | **Moving sliders.** | **Normal** | **Should create a fluid movement of the slider.** |  |
| **9** |  | **Pressing right click anywhere.** | **Erroneous** | **Should do nothing.** |  |
| **10** |  | **Pressing left click on anything that isn’t a button.** | **Erroneous** | **Should do nothing.** |  |
| **11** |  | **Try to reload weapon.** | **Erroneous** | **Should do nothing.** |  |
| **12** |  | **Opening Dropdown, not choosing an option, then pressing back.** | **Erroneous** | **Should just keep the original setting.** |  |
| **13** |  | **Changing weapon.** | **Erroneous** | **Should not change weapon.** |  |
| **14** | **Level Selection Menu** | **Movement.** | **Erroneous** | **Should not move.** |  |
| **15** |  | **Pressing the respective buttons for the maps.** | **Normal** | **Should load the respective map.** |  |
| **16** |  | **Pressing the back button.** | **Normal** | **Should take the player back to the main menu.** |  |
| **17** |  | **Left clicking anything that isn’t a button** | **Erroneous** | **Should do nothing.** |  |
| **18** |  | **Trying to select a weapon.** | **Erroneous** | **Should do nothing.** |  |
| **19** |  | **Trying to press buttons of maps not yet unlocked.** | **Erroneous** | **Should do nothing.** |  |
| **20** |  | **Pressing the escape key.** | **Erroneous** | **Should not bring up exit menu.** |  |
| **21** | **Player Movement** | **Using W, A, S, D to move.** | **Normal** | **Should move in the correct direction.**  **W – Forward**  **A – Left**  **S – Back**  **D – Right** |  |
| **22** |  | **Using more than one movement key at the same time. (Non-opposite).** | **Normal.** | **Should move in the vector sum of those directions. (i.e. W + D would move you forward + right).** |  |
| **23** |  | **Using more than one movement key at the same time. (Opposites).** | **Normal** | **Should cause no movement (as they are vectors, they should cancel out).** |  |
| **24** |  | **Using shift while moving.** | **Normal** | **Should move in the correct direction, but somewhat faster.** |  |
| **25** |  | **Jumping while stationary.** | **Normal** | **Should move up and down in a semi-realistic fashion.** |  |
| **26** |  | **Jumping while moving.** | **Normal** | **Should move up and down in a realistic fashion while still moving in the direction at the start.** |  |
| **27** |  | **Moving adjacent + touching a wall.** | **Normal** | **The player should move along with the wall, not clip through.** |  |
| **28** |  | **Moving directly into a wall.** | **Normal** | **Should come to a halt, not clip through the wall.** |  |
| **29** |  | **Colliding with non-static objects in the scene.** | **Normal** | **Should not clip through non-static geometry (i.e. tables, chairs).** |  |
| **30** |  | **Walking up stairs.** | **Normal** | **Should be able to walk up any stairs with ease (not get stuck).** |  |
| **31** | **Weapon Mechanics** | **Left click.** | **Normal** | **Should only shoot the weapon that is activated.** |  |
| **32** |  | **Changing weapon.** | **Normal** | **Should put away current weapon, take up pistol, disable old weapon.** |  |
| **33** |  | **Changing weapon.** | **Normal** | **After the animation, should be able to shoot.** |  |
| **34** |  | **Shooting on an empty magazine.** | **Normal** | **Should cause a click + not shoot.** |  |
| **35** |  | **Reloading the knife.** | **Erroneous** | **Should do nothing.** |  |
| **36** |  | **Reloading while shooting** | **Erroneous** | **Should carrying on shooting + ignore reload request.** |  |
| **37** |  | **Reloading while putting away weapon.** | **Erroneous** | **Should carry on putting away + ignore reload request.** |  |
| **38** |  | **Shooting while putting weapon away.** | **Erroneous** | **Should not shoot.** |  |
| **39** |  | **Holding down shoot button with anything other than rifle.** | **Erroneous** | **Should only shoot one shot.** |  |
| **40** |  | **Holding down shoot with the rifle.** | **Normal** | **Should fire continuous shots.** |  |
| **41** |  | **Single click with rifle.** | **Normal** | **Should only shoot one bullet.** |  |
| **42** |  | **Holding down shoot, then stopping with rifle.** | **Normal** | **Should stop shooting.** |  |
| **43** |  | **Changing to another gun while already changing gun.** | **Erroneous** | **Should carry on changing the initial gun.** |  |
| **44** | **Weapon UI** | **Switching between weapons to test display.** | **Normal** | **Light should change to green when weapon is active, red when not.** |  |
| **45** |  | **Moving between weapons rapidly.** | **Erroneous** | **Should, in the end, show the correct weapon as green.** |  |
| **46** |  | **Shooting updating ammo counter.** | **Normal** | **Should decrease the number displayed for the current mag as one.** |  |
| **47** |  | **Reloading weapon.** | **Normal** | **Should cause the ammo count to update correctly.** |  |
| **48** |  | **Changing weapon.** | **Normal** | **Should cause the ammo counter to display the ammo count for the appropriate weapon.** |  |
| **49** |  | **Getting shot.** | **Normal** | **Should cause the health counter to display the correct new lowered health.** |  |
| **50** |  | **Health goes below 50%** | **Normal** | **Should cause the health counter to go yellow.** |  |
| **51** |  | **Health goes below 20%** | **Normal** | **Should cause the health counter to turn red.** |  |
| **52** |  | **Having empty magazines.** | **Normal** | **If 0, they should not be displayed on the counter.** |  |
| **53** | **Enemy AI** | **Sees player.** | **Normal** | **Should turn to move to player and start moving.** |  |
| **54** |  | **Stop seeing player.** | **Normal** | **Should carry on trying to accost player.** |  |
| **55** |  | **Comes within stopping range of player.** | **Normal.** | **Should start shooting at the player.** |  |
| **56** |  | **Runs out of ammo.** | **Normal** | **Should carry on clicking, but not shoot any bullets.** |  |
| **57** |  | **Given a waypoint.** | **Normal** | **Should be able to path towards the waypoint when given.** |  |
| **58** | **Levels** | **Succeeding in a mission.** | **Normal** | **Should unlock next map.** |  |
| **59** |  | **Shooting non-static objects.** | **Normal** | **Should respond by being pushed back.** |  |
| **60** | **Sound** | **Sounds play simultaneously from the same source.** | **Normal** | **Should both play at the same time (fixable with Sound.PlayOneHit)** |  |
| **61** | **Escape Menu** | **Pressing Escape** | **Normal** | **Should return to game at full speed.** |  |
| **62** |  | **Pressing around the menu.** | **Erroneous** | **Should do nothing.** |  |

# Post-Testing Data

In order to further improve the project, there are two sets of people we should get data from:

1. **The main client – Hong**

His view is important, as he is the main client. Once the testing stage is complete, and the first model of the project is complete, he will be queried on whether he likes the solution. If he comes up with anything of note to add or change, that will be done.

