**Design of 2.5D Side-Scroller Game Development**

CAPSTONE PROJECT PROPOSAL - CMPS 451-01

Presented to:

*The Faculty of the Computer Science Department*

*at Southern University and A&M College*

In partial fulfilment

of the requirements for the degree:

**Bachelor of Science in Computer Science**

By: Jarrel Livous

**Abstract**

In this project, we’ve chosen the pitch of creating a zombie shooter side-scroller 3D (2.5D) game developed with the unity engine. The development will take place within the unity engine overview as most of the features needed to develop the scene, the hierarchy, the project and inspector tools all reside in the unity 3D projects panel inside the unity editor. Any scripts or GUI created will be developed with the provided IDE MonoDevelop inside the unity software repository. Construction of 3D modeling will be managed between existing available assets and created with the use of AutoDesk Maya while 2D modeling will be managed in Adobe Photoshop if needed.

**2.5D Side-Scroller Game Development***Table of Contents*

Abstract -------------------------------------------------------------------------------------------------------------- 2

1. **INTRODUCTION** -------------------------------------------------------------------------------------------- 3
2. Background ------------------------------------------------------------------------------------------- 4
3. Significance ------------------------------------------------------------------------------------------- 5
4. Statement of problems ---------------------------------------------------------------------------- 6
5. Research questions --------------------------------------------------------------------------------- 7
6. Delimitations ----------------------------------------------------------------------------------------- 7
7. **REVIEW OF RELATED LITERATURE** -------------------------------------------------------------------- 8
8. **METHODOLOGY**
9. Design ------------------------------------------------------------------------------------------------- 13
10. Instrumentation ------------------------------------------------------------------------------------- 15
11. Implementation ------------------------------------------------------------------------------------- 17
12. **RESULTS AND CONCLUSIONS REFERENCES**
13. Conclusion -------------------------------------------------------------------------------------------- 24
14. References -------------------------------------------------------------------------------------------- 25
15. Source Code ------------------------------------------------------------------------------------------ 25
16. Appendix A – Definition of Terms --------------------------------------------------------------- 61

**Chapter I**

**Introduction**

* 1. **Background**

In the last decade, there has been an influx of small development studio competing with publishing labels and their subsidiary studios. This however in the earlier developments of the video game industry would not be a possible task as the devices needed to begin an ambitious objective would require large amounts of analytic power which could only be achieved using mainframe computers. Much of the results of these video games was not aimed towards user experience, but more of the developers experience and were not commercially available. The first ever attempt of developing a video game has been attributed to a physicist by the name of William Higginbotham who developed a tennis simulator on an oscilloscope very similar to “Pong” seen during the 1970s. Even after the first developed and commercially sold video game of “Pong” there was not much innovation left to differentiate or compete with other development teams as the ploy of success was seen from the flattery of imitation. Though without the pioneers of the industry there would be no foundation that gave readily-available tools and knowledge on how to begin developing video games amateurly.

Many of the success stories by recent indie or independent game developers can be seen from the ready-available game engines, software libraries and coding modules available in public domain. Software such as the open-sourced *Pygame* whose modules are designed for the development of projects utilizing python programming language. Its creation being driven in attempt to become a successor from the widely used PySDL a Simple DirectMedia Layer (SDL) cross-platform development library which gave low-level access to hardware abstraction layer via OpenGL and Direct3D. Other software such as the *LWJGL* (Lightweight Java Game Library) java software library and its binding for OpenGL (Open Graphics Library), Vulkan, OpenAL (Open Audio Library) and OpenCL (Open Computing Library) gave way to game engines and libraries such as *libGDX* and the *jMonkeyEngine*. *Unity* cross-platform game engine and its scripting APIs in C# and lastly the widely used freeware (or freemium) *Microsoft XNA* and *DirectX* and its collection of APIs based in .Net Framework allowing for cross-platform development. These success stories would not be possible without the contribution of public software sourcing and support that encourages innovation of existing or new ideas, but again without licenses such as the LGPL (GNU Lesser General Public License), BSD (Berkeley Software Distribution) Licenses, MIT (Massachusetts Institute of Technology) Licenses and CopyLeft the civil liberties of free and open-source software (FOSS) and freeware that emancipates small-time developers and studios there would be a halt or restrain.

Not to mention, digital distribution platforms such as *Steam* that caters to in-development video game projects via early access and steam direct (better known as steam greenlight) that allows consumers to have direct interaction with developers and the progression of their projects and support the developers through purchasing alpha builds and following them through the release of beta into gamma or fully-released version. *Patreon* being another contributor maintaining its platforms assistance to up-and-coming artist and developers whose community support developers through crowd-funding by managing exclusive content or experiences via subscription to those that provide money. Desura being another digital distribution platform that focuses solely on independent game developers. It also caters to community plug-ins and modification built for the game in a maintained environment via its associated website ModDB (ModDatabase) that is maintained by the same company. In addition, it provides much of the freeware and free software at no expense that can utilized by developers. Without contributions such as the named digital distribution platforms previously there would be no lane of opportunity managed for independent developers and amateur codes to design video games. These readily available tools and platforms catered to these small studios gives rise to a larger market, an overall enjoyable experience for the consumers and a competitive edge for those wishing to dive into game development and will overall continue to compel innovation.

1.2 **Significance**

An unpopular opinion about games a lot of gamer communities use is that with every advancement in technology there is a developer in mind creating an application built off that premises of potential implementing the idea into a video game. This can be viewed from many perspectives, but as fellow gamers we can concur that as a consumer we pay for a product we can indulge our own adventures into. We as a functioning society stimulate our minds and feed our demons through the motives of entertainment whether it be cinema, theatre, television, music or print. Entertainment is therapeutic in such a way that some users substitute it as an escape or an outlet from everyday life. This is in such that some individuals whom would prefer to become lost in a book would value the same in a video games in the same respect of immersion inside a video game. It could be argued to the equivalent some live a second life through the immersion of the characters and the stories concocted no matter the genre.

These realms of existence are a big part thanks to the development teams with higher development budgets and to small studios with tight budgets and that are being managed by at most one developer. The quality is not always based towards the number of features, it’s appeals visually or its premises of background. It’s the atmosphere or meta, character development, art styles and structuring, attention-to-detail and feedback that amplifies the immersion in such the player feels attached to the game’s environment, plot or the relation to the degree the development studio and its representative cater to the consumers input and not just about how much money are they willing to sink into an early-access early-build project. Without these elements there’s not much left in optimism for the video game industry and the motivation for some unfortunate peoples’ everyday outlet mechanism for everyday life.

* 1. S**tatement of Problem**

As long-time gamers we’ve been through our fair share of communities built off the premises of humble understanding and the player base and catering to the players experience via suggestions, but only to find that the genuine comradery is only a guise for an opportunity to charge for an unfinished product. In hopes of a resolution, we’ve been inspired to challenge ourselves into leaping within the gaming development field and develop our own game dedicated to zombie shooter up. The bulk of the development with take place within the unity editor overview as the features needed to develop the scene, the hierarchy, the project and inspector tools all reside in the unity 3D projects panel inside the unity editor. Any scripts or GUI created will be developed inside the provided IDE MonoDevelop inside the unity software repository. Construction of 3D modeling will be managed with existing available assets and created with the use of Blender while 2D modeling will be managed in Adobe Photoshop if needed.

Starting out, our short-term goal is to use our game as a means of entertainment for our peers and ourselves. After beginning some researching for the beneficial capabilities of unity and what it can offer into developing a zombie shooter, we’d decided it would complement our vision with its wide variety of available resources and publicly it’s been highly recommended by many indie game developers (and our own Faculty and students) as a go-to for a beginner platform. The side-scroller technique was considered after we’d given thought to the opportunity of level design, asset design, animation and script development that this technique offers with the simplicity of importing and previewing we’d probably need for trial and error. This however is not for certain the method of technique that may be in the final iteration, but as a working idea we believe it will manage perfectly for the creation we’re planning to introduce.

* 1. **Research Question**

Alongside the development, we’ve came up with a few questions to help model throughout the development cycles such as does the game reflect similar in vision to what was initially set out to witness, what would be the best hardware requirement and were the planning procedures efficient throughout the development cycle?

1.5 **Delimitations**

During the development process, we would have to into consideration the delimitations during the development process. One of which consider that all the development experience will mainly take place inside the unity engine. The number of enemies will be limited at max, 10 or 15 depending on the progression that development exceeds in the time-limit of this semester. There will only be the usage of a static assault rifle weapon type. In future iteration there may be inclusion of projectile launching weapons and lastly there will only be a limited amount of functionality placed over Enemy AI. These actions will be limited to path-finding and attacking the player.

**Chapter 2**

2.1 **Review of Related Literature**

We were reading up on things involving the capabilities of what the Unity engine provides for its developers in functionality and ran across an article written by Mike Geig. We found that the article was self-explanatory for someone whose never been familiar with how Unity and its game development works and with knowing we’d be spending the next 7-9 months creating a game inside of it we felt that this piece of information was credible to shar. He spoke about how Unity provides a more visual development experience and how it simplifies the game development workflow such the ease of access in building and modifying a project, moving assets with-in the scene or managing how they behave. Overall, to our understanding, he was conceptualizing the fact that Unity can be accolated to anyone preference and is intuitive for the average person just getting into game development.