1. **Other clients**

In addition to the main client, it may be of some use to ask other clients about what they think of the game. As aforementioned, there are some others that are interested in the development of the project. I will send the solution to these people as well and ask them the same question as to Hong.

If any updates are desired by the clients, these will be implemented, tested, and then given back to the clients to see if they are satisfied with the outcome. If not, we can iterate development until they are happy.

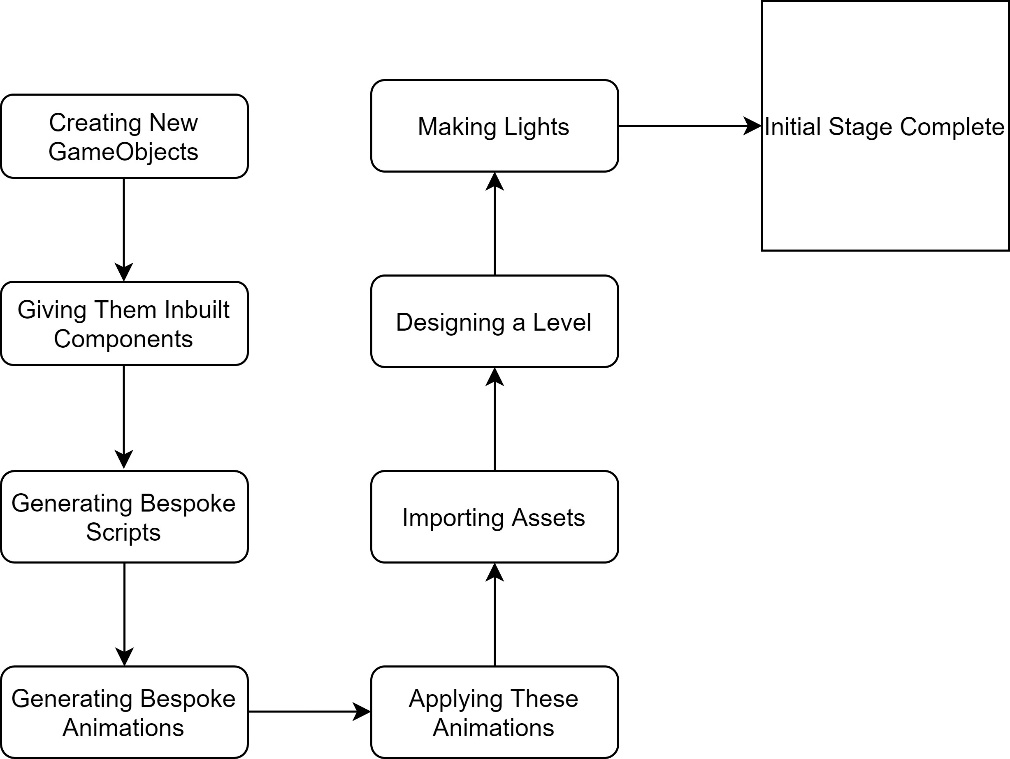
development

# Introduction To Development Stage

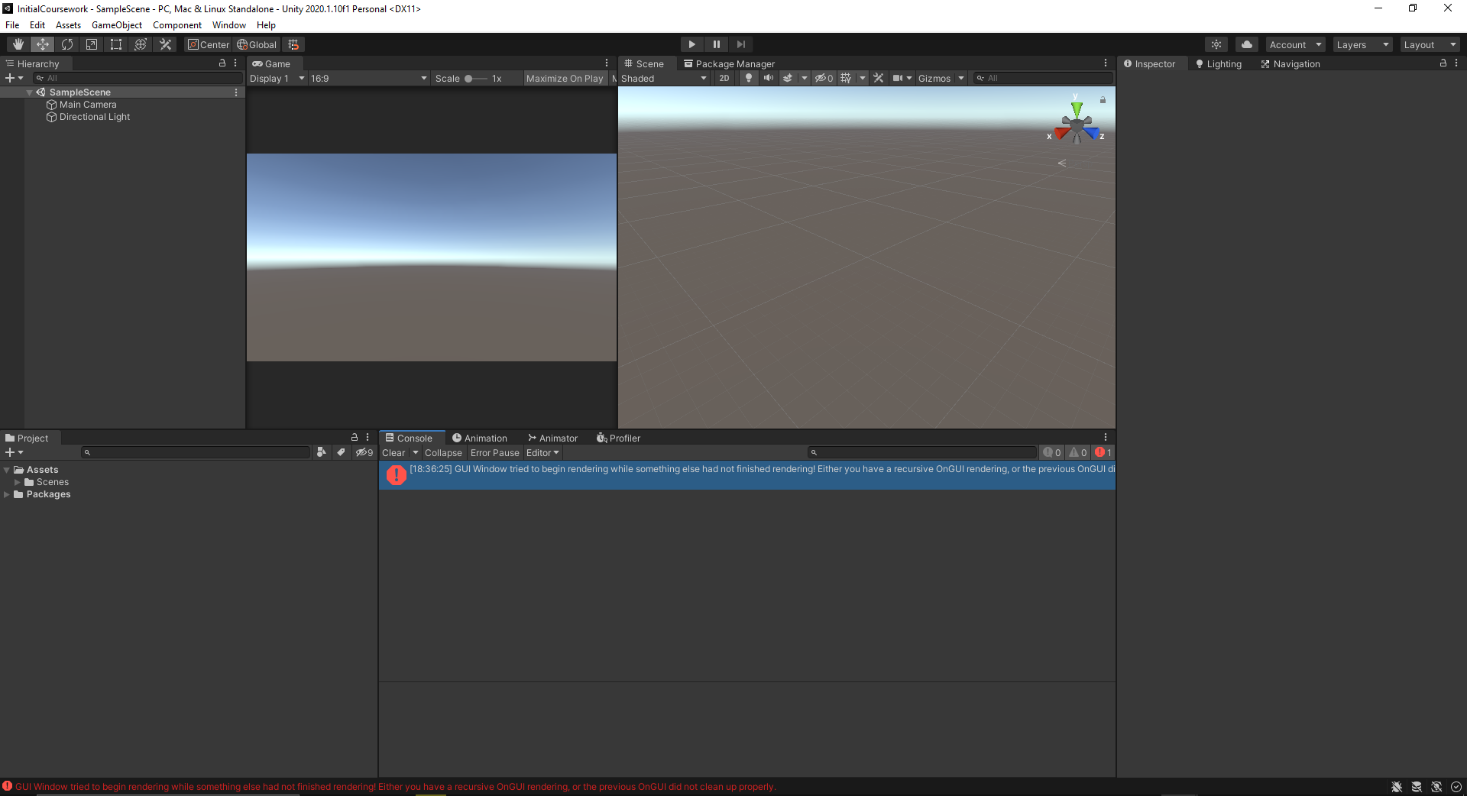
In order to start working with Unity, there are a few prerequisites. First, you must download a C# capable IDE. For this, I chose JetBrains Rider. This is because it has the best functionality of any IDE, I have seen to generate code for Unity. And of course, Unity itself needs to be downloaded, this is simply done as an option when downloading Rider.

**In this section, if you do not understand any of the words, please refer to** [**page 24**](#_Introduction_to_Algorithmic) **for an explanation.**

# Understanding Basics

Since Unity has many relatively complex features, it was necessary to start several test projects before committing to the actual project. First, I wanted to understand the machinations of Unity through the following:

**This section is used as a “Tutorial” into understanding the basics of Unity, which was essential in understanding it enough to use it.**

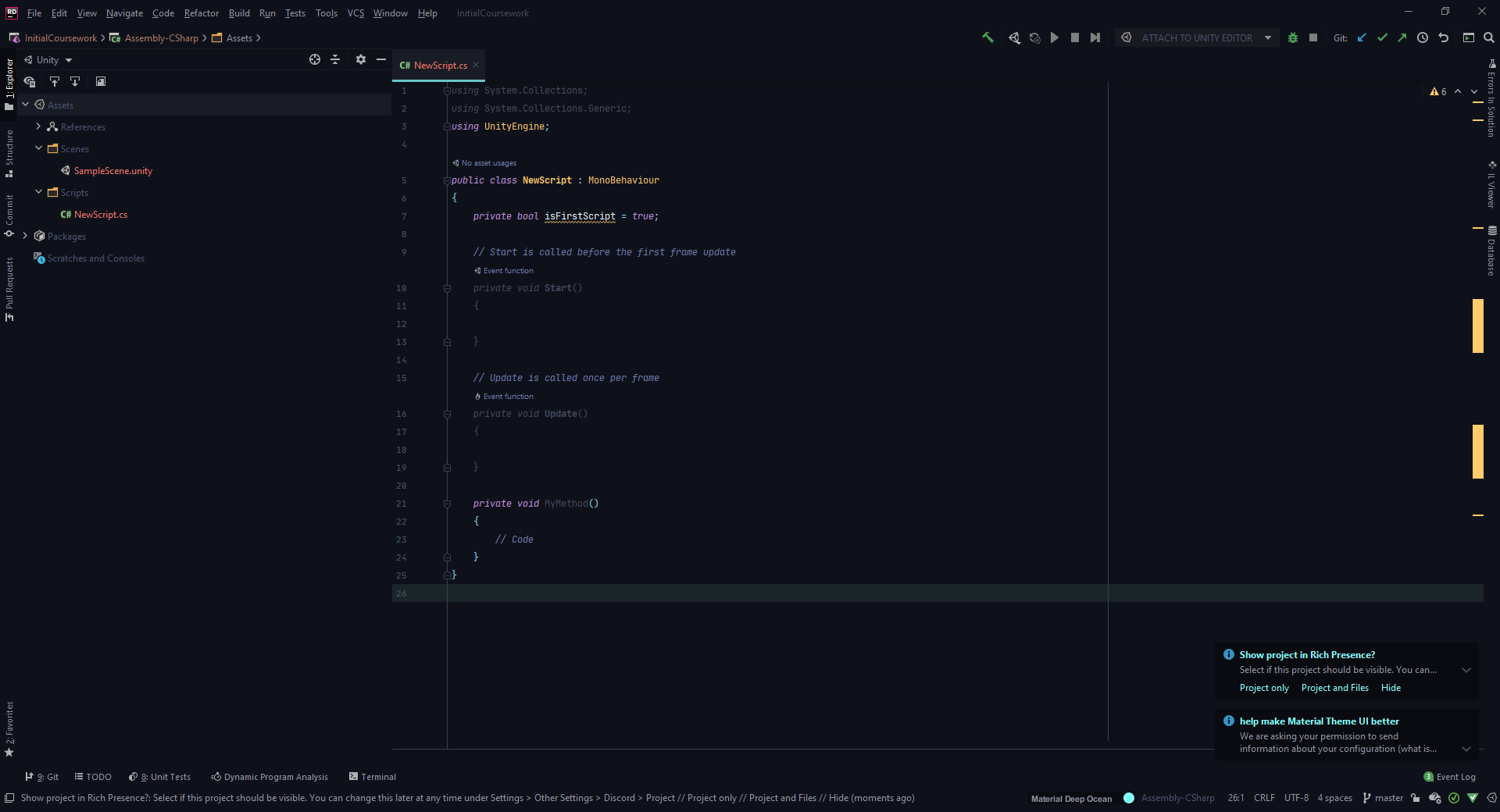
**Creating GameObjects**

**This is the main Unity Editor window; this allows us to see all that is going on in the project.**

**Hierarchy – List of Objects in the scene.**

**Project – List of all resources that we can access.**

**Console – Where any notifications are given to the developer.**

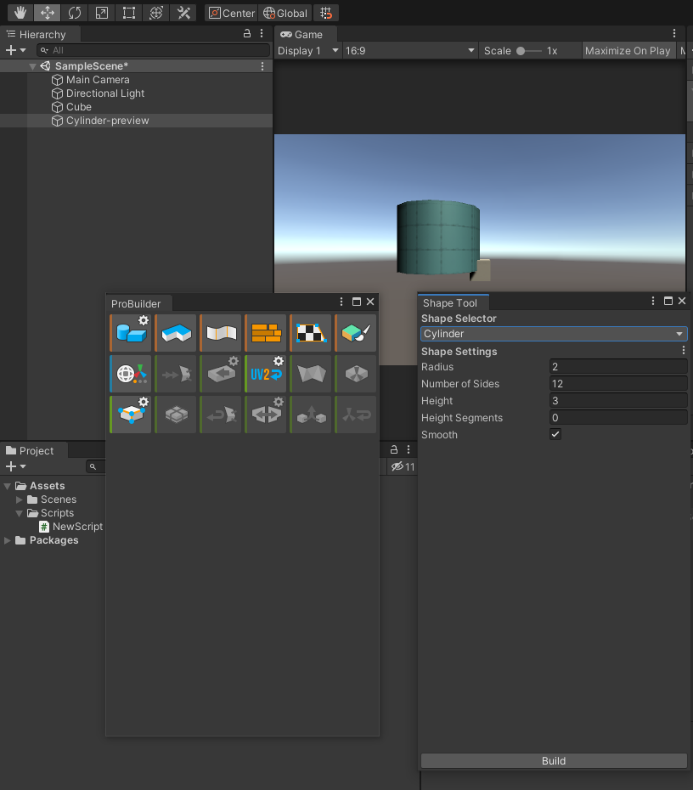
**Inspector – Where elements of a GameObject can be altered.**