Defining a horizontal side-scroller or platformer is a difficult task as most individual have only become familiar with household name video games such as Sonic the Hedgehog or Super Mario and peak with knowledge on the subject. We chose to share the viewpoints from the journal entry written by Luigi Di Biasi as he made simple a difficult explanation to side-scroller. In the entry, he proceeds on how this genre continues to exist by dissecting the elements that create this game style. Biasi mentions, “The first thing that characterizes a (HSSG) Horizontal Side-Scroller Game is the **scrolling**: We need to make available to the player an action that can scroll the level between start and end positions”. This can be viewed from player animation bonded to keyboard input and the visual cue of movement. It’s a self-explanatory cycle of understanding, but the task of creating a viable game with multiple keyboard inputs or controller inputs that coordinate. As he mentions, “second goal will be the implementation of the player runtime, an object that will handle all the player actions in the level”. This is what is seen as the playable character, the player or the visual representation of movement. This can be achieved through a sprite sheet or in our case of 3D modeling, movement development or animation fx. The, “third goal”, would be the implementation of the level runtime or as he mentions, “an object that will handle all the objects in the level”. Whether its autonomous random world generation, enemy generation or object/obstacle generation it’s the game management system in develop.

In the article written by Jon Brodkin on his experience using Unity’s MonoDevelop IDE and inXile Entertainmentdevelopment studio and their experience with Unity while developing *Wasteland 2* and their future installment of *Torment: Tides of Numenera*. Brodkin mentions that, “it’s possible to create a game in Unity without writing any code, but most projects require programming chops” which in innately true, but for most functionality portions that a developer may want to manage for scenario-based event then the development of scripts to better manage functionality is a necessity. He continues to mention that Unity provides a multitude of languages to primarily code with such as C#, JavaScript or Boo inside the MonoDevelop IDE. He continues further about MonoDevelip with mention that, “The development environment runs on Mono, an open source version of the .NET Framework”. The article descends into mention on inXile projects, but the most important information regarding the inXile development relationship in Unity regards the quote he took from inXile technical director John Alvarado. He mentions, “Every game object, you can attach scripts to. You write your scripts and derive them from a certain class and model behavior, and automatically when you drag scripts onto a game object that you create in the editor it will run your script. … It’s easy to add code components to any object you create in the game, whether it be a box you just made or an animated character. It’s a very modular, object-oriented way of adding functionality to an object in the game”. This concept is exactly why many developers beseech Unity as the beginner or go-to development environment for amateur developers or studios. It’s an intuitive system through game development and code implementation.

Ellison Leão explains in a new article the process of game design in developing a video game. Leão details in step the processing the step necessary to development:

“Step 1 — Emotion, mechanic and theme. Here you will define how the game will act. You can choose from a first-person shooter, to an infinite runner. You will need to combine the desired mechanic with some emotion. Step 2 — The main character. After finishing the step 1 you will have to think about the main character of the game. This main character will rise with the proper mechanics you choose to your game and perhaps some powers. Start drawing it in a paper and give it a name and a brief history. And when we say powers, that doesn’t mean that your main character is always a super hero. The power can be anything involved with the core mechanics of the game, like the ability of stealing, or the ability of not jump so well. Step 3 — Objects. A good game has some objects who can interact with the main character. Objects can be represented as some powerup which will improve the main character abilities during the game, or some new weapon to be added into the arsenal. You will need to make sure that these objects will interact directly with the core mechanics of the game as well. It is nice to always start with the very basic types of objects. Step 4 — Obstacles. A good game also comes with some difficulty. Some obstacles along the way is great to keep the player “stuck” in the good way on the game. Like the main character, is always nice to draw all the possible game obstacles, even if they are a small rock in the path or a huge wall dividing two countries. Step 5 — Level design. This is the part when you create your game environment. Where will the game be located at? A florist, a school, a house? You can make a platform level or a room full of obstacles. Just make sure that the player will be able to try as much as objects and obstacles as he can. It’s always good to keep the balance along the levels, to not make them all easy or all very difficult. Not forgetting to draw everything you wish to make for that level. Every obstacle and objects position. Step 6 — Putting all together. After making the first 5 steps, the main point of the 6th step is trying to put all things together. Those things together will not make a final game, this is just the beginning of a possible prototype that will evolve with time.” (Leão, 2015)

We chose this excerpt out of Ellison Leão article as this detail a very self-explanatory example of what development cycles look like when game developers design a road-map, timeline or procedural steps during development. It is efficient in giving the reader a larger idea to what takes place in the mind of the developer, what is usually deemed important for accomplishment and how each step wraps up towards then in to manifest cohesive system after its been implemented.

While reading up on the development in gaming technology, we stumbled across and article written by John Blow regarding simulation in game development or the realm of realism through game development. He did a very introspective look from a humanity standpoint into gaming development through real world entities such as how the term “tunneling” comes from collision detection, “we move entities essentially by teleporting them small distances through space; if we move an entity too quickly, it passes through a solid object like a wall, unless we take extra steps to detect that situation”. He makes a believable stance to introduce this conversation, but we believe our interpretation is why we gravitated toward this article. By persisting that gaming development was merely for entertainment value and did not cut close to emulating real life experience, he makes one consider the achievement of gaming progressiveness as a humankind accomplishment, “Game code is inherently about simulating world. In early games, the simulations were simple and primitive. For a while we focused mainly on graphics, which is a simulation of how light behaves in the game world. But now we are entering a time when the portions of the simulation governing physics and AI can be more important to the end user’s quality of experience than the graphics. Since generalized AI is such an unsolved problem, nobody knows what it will look like in the future. Physics, though, we have some grasp of. Working on physics has educated us about some issues that can be generalized as pertaining to all manner of simulated time-evolving complex systems “. The small line that is mention, “Working on physics has educated us about some issues that can generalized as pertaining to all manner of simulated time-evolving complex systems” leaves a mountain of suggestions, but mostly pertaining to real life applications. Can games truly be a frontier for experimentation and can we emulate reality explicitly?

**Chapter 3**

**Methodology**

3.1 **Design**

Upon beginning a new Unity project, we’d gathered up some pre-made assets to serve as the foundation of our base platform and character model as to serve for placeholders while the core elements of the game are still being refined. We’d imported these models via the Autodesk Maya 3D Graphics application into Unity directory and begin creating sub-folders with-in the assets folder. Via the Hierarchy, we’d dropped the platform inside the scene and begun modeling the imported mesh with our desired PNG image to serve as the texture for the platform. After setting-up the platforms Box Collider, adjust its orientation in respects to the main camera and general lightening in the hierarch and adjust the size, height, color, material and illumination we’d move on to placing the character model inside the scene.

After placing the main character model inside the hierarchy and correcting the rudimentary fixes to the appearance of the model’s appearance, size and orientation we’d begun our process of managing the character model’s properties. Before anything, because this character model serves as a non-static object and possessed animation we’d go into the characters animation setting and spliced the entire animation into segments that will be used inside the game such the playerIdle and playerRun. After managing the animation, we assigned a RigidBody component to the model so that it may act accordingly in respects to the physics engine and collision detection so that it would fall accordingly. In addition, we’d added a Capsule Collider also to the character model in respects to the shape and size of the model as it will serve as the character models hitbox or surfaces of interaction, but also a preventive measure of the model from falling through the platform.

Lastly, we’d wanted to begin creating those same animations we’d spliced previously into a designated set of movements controlled by player of the character model. Upon enter Unity’s animator, we’d dragged the playerIdle and playerRun animations inside the animator’s screen and assigned a relationship between the two via float parameters. Naturally the player model idles so if the player models speed exceeds 0.01 then activate the playerRun, but if the player playerRun decreases below 0.01 then idle. So, following, we assigned a Player Controller script component to

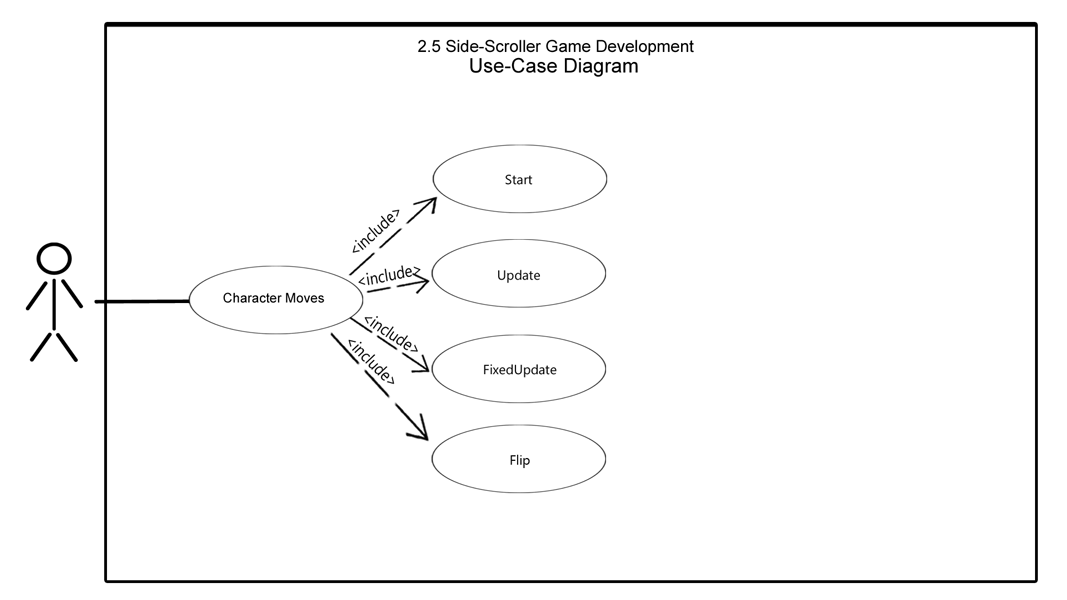
the character model to enforce the parameters prepared inside the animator.

Figure 1: The Use-Case Diagram provides a basic visual representation of the Actor’s (The users) interaction with the in game character’s movement ability in relation to what corresponds inside the playerController script.



Figure 2: In this figure, the initialization of the instance variable myRB ( relating to the RigidBody component on the player model) myAnim (referring to the Animator and its path of fluidity actor movements), float runSpeed, and the bool facingRight are assigned local scale values.

Inside MonoDevelop, we’d initialized a public variable for runSpeed that would define the rate the character and three private variables such myRB that is reference to RigidBody given to the model previously, myAnim which is about the Animator and later facingRight. Inside the Start() function we’d assigned the reference of RigidBody properties to the variable myRB via GetComponent and the same with myAnim and the Animator.

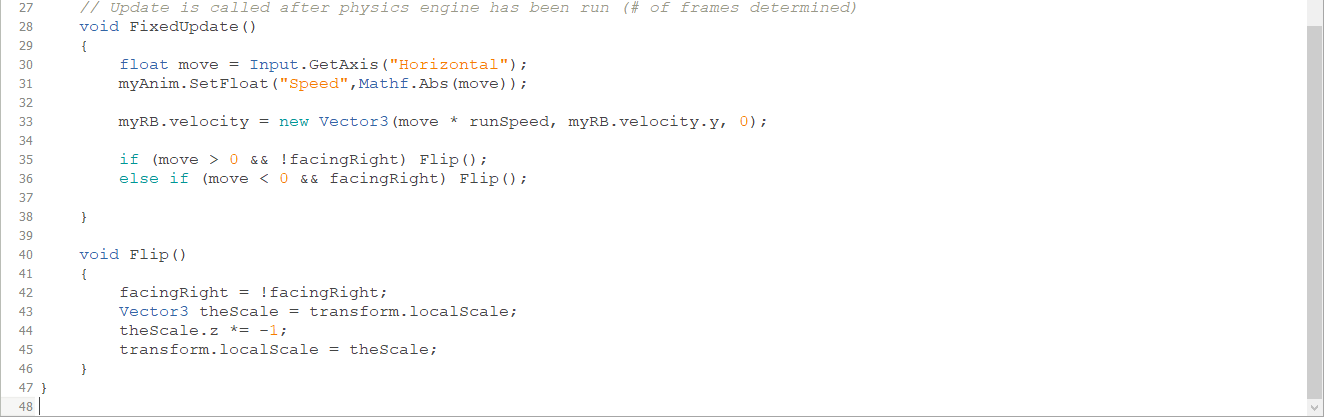
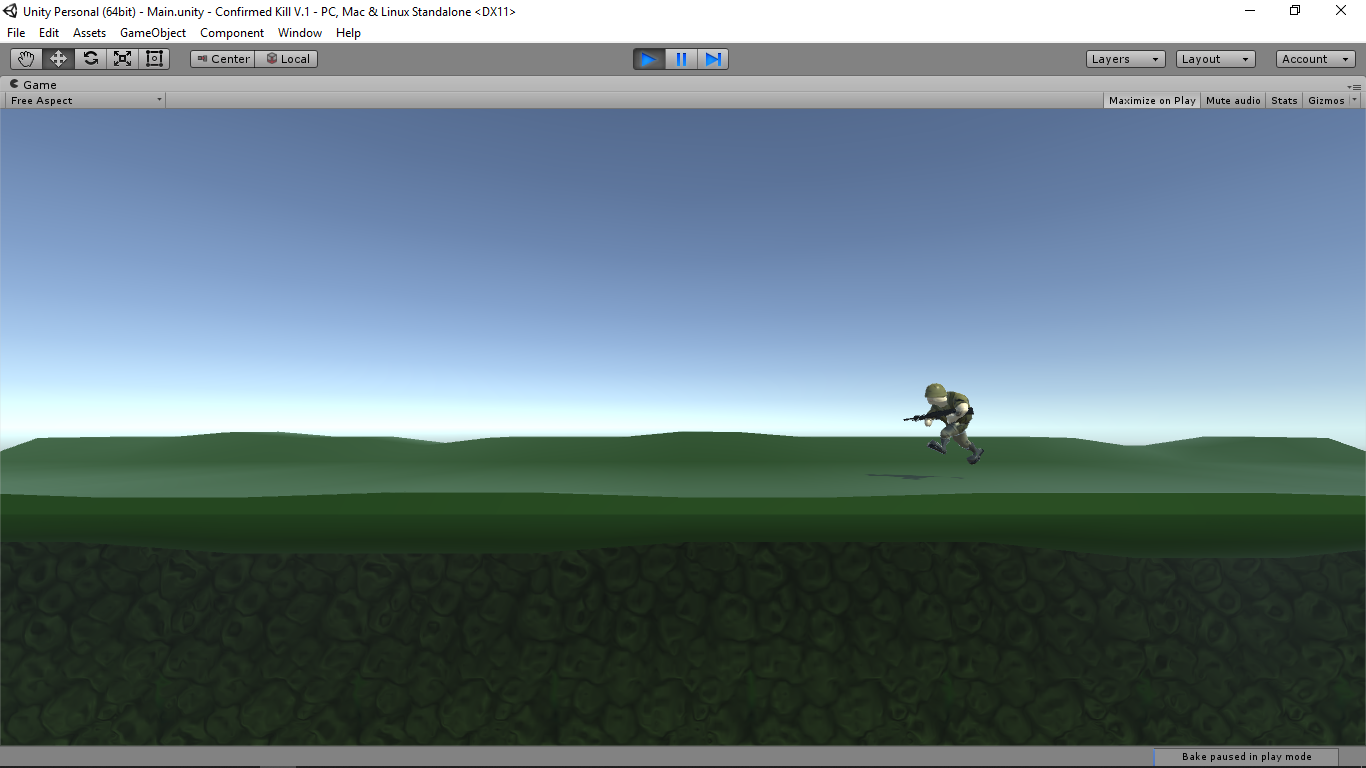


Figure 3: FixedUpdate() method takes into consideration a new vector (created from the direction, animation "runspeed" and displacement from locale) and decides inside the Flip() method (Fact: the models location is flipped, not the model itself).

We then created the function FixedUpdate() to detect when a player has pressed a button. It is called on a fixed rate after the physic engine has been run. Gathering the input from the direction of movement given from the players movement via GetAxis we’d used a predefined axis (hardcoded in Unity) associated with different keyboard keys to return to the float move as either a 1 or -1 (or 0 for idle). These values are the sent to the Animator, which was assigned to the value myAnim and due to the Mathf.Abs constraint this makes the value returning 1. To decide the velocity in the direction of which the player has decided to run we’d assigned a vector (Vector3 as it has an x,y and z component) that is determined via the move and runspeed, if the player is falling and nothing for Z assigned to the RigidBody myRB. To fix the issue of the character running in one direction the creation of the Boolean variable facingRight was made to decide to Flip() in the direction in which the player is pressing the button.

The Flip() function is used to flip the character model itself ( not to be mistaken with the Z component). The properties of localScale takes the set values of the scale made from the model and assigns it to the vector theScale. This then is multiplied by -1 to then turn the player model in the opposite direction and the new theScale is assigned back in the localScale.