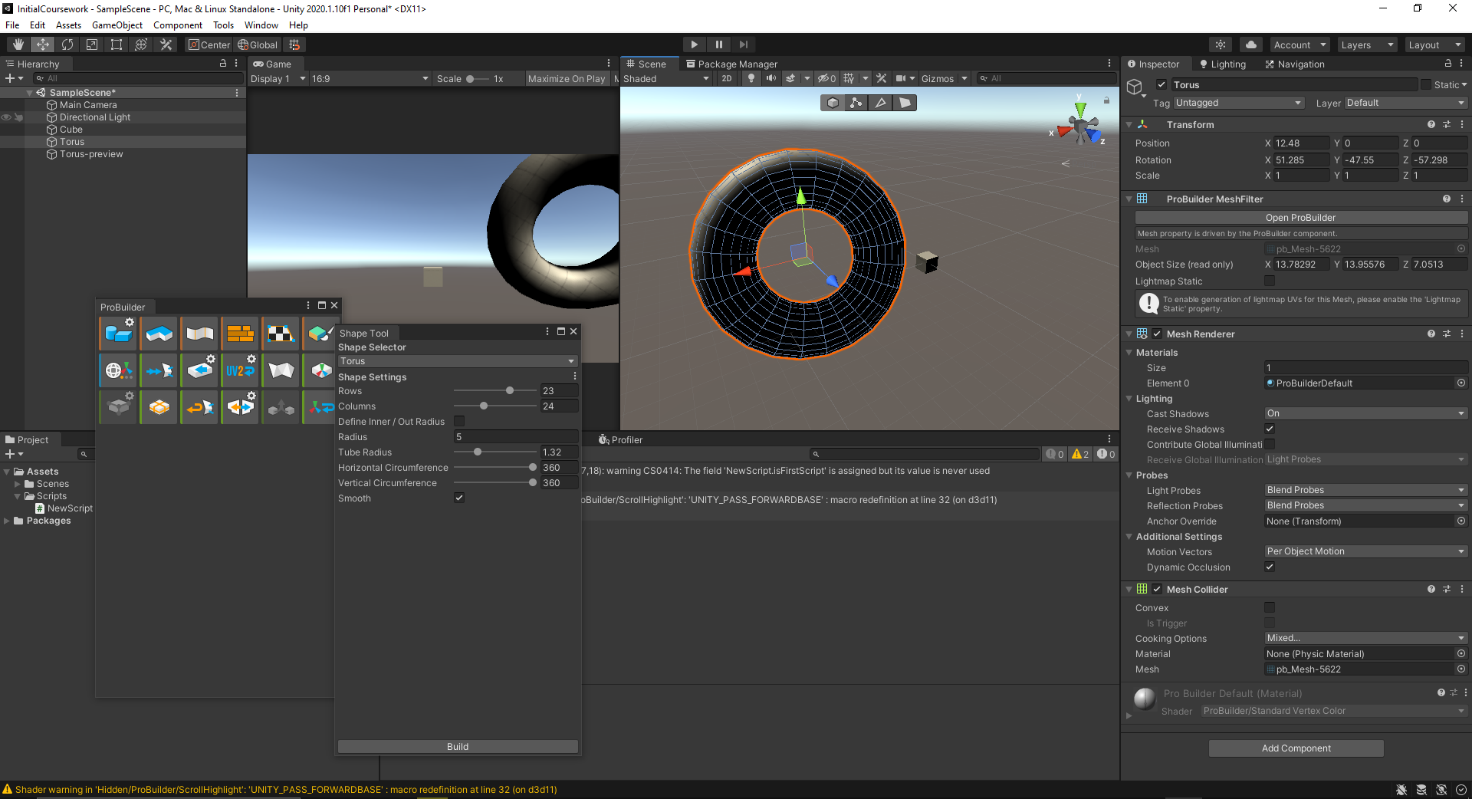
**This is JetBrains Rider IDE.**

First, I will create a new cube object. This is done in the left-hand side:

**This is the easiest part and will be done many thousands of times in order to finish this project.**

Of course, I cannot make an entire project made entirely of cubes. Therefore, I will be using Unity’s proprietary ProGrids and ProBuilder. ProGrids is used to allow for easy of manipulation of objects (i.e. where to put them, how to snap them together), and ProBuilder is used to generate new, interesting GameObjects. Here is the ProBuilder Window:

****

As you can see, there are two windows. One is to choose what to do with the object (cut, add new edge, etc.), and the other allows you to manipulate a new GameObject. Here I have added a new Torus GameObject into the scene:

**Giving GameObjects Inbuilt Components**

This is a vital step, as we need inbuilt components to deal with low-level behavior of objects, whose development for the sake of this one project would be too much to handle. Things such as physics and rendering fall under this category. As you may have noticed from previous screenshots, there is a certain number of Components already added to an object when you create one. One of these is the Mesh Renderer, which essentially deals with all the graphical elements of an object, such as the colour, and shape.