**3.2 Instruments**

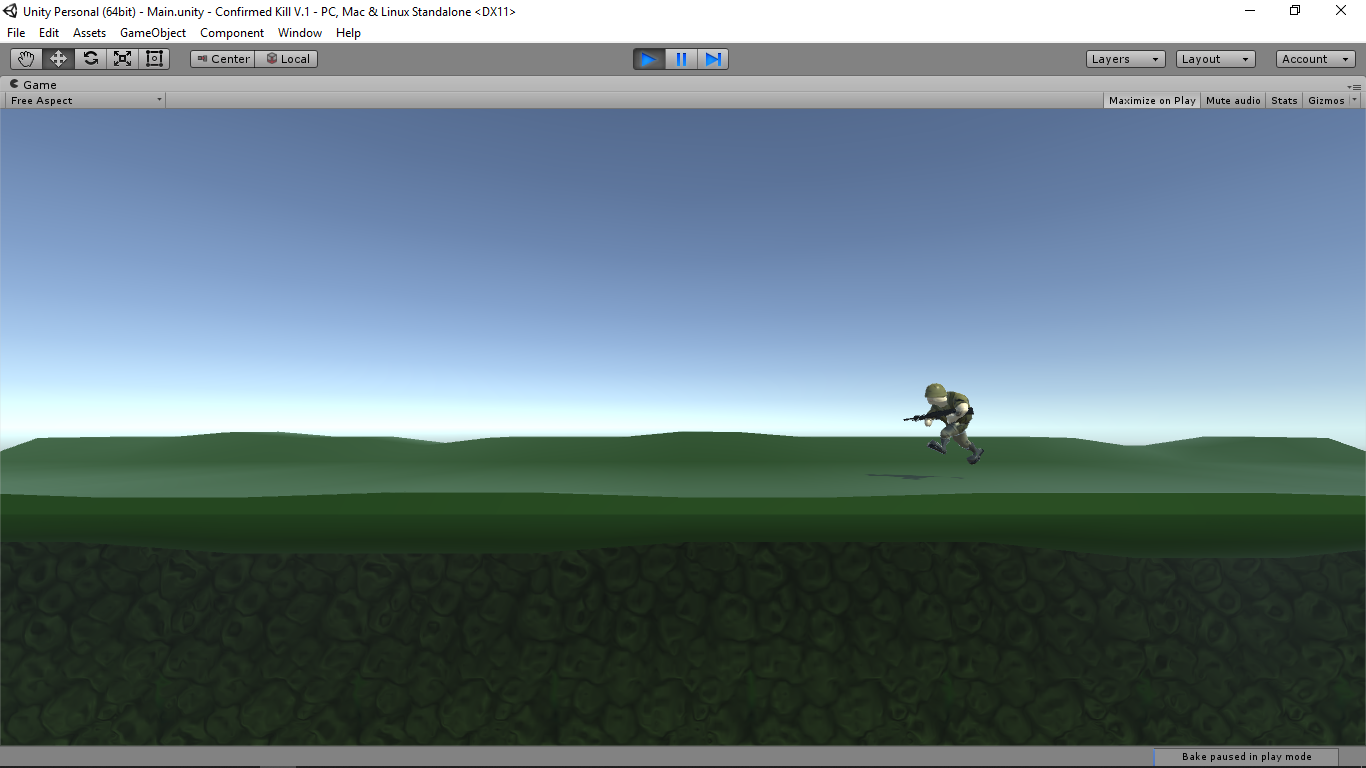
**3.2 Instruments**

Figure 4: Screenshot of the model's running animation on a platform.

For the instruments, we’ve been developing the game on a Dell Inspiron 3542 Windows 10 Home Edition OS x64 Bit System with an Installed RAM of 4.00 GB, BIOS A01, an Intel(R) Core(TM) i3-4030U CPU @ 1.90GHz, 1900 Mhz, 2 Core(s) - 4 Logical Processor(s) and 931 GB of Storage. We used AutoDesk Maya 2018 to import the models into Unity Projects and MonoDevelop to develop all the scripts.

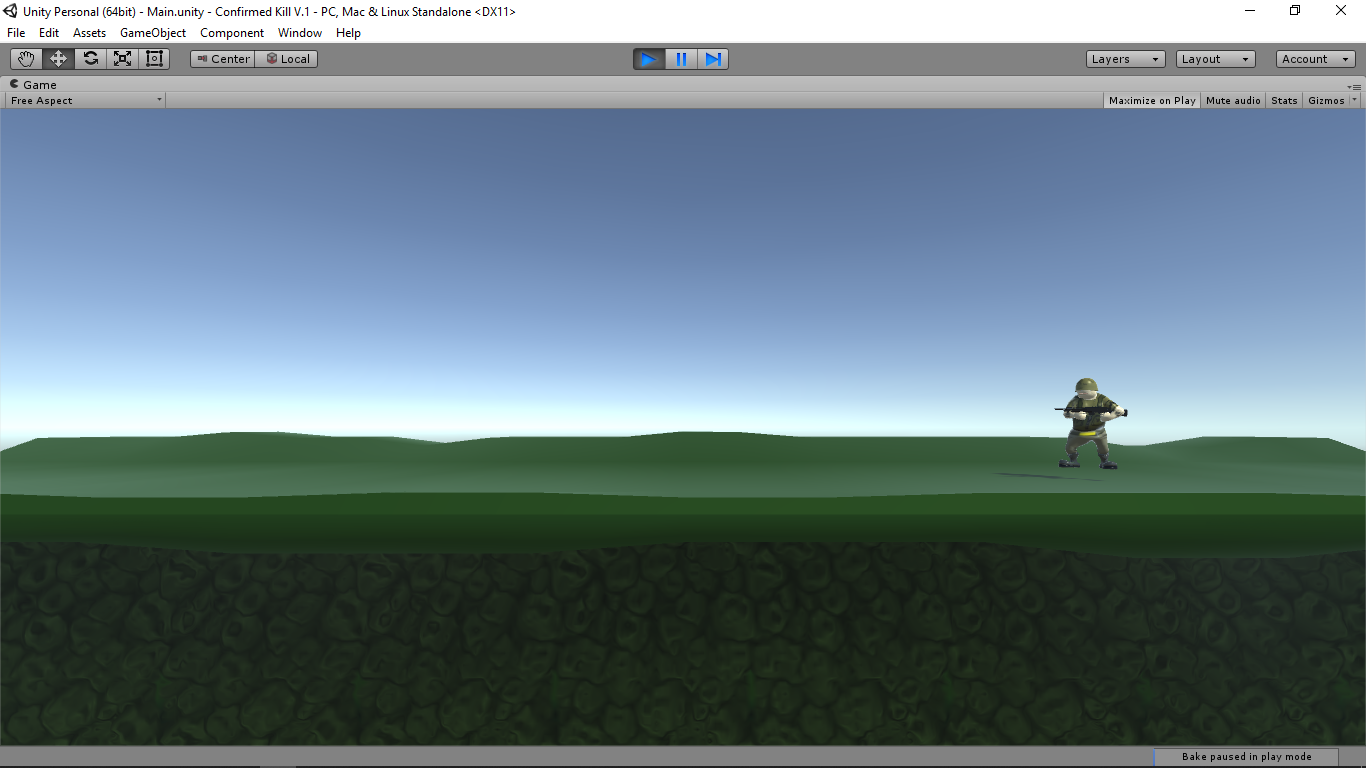


Figure 5: Screenshot of the player model's idle animation.

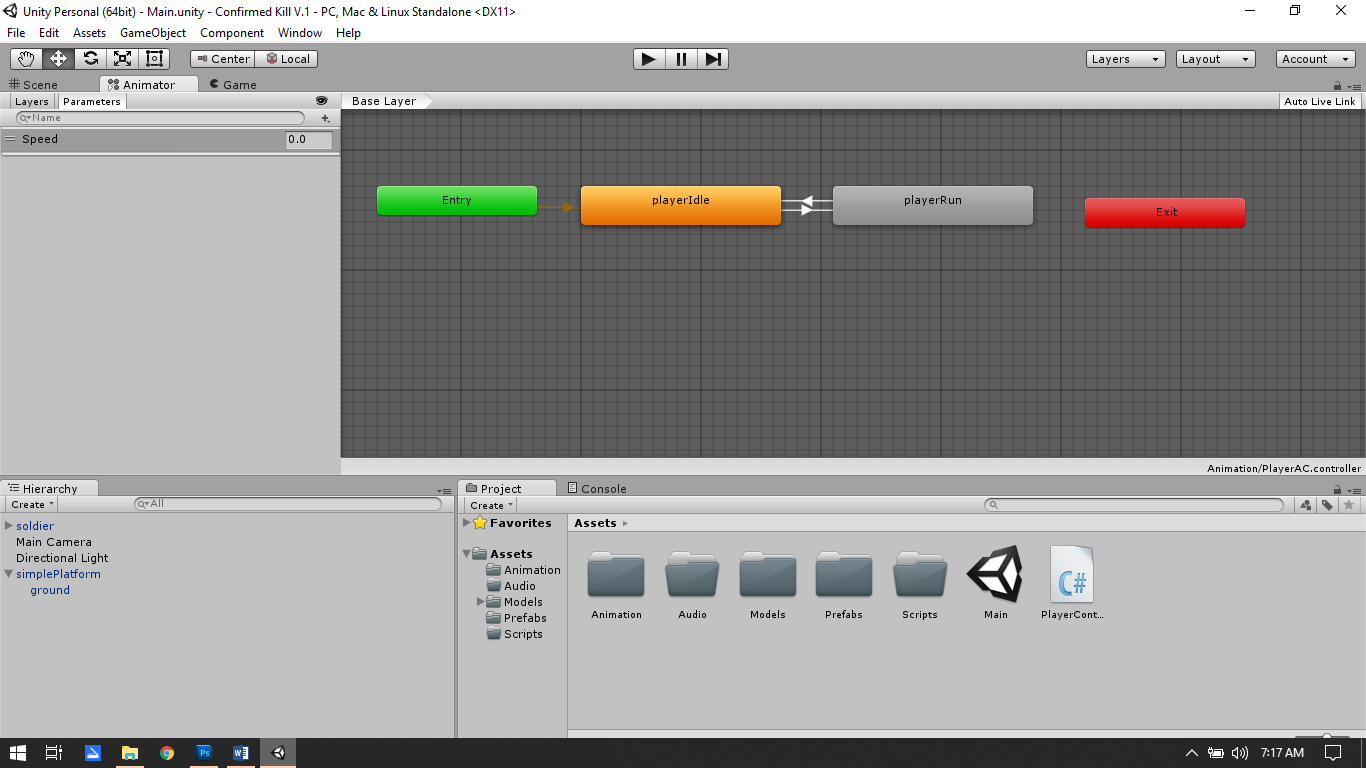


Figure 6: As labeled, this screenshot shows the hierarchy, projects folder and the base layer. What’s on the main window is the Animator where assigned movement (chopped by a loop) uses a logistic parameter to determine what player movement corresponds

**3.3 Implementation**

After the installation of both programs, Unity and Maya, we managed the first iteration of the “Assets” directory folder which would hold all the elements necessary during the developmental cycle. These folders include Animation, Models, Prefabs and Scripts which later would include Materials, Fonts, Accessories and Textures. After a quick search on unity assets store and their public forum, we’d managed to find ourselves some beginning assets being mainly an object that would acts as a platform, a character model and an enemy type. After downloading the (.zip) files, we unpacked the models and their extensions we placed them into a temporary folder to be imported. Proceeding after their download, we imported the two humanoid assets with the main proprietary file format ( .fbx - filmbox) into Maya to be properly generated. This was necessary for a proper assessment in respect towards the models rig/joints when being imported directly from the application into a unity directory “Models” through the file explorer. These two files include, the “solider” model and the “zombie” model whom both would share many similarities in animation and functionality.

Following, we’d begin the procedure of unpackaging the animations from the soldier’s import settings including segmenting the iteration of time for each animation type. This would later be necessary for placing the animation states inside the native “Animator” within the unity engine which includes the playersIdle, playerWalk, playerRun, playerJump, playerLand, playerMelee and playerNeutral. Then, the importance of a rig/camera or instance of a camera angle “Main Camera” (later changed to playerCamera) and a lighting source for visibility and dynamic lighting such as Directional Lighting for the scene became a necessity that segued to the first presentation of a management script described as cameraFollow. This script functionality in short is to update toward the position of the player and give the illusion of following the players movement at each step which is managed through smoothing the transitions. These, besides the actual script which was assigned as component to the GameObject “Main Camera”, were visible with-In the hierarchy of the scene but were not explicit inside the rendered game.

The first rendered object inside the space of the scene was the platform, later assigned simplePlatform that would also be the first “Prefab” asset. Upon generation of the asset, there would be given a transform component and a material component all of which can be assigned value. For the player and other asset model’s interaction to take place with this object was there given the component of a physics box collider. There would later be necessary a layer assignment of “ground” to the object to distinguish and modify its properties in reaction to a player or zombie types collision detection during their animation. After applying the changes, the asset was placed inside the prefab folder for duplication.

Continuing, the solider model was assigned a few colliders of its own including the physics of a RigidBody in necessity to be affected by different forces within the scene, capsule collider in necessity of a hitbox or detection zone for later usage of the health system which also doubled as its collision detected. These changes visibly made the character a credible character but needed functionality or procedural system to allocate a logic towards the movement of the character, so we proceed in fundamentally animating the solider with condition from inside the native “Animation” station. First, we added a new Animator component with possibility of an animation controller and an avatar all of which would be procedure through the Animation station and the solider model. Upon this procedure a new PlayerAnimationController (PlayerAC) was generated and ready to be assigned parameters inside the Animation station. The PlayerAC when booted had previously no state, but an entry point, but after dragging the soldier’s animation states of playersIdle, playerWalk, playerRun, playerJump, playerLand, playerMelee and playerNeutral into the station would we begin assessment of the animation controller. There would be a logic parameter of Speed, Sneaking, VerticalSpeed, Shooting, GunMelee and Grounded used in the logistic of transitioning between animation states and a blend tree modified inside the station for the properties of jumping through any state of being for the freedom of movement and smooth animation transitioning. There was then the script component of a Player Controller script added inside the solider GameObject which would instantiate a Vector3 (x, y, z) value based off the rigidbody component and model’s movement, the animation state and the direction of facing on the x-axis for a plethora of scenarios. This script would emulate the same logic given from the Animation station, but in refined detail towards player conditions and how the movement would interact throughout each animation state from the users input. This can be seen in manifestation of its reference to the player Animator in myAnim and its components. These references towards the Animator would then be passed and reapplied and given numerical value that would call upon the Animation state to input those numerical values inside of its parameters given and would use due towards the soldier’s movement.

In addition to this, there would be the implementation the solider melee ability from the Animation station and the Animator itself which also is given logistical entry from any state of existence to perform or instantiate the Melee animation. This would then be reinforced inside it’s on script, meleeScript, which references both the component of the Animator and the values recorded inside the PlayerController (as previously seen inside the playerController script used to animate the model from the corresponding inputs given by the user) in which it references a new corresponding input. This was then achieved by given the input a logic towards the consistency of input until the next iteration of the animation cycle, corresponds a numerical value of “damage”, “knockBack”, “knockBackRadius” and “meleeRate” to apply towards an enemy type. These values will then be transposed negatively to the effected GameObject via its layer assignment such as the value “enemy” or “destructible”. These same techniques would later correspond towards the players ability to shoot their weapon and would also be assigned to the same fixed point to share as hitbox for player inflicted damage.

Inside the solider class, there would need to be assigned the model assaultRifle along the solider models existing structure so that we could ultimately perform the main function of shooting a weapon. This was accomplished through first assigning the assaultRifle model onto the rig/skeleton of the solider model and aligning it in such a way it seems the solider would be carrying it. Assigning the model inside the rig to the solider prevents the Flip() function seen in the playerController helps from reversing the object in juxtaposition to the players direction and keeps them both facing the same direction. After successfully having them face the intended direction, there was a gameObject assigned called gunMuzzle that would later be the method we proceed in creating to give the illusion of a firing weapon. The material asset of BulletShader was then created to emulate the muzzle fire and shell of a bullet. Once managed, we proceeded into creating a script called gunMuzzle where the logistics of what would be the firing mechanics would be managed. Inside gunMuzzle, there would be reference towards the projectile, the estimated time between bullets or shots, the number of bullets, bullets during instantiation, the sound while shooting and reloading and late the values given from the gunMuzzle to the User Interface (UI) system to represent the number of bullets in a gun. There would also need to be a life expectancy or a life span for each bullet which would then be the development of the second script, the bulletTime. Inside the bullet Time, the created material asset of the BulletShader and visibility of what would be the generate muzzle fire, bullet stream and its impact would be generated inside this script. This would also be performed and affected by any GameObject assigned the Layer Mask of “Destructible” and GameObject assigned “Enemy”. This would later make references also towards the enemyHealth script assigned to zombies and GameObjects alike that would be affected by the players damage. There would be set a range and damage size by a numerical value of a float and be detected through the form of a transform position via the gunMuzzle to location of the target object so long as the assignment to the Layer mask correspond.

There was also the need to verify that these Bullets did not last forever on screen that we neglected originally so then began the creation on the timeToLive script. Very simple script, the only thing needed for future references for a one-size-fits-all situation did we have where on numerical value was given a time allotted to instantiated and once the time ended destroy the object.

Following the completion of the players new abilities of movement and destruction we focused towards the main enemy type that would be prevalent inside the scene which included the zombie. All the same procedures necessary for interaction with the environment and the user-controlled solider were performed the same, but before we began performing any task first we needed to create a new animator component with the same procedural generated controller that is was given to the player and assigned avatar in which the same logic would be communicated inside the Animation station and the zombie model. As mentioned previously, this produces a new ZombieAnimationController (ZombieAC) and need to be assigned a set of parameters, such as Run and Detection, inside the Animation station, but unlike the PlayerAC the zombie’s detection parameters and control would also come inside of a few more methods of components. The ZombieAC when booted had previously no state, but an entry point, but after dragging the zombie’s animation states of ZombieIdle, ZombieWalk and ZombieRun into the station would we begin assessment of the animation controller. There were few transitions made between the states, one being between the idle and two between Walk and Run, but beside those few that was all that was need in the Animator.

The major differences between the solider GameObject and the zombie GameObject is that the movement of course would not be dictated through user input, but through a detection field that was later created as the aggressionZone component. This was achieved by generating a box physics collider that would then act upon anything permeating inside its field as a trigger. This of course would then alert the zombies parameter and trigger its detection outside of its idle state which would then compel the zombie to run towards the players. The idea is that the zombie never stops searching for its target after its initial detection, so the zombie never returns to its idle state after its initial state, but it still walks towards the last detected position. Somewhat of a memory ability or if the zombie itself may have some artificial intelligence (AI). This is can be seen inside the generate script component, zombieController where the zombie’s detection and its ability to detect is given received through the form of a bool. Upon instantiation all the field are false or set to 0, but once triggered or Detected the players position is the determined and the zombie is then given the ability to run from any facing direction towards the player and after the player moves from outside the detection radius its slowed to walk and then the zombie proceeds towards the last known Vector3 position. The bool of detection is never returned to false, so the zombie will roam aimlessly in a direction until either killed or it kills itself. Its been assured only if the “Player” GameObject enters the view does it trigger the attack animation so that it references the Animator and as mentioned, move the zombie and the player is ultimately damaged the zombie as the player breaches the final physics capsule collider that is also a trigger.

After committing in the ability for detection to be done through the breaching of a collider did we begin the zombie or any enemy types ability to perform damage upon the player itself. This was also created and applied to the zombie via the enemyDamage script where values such as the damage, damageRate and the pushBack force will be used as numerical value and considered when applying towards a later PlayerHealth script. First, the playerHealth will be referenced (this was originally not applied but added later after the implementation of the health script) and the PlayerRange from that enemy type will be considered when proceeding to add damage towards the player. The solider model was also giving a capsule collider that when breach instead of causing damage would later cause the player harm so when a GameObject given the script enemyDamage proceeds through the collider and share the Layer Mask of “enemy” would it proceed to harm the player. The damage rate is determined by the Attack() method where the time take from the existing damage rate would then apply when the next damage would take place alongside how the player would be pushed back which then calls the method pushBack(). In pushBack(), it takes the players position through a Vector3, pushDirection and applies the numerical value of the pushBackForce x PushDirection and would set it to be the new value which then would be applied to the RigidBody of the player. Which then supports the idea that an enemy type whether the zombie or an environmental object can potentially harm the player so long as it applied the “enemy” Layer Mask.