Here is an example of added component, the RigidBody:

**This shows all the properties you can add to a GameObject with a RigidBody.**

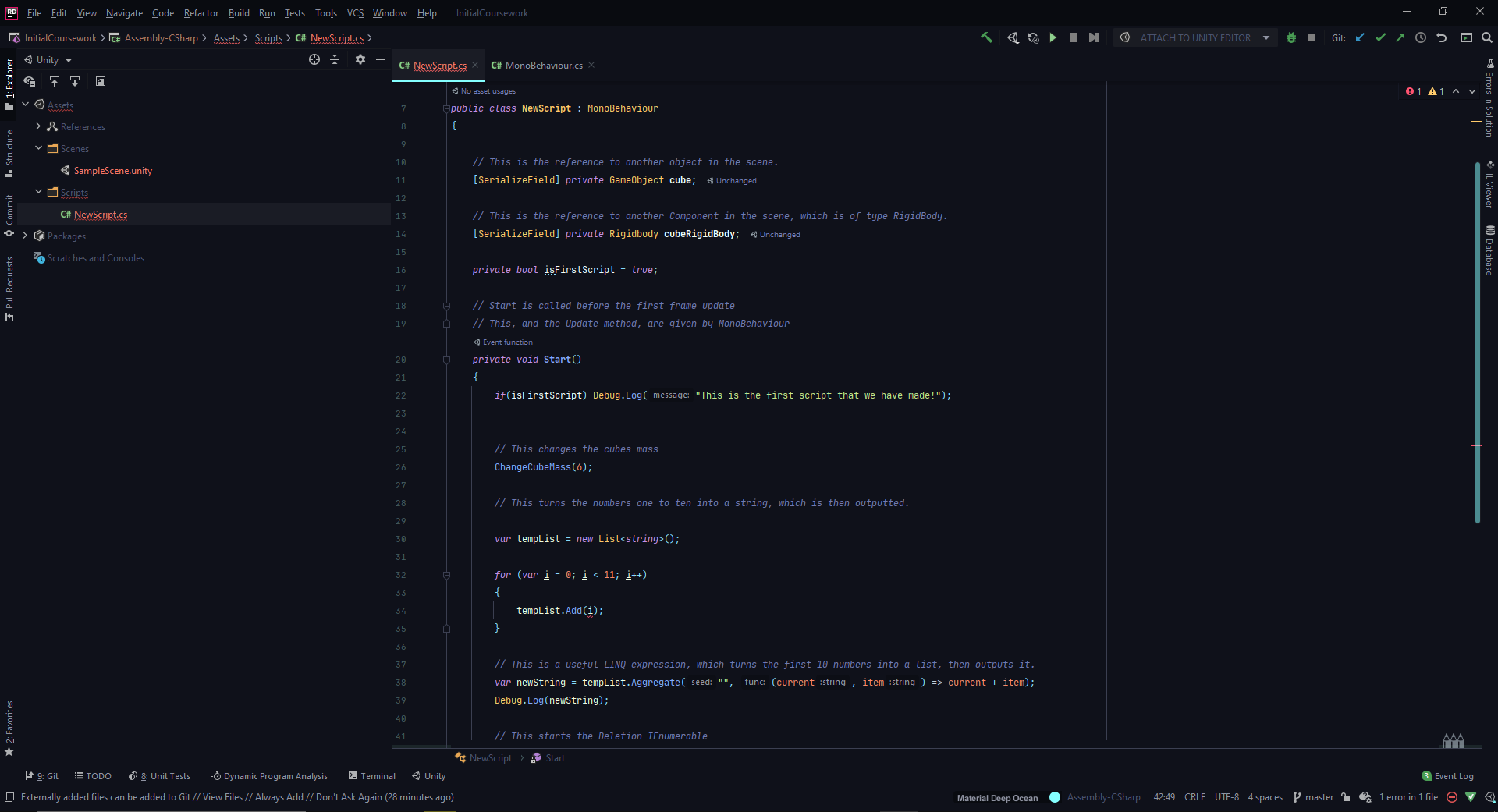
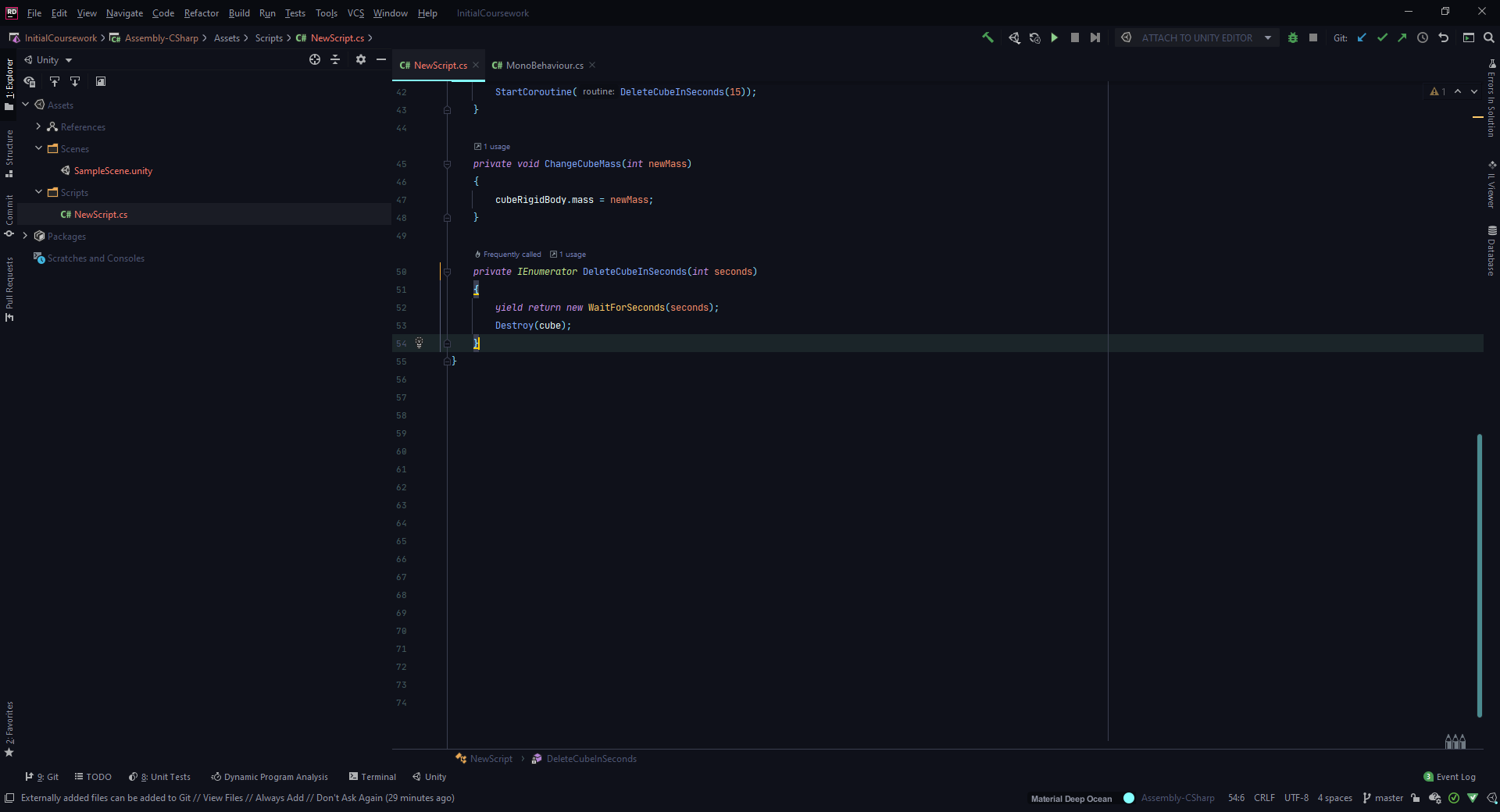
After adding this Component to the torus, when the scene is running, the torus fell indefinitely due to the “Use Gravity” Boolean being set to true.