Seeing through the application of the enemyDamage, we created a playerHealth script, but not necessarily for the first time. Though not mentioned previously, there had been an iteration created to emulate player health (where we focused on creating through the native special effects component a believable blood splatter through tweaking some of the given feels and a rendered red dot image into a system of splattering particles to make believable blood – this would be the playerDeathFX) for a temporary fix but was intended for testing and not intended to be the permanently solution and was ultimately scrapped. Upon the new creation of the playerHealth script did we come up with a few float numerical values that would hold the amount of health currently and the maxValue when instantiated when the game began. There would be a reference check of a bool damaged which would later be what initiates the visual cue of the UI blood splatter effect on the screen performed in part by a Color.Lerp that when utilized correctly would overtime create a bleeding effect on-screen for the player to see. The damage received in reference to the enemyDamage script in the method addDamage() would directly decrease the health pool given inside the currentHealth or fullHealth float values. These values would also later be visually represented in real time on the UI by a slider mechanic that would show a proper representation of values. In addition, adding damage to the player there would later be a system to addHealth() to a player which also given the same method of functionality, but obviously in the positive direction. There in the likely the player does pass the threshold of what’s given in the health pool will die or become dead. This is accomplished by the method isDead() in which the instantiation of the playerdeathFX, its transform position or its current position and conversion of the Vector3 is then recorded, the GameObject or player is destroyed, the visual cue is initiated and much later the losing conditions will be triggered.

We created a health pool application for the same likelihood of player-inflicted damage against the enemy combatants so thus the creation of the enemyHealth system was conceived. Many of which fields such as the enemyMaxHealth, the assignment of the damage particles (ZombieDeathFX), damageModifier, possibility of drop pickups, a UI indicator for damage and later field deathSounds was added to give a one-size-fits-all approach for any and every object seen inside the scene that could potentially harm the player. The same addDamage script seen in the playerHealth script was given to the enemyHealth script, but a certain threshold the enemy would initiate a makeDead() method which would prompt the deathSound, a ragdoll effect if the enemy was a zombie (Projected, but not executed) and would instantiate a drop if selected when destroyed and from the spot of death would the zombie also trigger the zombiedeathFX.

So, upon the major triumph of having a movable player, with a health system and an enemy player with an enemy health system we began to take in account the potential obstacle of level design and looked towards a few simple solutions. We a player falls between two platform or for instance runs straight of the platform they virtually fall forever. This also goes for other GameObject with-in the scene itself but given two options two either potentially create an invisible boundary, we chose the more applicable challenge to create a gameManager and set a boundary that destroys the GameObject. An empty GameObject was created and set underneath the two platform in fit that when the boundary or box collider is triggered there would be the instantiation of simulated death. To accomplish the destruction, we created a script we called Cleaner where of course the sole purposes is to remove an object. OnTriggerEnter() in short initiates in the playerHealth script the isDead() method and any other object is just completely destroyed.

We begin the work on the next environmental addition which would be the ammo and health pickup objects that we set to be an addition inside the bulletFire script and the playerHealth script. When creating both assets, we had already imported two mesh models without any materials so once dragged inside the hierarchy they had no functionality and no decal. When assigning a texture, one must create a material, once the material is assigned there’s going to be a need of a form of a physic collider and from there is up to the developer’s interpretation of what they want to fit their vision. We chose, for experience outside of the original import into Maya > Unity method to animate our own object in the native animator. How this work is, you open the Animation Tab, select the forms of transformation you would like to be focused on and press the record button. While moving the GameObject in whatever fashion will it now record whatever you proceed to transform inside the scene and will loop it. We chose to transform the transform the position, rotation and behavior of the halo effect so that the object may glow. This in turn creates and animator where you can mange the states between the Animation tab (Not to be mistakes with the Animation station). This method was duplicate for the ammoPickup and the healthPickup. In addition to Animator and the physics collider were two separate scripts created to make sure when these objects move and rotate inside the scene can the player interact and see and hear a difference in the status of their health and ammo count.

Inside the ammoPickupController script, the OnTriggerEnter() method calls the bulletFire script and reloads the gun for the player which completely refills the ammo capacity, initiate the gunReload sound effect and destroys the GameObject itself. Inside the healthPickupController script, the OnTriggerEnter() calls inside the playerHealth script to the method addHealth() and signifies that method to give a specific amount of health given from the healthAmount value numerical value. Then as the ammoPickupController does, the trigger sets off the sound of a health pickup and ultimately destroys the GameObject itself.

Following, we wanted to provide the visual of a variety of zombie types that manifest inside the scene so we came up with script randomZombieSkins. Using this method would allow for more than on variation to happen at random times without the need to hand place them with different skins at different times. Overall the process begins by creating 7 different zombie textures and converting them into materials. After, creating the randomZombieSkins script with help through a renderer and a created array to hold the zombieMaterials[] use the components of the created zombie Material and to reference the skinMeshRenderer and create 7 fields and allow for the selection of those 7 materials. As unity doesn’t make things easy, to allow for all 7 material to co-exist, we had to lock the inspector into place and drag all 7 skins inside the tab. Upon unlock there was an array size of 7 different zombie materials sitting inside the randomZombieSkins fields with the default zombieMaterial selected below the script and on play generated a different zombie.

Coming closer to completion I realized that there would be a need for the user to have the ability to restart the game and that would have to be under specific condition, but I wanted it to first deal with the inevitable lose. So for this I created a script that referenced the isDead() feature inside the playerHealth script and called it the restartScript. This generated a restartTime, a bool when to restartNow and float of the resetTime. In short, the method restartTheGame() is called first and resetNow is set to True, the resetTime is added to the current time and set to the new current time which is then ran through the Update() where if resetNow and the resetTime is less than or equal to the current time it reload the level. In addition, during the development of the UI did we make the option for the “escape” button and a UI selected “X” button be used to exit the application entirely.

Creating the UI was now worthy of being completed, as previously, there was a system already set into place for to check for the correction of the functionality but was not intended for the implementation. So, the necessary correction was managed previously from a system created from a two blank Canvas GameObject, playerCanvas and the featuresCanvas. In the playerCanvas, the player damage indicator or the visual cue of blood splatter mentioned previously was imported as a .JPG texture and converted as a 2D Sprite image as 2D images are necessary for the UI in unity. There is a playerInfoGroup child that holds the view of the weaponIcon, the playerHealthSlider and playerAmmoSlider and all of which use 2D sprite images to personalize. Lastly for the losing and win conditions are there and endGameText child that hold the text, “Killed in Action” and font *DoubleFeature* to prompt the user upon the completion or failure of the win or losing conditions. In the featuresCavas, there’s only the quit button used to also exit the game as mentioned in the following restartScript. There’s also the minor UI addition of the visual health slider for the zombie UI, but is incorporated inside the emenyHealth script

Lastly, after finally placing all the models and gameObjects necessary for the scene, creating the skybox and managing the lighting effects did we create the winning condition, but in an unordinary way. Upon creating our final object for the scene, we created a door on a building that would called the shelter. This door method of opening and closing the door would be realistic to the method a regular door would shut but would give visual cue and sound of completion that the user would know to be the completion of the game. Same method of animating the health and ammo pickups did we animate doors rotation only and set the idle and opening animation but allowed the opening of the door to stay in place. There was a box collider with a trigger placed at the initial entrance of the door so that when trigger by the player, the door would shut and the endGameText and win conditions will prompt “Shelter”. Inside the child of the door is a hinge, where the script component shelterDoorComponent was added. Inside the script, there has been placed an audioSource, the bool inShelter is set to false, and the restartScript and text are listed. A short synopsis of what functionality this script bring is when the player hits the trigger, OnTriggerEnter() prompts and the parameter “shelterTrigger” is triggered and the Animator begins the animation to closing the door and the sound is also initiated. The condition of the endGameText is triggered, but instead of “Killed in Action” it is overridden with “Shelter” so then the endgame conditions or the winning condition is performed, and the game is restarted.

There was also an explosion script created for the likelihood of burning barrels being implement, but was ultimately successful in production, but not in implementation due to the problem with collider detection and bulletFire to a “destructible” object creating a drop of an “explosion” and the instantiation of the explosion effecting the players health. The creation of fire that harms the player when the player reaches a specific radius, the fire special effects and smoke special effects we’re used in the visual implementation of the game, but not for its intended purposes all of which used the same method of creation as the playerDeathFX and zombieDeathFX. The creation of Ragdoll effects as mentioned previously in the ZombieController was conceived, but also created errors during runtime during the death effects of the zombie as it created a multitude of ragdolling zombies and an invincible zombie that took no damage, but still stayed animated and harmed the player. Lastly, there was a failedScript (named that after the creation of a working script) that managed the opening and closed of the shelter doors seen in the scene for the completion of the winning condition for the game but was ultimately opted out when a better solution was found.

**Chapter 4**

4.1 **Conclusion**

With the semester’s closing, we have come to the realization that this has been one of the most daunting tasks we’ve undertook, but some of the most fulfilling. As far as our research questions reflected, we believe that we have accomplished what we initially set out to witness. Every feature that has been added has been from the ground up built with the initial idea in mind and anything ambitiously sought after has been after the initial boundaries we’re satisfied.

As also questioned to what hardware this game would be best required to run on. During the build process when exporting the final alpha version of this game did we realize that Unity provides the conversion for different platforms applicable in the build menu. So there really is no best hardware for this game, it’s a platform-friendly solution to whatever version whether MacOS, WindowsOS, PS4, XBOX One, mobile IOS or mobile AndroidOS these are all totally viable options.

The planning procedures were a bit overstretched initially, many of the allocated times to our development cycle has been the inclusion of Fridays from 7 AM – 3 PM and Wednesdays (Occasionally) from 3 PM – 6 PM. Even with a set schedule, there’s been obstacles that have hemmed progress, whether a game-breaking error in run-time, a mistake on the development part, a lack of resources in the instrument department, the developer experiencing family emergencies or overall just plain becoming sick or stressed out. This has shown the true pitfalls of indie development.

For anyone considering developing a game, first time developer we would recommend try their hand with Unity and get involved in community. Not only does the forums provide an easy resource to assets not provided by the assets store, there are tips and individual willing to assist, some willing to mentor (some for a fee). There are also online videos, paid tutorials, textbooks and other resources at your disposal, but more importantly we would advise ample research on licensing and creative commons licensing. Know what you’re getting yourself into before you download and use someone’s work, because things can go bad face without proper research. Never be afraid to ask question, it comes a time we all will need somebody.

4.2 **References**

Biasi, L. D. (2016, October 31). *What is behind horizontal side-scrolling games (like Super Mario Bros).* Retrieved from Code Project: https://www.codeproject.com/Articles/1151019/What-is-behind-horizontal-side-scrolling-games-lik

Brodkin, J. (2013, June 3). *How Unity3D Became a Game-Development Beast.* Retrieved from Dice: https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/

Geig, M. (2013, April 4). *Why You Should Be Using the Unity Game Engine.* Retrieved from InformIt: http://www.informit.com/articles/article.aspx?p=2031153

Leão, E. (2015, March 3). *A brief overview on 2D Game Development.* Retrieved from Medium: https://medium.com/ellisonleao/a-brief-overview-on-2d-game-development-438927b44952

Robinson, B. (2017, November 23). *How to Grow 3D Game Development Skills With Blender 3D.* Retrieved from DZone: https://dzone.com/articles/how-to-grow-3d-game-development-skills-with-blende

Blow, J. (2004, February 24). Queue - Game Development. *Volume 1, issue 10*.

Zombie Audio – Propert of Mojang.net (“Minecraft”)  
Other Audio / Music / Player Sound(s) – Ryan Johnson (RjazzJohnson on Unity Forums)   
Font - Double Feature by Coda Gardner (Dafont.com)  
Solider / Zombie / Platform asset(s) – Ryan Johnson (RjazzJohnson on Unity Forums)  
AssaultRifle – Free asset courtesy of Unity.com  
Steel Fence Asset - Awais Amjad (awaisamjad66a of free3d.com)  
Simple Metal Fence – Vito (**ktulhu** of free3d.com)  
Tire Stack - Yusup(ysup12 of free3d.com)  
Ruin Building - ğuzhan Dağdelen(oguzhandagdelen of free3d.com)  
Building in the City - Jaka Ardian(zakardian of free3d.com)  
Old Bridge LP - Lee Snow (g3ordie of free3d.com)  
Warehouse – Oxidenio (Oxidenio of free3d.com)  
Street System V1.0 – Hago Juegos Estudios (tinchoz77 of free3d.com)  
Old House - Hassan (hassanalmalki of free3d.com)  
Urban Debris - Adithep (adithep001 of free3d.com)