**Using Bespoke Scripts**

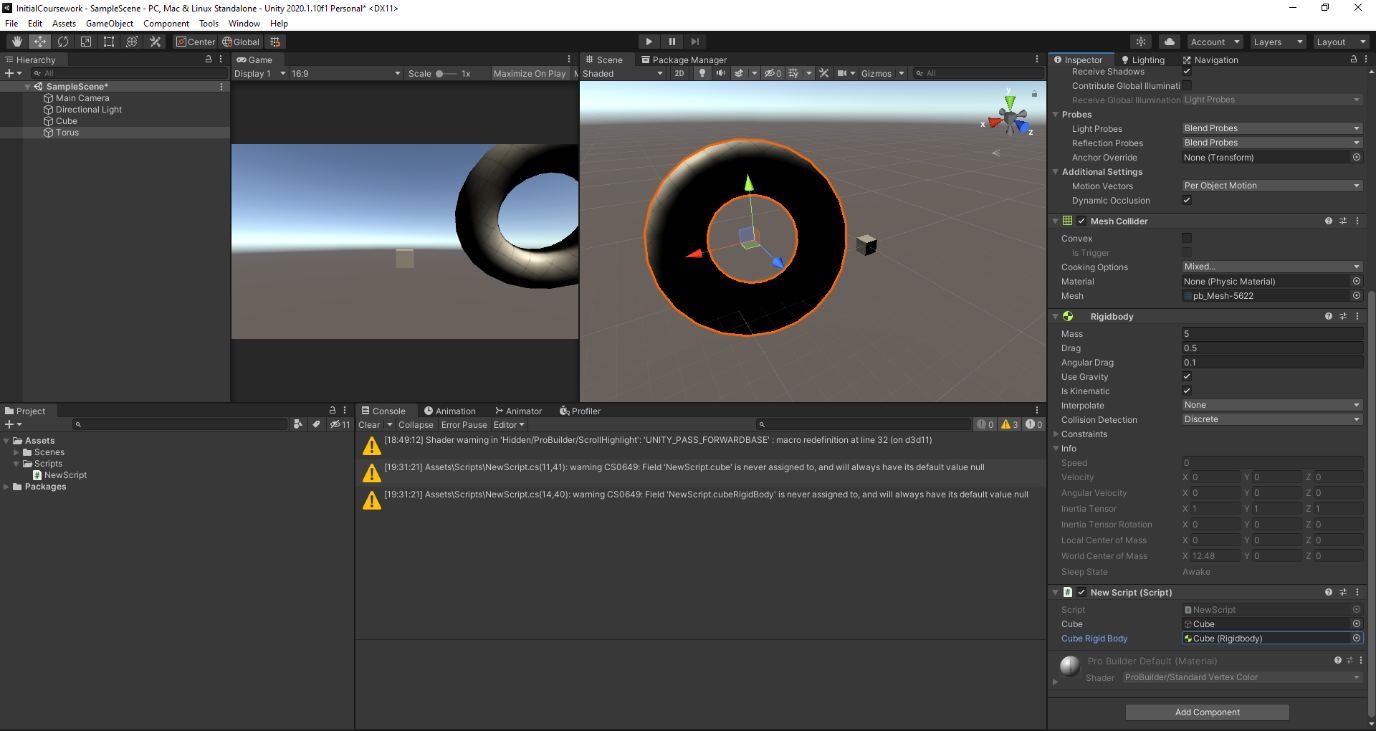
There would not be much point in creating a game without adding some level of uniqueness into it. This is where bespoke scripts come into play. With the use of the IDE, I will generate many scripts based on the pseudocode shown in the earlier chapter, as well as some other, less important scripts that cause minor changes to the game.

In order to make sure I understood how scripts work in Unity, I decided to use as many techniques as possible in this script, they include:

1. Referencing other scripts
2. SerializeField attribute (A property given to fields allowing reference of an object)
3. Private variables
4. Methods
5. Using MonoBehaviour
6. String manipulation
7. Debugging (Using Debug.Log() to send comments to the console during runtime)
8. IEnumerables (A way of iterating over a unique value type (i.e. seconds))



**The comments in the code clearly define what is going on at each point.**

Here is the way the SerializeField attributes are defined:

**The bespoke script is added to the Torus, which executes it. Under the script (bottom right) is where the SerializeField property is used.**

After starting the game this is what is achieved:

1. In the console, a “This is the first script we have made!” comment is made
2. Next, the list of numbers from one to ten is also in the console.
3. The weight of the cube is set to 6.
4. After 5 seconds, the cube disappears (it has been destroyed).

**Animations – Creation and Usage**

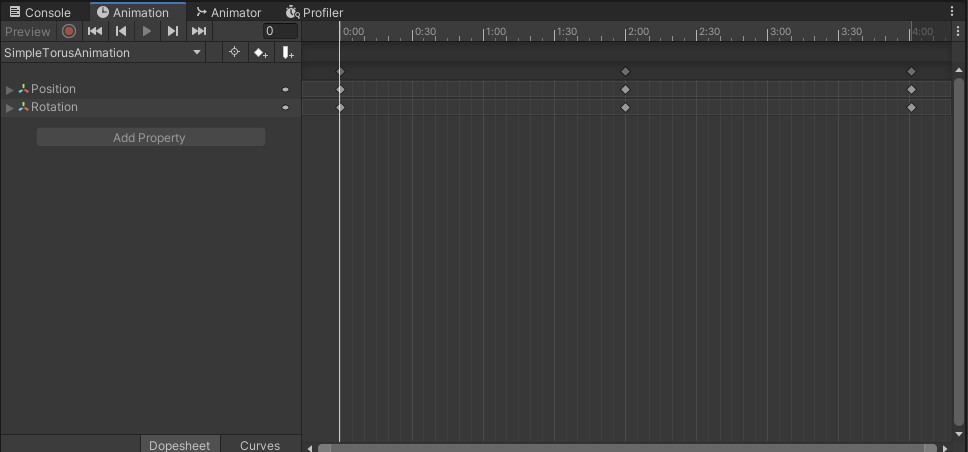
Animations are used extensively in this project, especially when it comes to reloading, shooting, and putting away weapons. This is because it allows an easy way to make a GameObject execute a movement, no matter where it is, or how complex the movement is. Animation is encompassed in two major aspects:

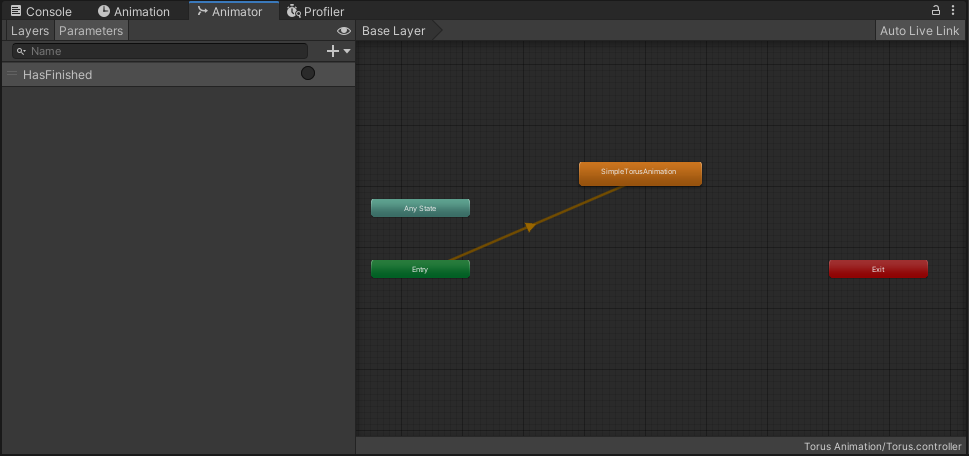
1. **The Animation Files**

This the file that is used to define an actual animation. It contains all of the positions, rotations, and size vectors as well as other more complex data to complete an animation.

1. **The Animation Controller**

This is what is used to control animations. This is a component attached to a GameObject which is to be animated. This controls what animations occur at what times. They use triggers and Booleans to run animations, as well as smooth transitions between animations.

As an exercise to understand animations, lets create a simple animation for the torus:

**This is the animation window. Each diamond represents a change in the properties of the torus. There is automatic smoothing between each point, to create a nice animation.**

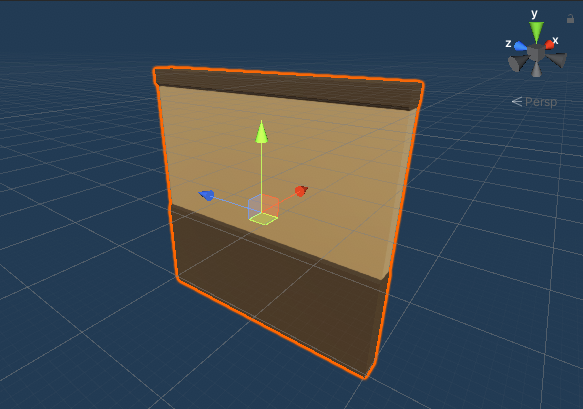
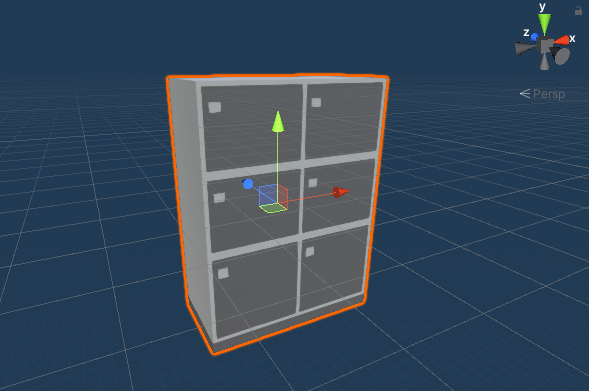
**This is the Animator Controller window. This allows for defining when animations should happen. Here, the Component simply states that on entering the scene, the SimpleTorusAnimation should happen.**

When ran, this executes the animation. Now we know how to animate.

**Using Assets**

Unity is renowned for having a large community that produce lots of assets that are free to use. I have taken advantage of this on occasion, as generating our own assets would be costly and inefficient in terms of time. Only a few assets are used, which are listed below. Most of the lights used in this project are made by me, as I never found any assets that I found appealing to the design of this project.

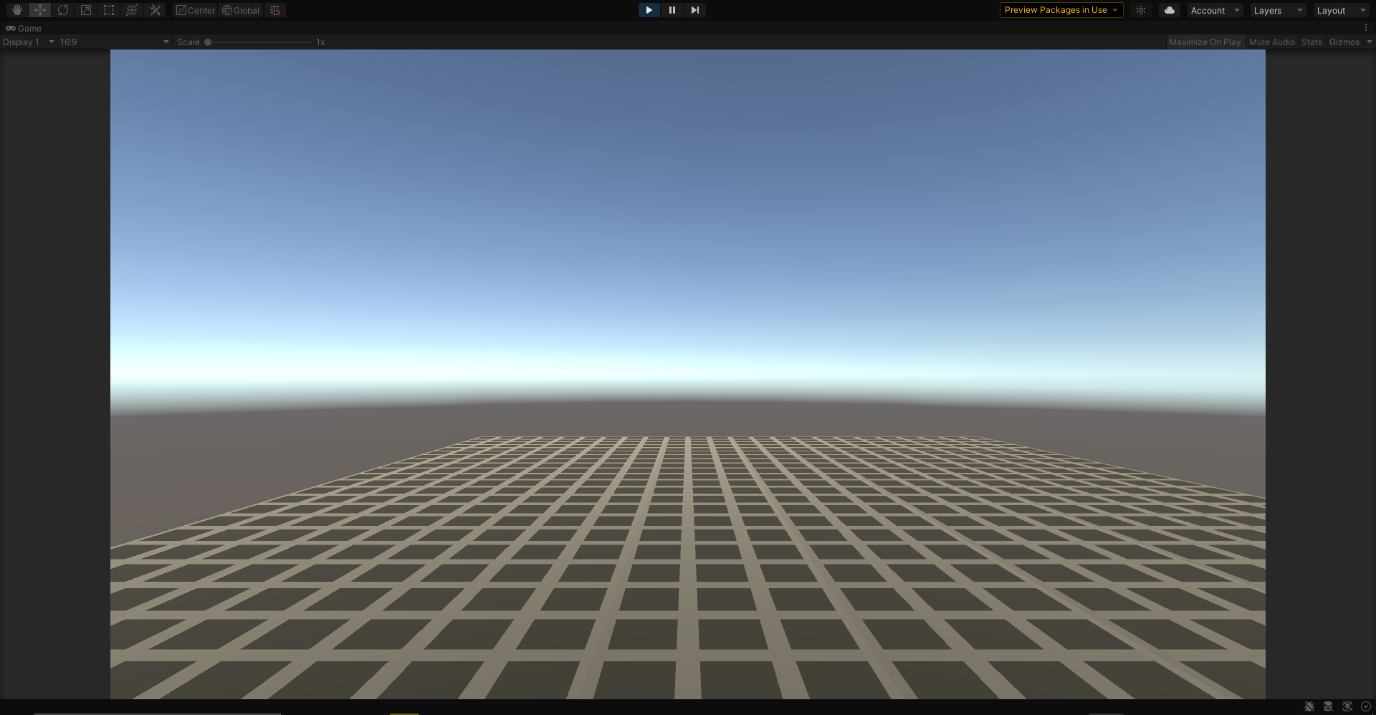
|  |  |
| --- | --- |
| *Name* | Link |
| snaps prototype (unity official) | <https://assetstore.unity.com/packages/3d/environments/snaps-prototype-office-137490> |
| low poly fruits | <https://assetstore.unity.com/packages/3d/props/food/low-poly-fruit-pickups-98135> |
| simple ui | <https://assetstore.unity.com/packages/2d/gui/icons/simple-free-pixel-art-styled-ui-pack-165012> |
| low poly weapons | <https://assetstore.unity.com/packages/3d/props/guns/low-poly-weapons-vol-1-151980> |
| fighter interceptor | <https://assetstore.unity.com/packages/3d/vehicles/air/fighter-interceptor-102321> |



**These are some of the assets in the official Unity Snaps prototypes.**

**Generating a Level**

In order to have a place to test and develop the solution further, it is imperative that we make a scene that can contain these things. So, the Unity Prototypes will be used to generate a building, originally.

First, I made a floor:

**This was a good learning experience to make each section fit well with each other.**

Next, I made the walls and ceiling:

**It is completely black, as I have not made the lights yet.**

**Making Lights**

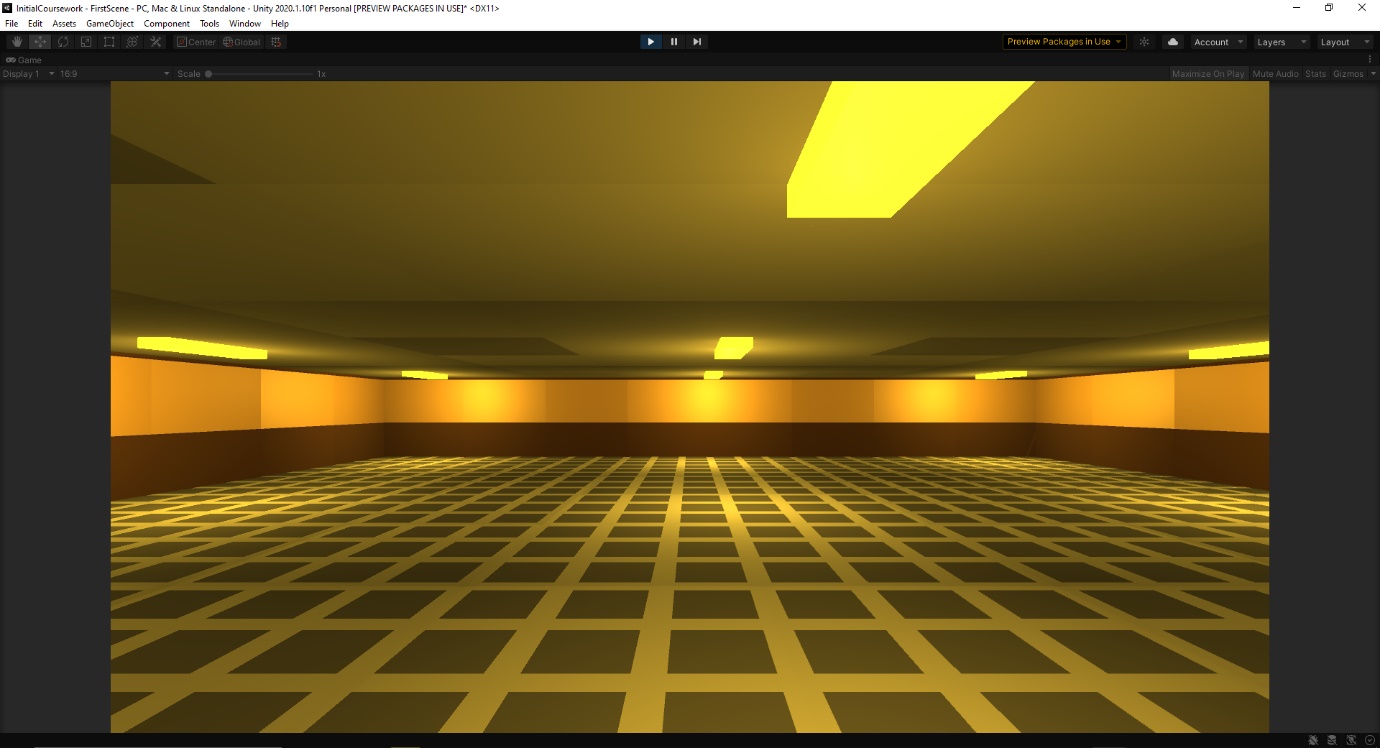
As just shown, there is a problem with having an enclosed space: there is not light!

In order to light the levels, lights must be made. These are produced specifically for this project. The lights are made of two components:

1. **A Light Object**

This is supposed to be what the player sees. This is essentially the ‘lightbulb’

1. **The Point Lights**

These objects are the things that produce light, yet they have no physical form. They can be adjusted to fir the colour of light, as well as intensity.

**The patchiness seen is fixed with post-processing + a global point light.**

All lights are prefabs, with many of them being different colours (to be used in different areas of the game).

1. <https://www.callofduty.com/modernwarfare> [↑](#footnote-ref-2)
2. <https://www.techspot.com/news/82281-call-duty-modern-warfare-system-requirements-revealed.html> [↑](#footnote-ref-3)
3. <https://store.steampowered.com/app/274900/Murder_Miners/> [↑](#footnote-ref-4)
4. <https://store.steampowered.com/app/450540/Hot_Dogs_Horseshoes__Hand_Grenades/> [↑](#footnote-ref-5)