4.3 **Source Code**

|  |
| --- |
| **Player Health Script** |
| using UnityEngine;  using System.Collections;  // Access to all Unity UI types  using UnityEngine.UI;  /\*  General script for players health and ability to take damage.  Special acknowledgement of contribution/credited towards:  - Ryan Johnson, RJazzJohnson of the unity forums and youtube, alumini of George Brown College in Toronto, Canada. provided contribute to HUD method and conversion of Quaternion.Euler. (None of which verbatium, but helpful tips).  \*/  public class playerHealth : MonoBehaviour  {  public float fullHealth;  float currentHealth;  public GameObject playerDeathFX;  //HUD  public Slider playerHealthSlider;  public Image damageScreen;  // Allows us to change the color - how bright the screen flashes. Full brightness or partial brighteness  Color flashColor = new Color(255f,255f,225f,1f);  // Smooting speed and indication of damage for flash  float flashSpeed = 5f;  bool damaged = false;  // End-game conditions  public Text endGameText;  public restartScript theGameController;  //Reference to audio source  AudioSource playerAS;  // Use this for initialization  void Awake ()  {  // Health does not need to start off at full health.  currentHealth = fullHealth;  //UI Value is set to fullHealth - maximum value  playerHealthSlider.maxValue = fullHealth;  playerHealthSlider.value = currentHealth;  playerAS = GetComponent<AudioSource> ();  }    // Update is called once per frame  void Update ()  {  // Are we hurt?  if (damaged)  {  damageScreen.color = flashColor;  } else  {  damageScreen.color = Color.Lerp (damageScreen.color, Color.clear, flashSpeed\*Time.deltaTime);  }  damaged = false;    }  // Any enemy or object type that harms the player will call this script.  public void addDamage(float damage)  {  currentHealth -= damage;  // indication to show players current health during run-time  playerHealthSlider.value = currentHealth;  damaged = true;  // Plays the Audio Source of AS  playerAS.Play ();  if (currentHealth <= 0)  {  isDead();  }  }  public void addHealth(int health)  {  currentHealth += health;  if (currentHealth > fullHealth)  currentHealth = fullHealth;  playerHealthSlider.value = currentHealth;  }  public void isDead()  {  Instantiate (playerDeathFX, transform.position, Quaternion.Euler (new Vector3(-90, 0, 0)));  damageScreen.color = flashColor;  Destroy (gameObject);  Animator endGameAnim = endGameText.GetComponent<Animator> ();  endGameAnim.SetTrigger ("endGame");  theGameController.restartTheGame();  }  } |
| **Player Controller Script** |
| using UnityEngine;  using System.Collections;  public class PlayerController : MonoBehaviour  /\*  Special acknowledgement of contribution/credited towards:  - Ryan Johnson, RJazzJohnson of the unity forums and youtube, alumini of George Brown College in Toronto, Canada.  provided contribute in correction toward getfacing(), fixedUpdate() and flip() method (None of which verbatium, but helpful tips).  \*/  {  // movement variable  public float runSpeed;  public float walkSpeed;  public bool running;  Rigidbody myRB;  Animator myAnim;  bool facingRight;  // for jumping  bool Grounded = false;  Collider[] groundCollisions;  float groundCheckRadius = 0.2f;  public LayerMask groundLayer;  public Transform groundCheck;  public float jumpHeight;  // Use this for initialization  void Start()  {  myRB = GetComponent<Rigidbody>();  myAnim = GetComponent<Animator>();  facingRight = true;  }    // Update is called once per frame (constant)  void Update()  {    }  // Update is called after physics engine has been run (# of frames determined)  void FixedUpdate()  {  running = false;  if (Grounded && Input.GetAxis("Jump") > 0)  {  Grounded = false;  myAnim.SetBool ("Grounded", Grounded);  myRB.velocity = new Vector3(myRB.velocity.x, 0, 0);  myRB.AddForce(new Vector3(0, jumpHeight, 0));  }    groundCollisions = Physics.OverlapSphere (groundCheck.position, groundCheckRadius, groundLayer);  if (groundCollisions.Length > 0)  Grounded = true;  else  Grounded = false;  myAnim.SetBool ("Grounded", Grounded);  //jumping animation blending  myAnim.SetFloat ("VerticalSpeed", myRB.velocity.y);  float move = Input.GetAxis("Horizontal");  myAnim.SetFloat("Speed",Mathf.Abs(move));  float Sneaking = Input.GetAxisRaw ("Fire3");  // Coordinated to the left-shift button  myAnim.SetFloat ("Sneaking", Sneaking);  float firing = Input.GetAxis("Fire1");  myAnim.SetFloat("Shooting", firing);    if (Sneaking > 0 || firing > 0 && Grounded )  {  myRB.velocity = new Vector3 (move \* walkSpeed, myRB.velocity.y, 0);  } else {  myRB.velocity = new Vector3 (move \* runSpeed, myRB.velocity.y, 0);  if (Mathf.Abs(move) > 0) running = true;  }  if (move > 0 && !facingRight) Flip();  else if (move < 0 && facingRight) Flip();  }  void Flip()  {  facingRight = !facingRight;  Vector3 theScale = transform.localScale;  theScale.z \*= -1;  transform.localScale = theScale;  }  public float GetFacing()  {  if(facingRight) return 1;  else return -1;  }  public bool GetRunning()  {  return (running);  }  } |
| **Zombie Controller Script** |
| using UnityEngine;  using System.Collections;  public class zombieController : MonoBehaviour  {  public GameObject flipModel; // Portion of the zombie model to flip direction  // Audio Options - Array of zombie sounds  public AudioClip[] idleSounds;  public float idleSoundTime; // How often a zombie can make an idle sound  AudioSource zombieMovementAS;  public GameObject ragdollDead;  float nextIdleSound = 0f; // 0f so that it occurs immediatetly  public float detectionTime; // How much time between when the player is detected  float startRun; // When the zombie begins to run  bool firstDetection; // Once detected - the player is always detected, but the decision between animation is based on this.  // Movement Option - How fast the zombie can run.  public float runSpeed;  public float walkSpeed;  public bool facingRight = true; // Zombies are not generated (They're placed facing right) so they have to be mapped to cycle direction.  float moveSpeed;  bool running;  Rigidbody myRB;  Animator myAnim;  Transform detectedPlayer; // Save the position in relation to the zombie.  bool Detected; // True or False - player is detected  // Use this for initialization  void Start ()  {  myRB = GetComponentInParent<Rigidbody> ();  myAnim = GetComponentInParent<Animator> ();  zombieMovementAS = GetComponent<AudioSource> ();  running = false; // Intial movmemnt will not be running;  Detected = false;  firstDetection = false;  moveSpeed = walkSpeed; // Changes depending what state detection is in.  if (Random.Range(0, 10) > 5)  {  Flip();  }  }  // Physics Object  void FixedUpdate ()  {  if (Detected)  {  if(detectedPlayer.position.x < transform.position.x && facingRight)  {  Flip ();  }  else if (detectedPlayer.position.x > transform.position.x && !facingRight)  {  Flip();  }  }  if (!firstDetection)  {  startRun = Time.time + detectionTime;  firstDetection = true;  }  if (Detected && !facingRight)  {  myRB.velocity = new Vector3((moveSpeed \*- 1), myRB.velocity.y, 0); // Changes X, Gravity effects Y and Z is not changed.  }  else if (Detected && facingRight)  {  myRB.velocity = new Vector3(moveSpeed, myRB.velocity.y, 0);  }  // Gets the zombie moving if the player is detected  if(!running && Detected)  {  if(startRun < Time.time)  {  moveSpeed = runSpeed;  myAnim.SetTrigger("Run");  running = true;  }  }  // idle or walking sounds, cycles through an array of idle noises provided the zombie is not running  // Method Provided by: RJazzJohnson  if(!running)  {  if(Random.Range (0, 10) > 5 && nextIdleSound < Time.time)  {  AudioClip tempClip = idleSounds[Random.Range (0, idleSounds.Length)];  zombieMovementAS.clip = tempClip;  zombieMovementAS.Play ();  nextIdleSound = idleSoundTime + Time.time;  }  }  }  void OnTriggerEnter (Collider other)  {  if (other.tag == "Player" && !Detected)  {  Detected = true; // detects player;  detectedPlayer = other.transform; // finds players position and direction.  myAnim.SetBool("Detection", Detected); // Moves from animation idle to walking after direction.  /\* Players position vs Zombies Position and is not facing current direction - it will flip the zombie to face the player. \*/  if(detectedPlayer.position.x < transform.position.x && facingRight)  {  Flip ();  } else if (detectedPlayer.position.x > transform.position.x && !facingRight)  {  Flip();  }  }  }  void OnTriggerExit(Collider other)  {  if (other.tag == "Player")  {  firstDetection = false;  }  if (running)  {  myAnim.SetTrigger("Run");  moveSpeed = walkSpeed;  running = false;  }  }  /\* Intially flips the Zombie model left or right once (random direction) based off the "Z" direction of the zombie model  This is achieved by multiplying the scale by -1 and reassigning the new direction with the "-Z". Same as the playerController.\*/  void Flip()  {  facingRight = !facingRight;  Vector3 theScale = flipModel.transform.localScale;  theScale.z \*=-1;  flipModel.transform.localScale = theScale;  }  /\*  \* Having a hard time instantiating this script correctly --  \*  \* \*/  public void ragdollDeath()  {  GameObject ragDoll = Instantiate (ragdollDead, transform.root.transform.position, Quaternion.identity) as GameObject;  // Find the master of the root node of the rig of the ragdoll and the zombie - syncs the orientation.  Transform ragDollMaster = ragDoll.transform.Find ("master");  Transform zombieMaster = transform.root.Find ("master");  // Working around issue that comes with Flip function.  bool wasFacingRight = true;  if (!facingRight)  {  wasFacingRight = false;  Flip ();  }  // Find tranforms of the joint of the ragdoll & current zombie  Transform[] ragdollJoints = ragDollMaster.GetComponentsInChildren<Transform> ();  Transform[] zombieJoints = zombieMaster.GetComponentsInChildren<Transform> ();  // Iterates through all joints and set them to 0.  for (int i = 0; i < ragdollJoints.Length; i++)  {  for (int q = 0; q < zombieJoints.Length; q++)  {  if(zombieJoints[q].name.CompareTo(ragdollJoints[i].name) == 0)  {  ragdollJoints[i].position = zombieJoints[q].position;  ragdollJoints[i].rotation = zombieJoints[q].rotation;  break;  }  }  }  // Making rotations of zombie and ragdoll the same.  if (wasFacingRight)  {  Vector3 rotVector = new Vector3 (0, 0, 0);  ragDoll.transform.rotation = Quaternion.Euler(rotVector);  }  else  {  Vector3 rotVector = new Vector3(0, 90, 0);  ragDoll.transform.rotation = Quaternion.Euler(rotVector);  }  // Matching the randomiztion of zombie material/mesh information  Transform zombieMesh = transform.root.transform.Find ("zombieSoilder");  Transform ragdollMesh = ragDoll.transform.Find ("zombieSoilder");  /\* Theres an occuring issue where - when the player kills the zombie and it attempts to match the same skin - it instantiates.  \* A new zombie in its place which has no HP and still pursues and kills the player while dropping ragdolls. \*/  // ragdollMesh.GetComponent<Renderer>().material = zombieMesh.GetComponent<Renderer>().material;  }  } |
| **Random Zombie Skin Script** |
| using UnityEngine;  using System.Collections;  public class randomZSkin : MonoBehaviour  /\*  General script for zombie skin/material randomization  Special acknowledgement of contribution/credited towards:  - Ryan Johnson, RJazzJohnson of the unity forums and youtube, alumini of George Brown College in Toronto, Canada.  All of which the functionality has been provided verbatium from his provided his script besides naming.  The conversion from language Java to C# all mananged from myself.  \*/  {  /\* Randomizes the textures of zombie enemies.\*/  public Material[] zombieMaterials;  // Use this for initialization  void Start ()  {  // Assigns a particular material to a zombie.  SkinnedMeshRenderer myRenderer = GetComponent<SkinnedMeshRenderer>();  myRenderer.material = zombieMaterials[Random.Range(0, zombieMaterials.Length)];  }    // Update is called once per frame  void Update ()  {    }  } |
| **Enemy Health Script** |
| using UnityEngine;  using System.Collections;  using UnityEngine.UI;  /\* Script access the player canvas UI health bar and interactables \*/  public class enemyHealth : MonoBehaviour  {  public float enemyMAXHealth;  public float damageModifier; // Multiplier  public GameObject damageParticles;  public bool drops;  public GameObject drop; // Reward for killing the enemy  public AudioClip deathSounds;  float currentHealth;  public Slider enemyHealthIndicator;  AudioSource enemyAS;  // Use this for initialization  void Awake ()  {  currentHealth = enemyMAXHealth;  enemyHealthIndicator.maxValue = enemyMAXHealth;  enemyHealthIndicator.value = currentHealth;  enemyAS = GetComponent<AudioSource> ();  }    // Update is called once per frame  void Update ()  {    }  /\* Public access to the enemies current health. Any damaging interaction will subtract to the enemies health and visually by the GUI  Prevents multiple current health bars from being displayed at a time\*/  public void addDamage(float damage)  {  enemyHealthIndicator.gameObject.SetActive(true); // Turns on the visibility of the game object;  damage = damage \* damageModifier;  if (damage <= 0)  return;  // currentHealth = currentHealth – damage;  currentHealth -= damage;  enemyHealthIndicator.value = currentHealth;  enemyAS.Play ();  if (currentHealth <= 0)  makeDead();  }  void makeDead()  {  // turn off movement & create ragdoll  // Destorys entire hierarchy of zombie  /\* Instantiate the zombie model to the zombie ragdoll mob as closesly to realistically as possible.  Found from the ZombieController flip() method.\*/  zombieController aZombie = GetComponentInChildren<zombieController> ();  if (aZombie != null)  {  aZombie.ragdollDeath();  }  AudioSource.PlayClipAtPoint(deathSounds, transform.position, 0.15f);  Destroy (gameObject.transform.root.gameObject);  if (drops) Instantiate (drop, transform.position, transform.rotation);  }  public void damageFX (Vector3 point, Vector3 rotation)  {  Instantiate (damageParticles, point, Quaternion.Euler (rotation));  }  } |
| **Enemy Damage Script** |
| using UnityEngine;  using System.Collections;  public class enemyDamage : MonoBehaviour  {  public float damage;  public float damageRate;  public float pushBackForce;  float nextDamage;  bool playerInRange = false;  GameObject thePlayer;  playerHealth thePlayerHealth;  // Use this for initialization  void Start()  {  nextDamage = Time.time;  thePlayer = GameObject.FindGameObjectWithTag("Player");  thePlayerHealth = thePlayer.GetComponent<playerHealth>();  }    // Update is called once per frame  void Update()  {  if (playerInRange) Attack();  }  void OnTriggerEnter (Collider other)  {  if (other.tag == "Player")  {  playerInRange = true;  }  }  void OnTriggerExit(Collider other)  {  if (other.tag == "Player")  {  playerInRange = false;  }    }  void Attack()  {  if (nextDamage <= Time.time)  {  thePlayerHealth.addDamage(damage);  nextDamage = Time.time + damageRate;  pushBack (thePlayer.transform);  }  }  void pushBack(Transform pushedObject)  {  Vector3 pushDirection = new Vector3(0, (pushedObject.position.y - transform.position.y), 0).normalized;  pushDirection\*= pushBackForce;  Rigidbody pushedRB = pushedObject.GetComponent<Rigidbody>();  pushedRB.velocity = Vector3.zero;  pushedRB.AddForce(pushDirection, ForceMode.Impulse);  }  } |
| **Bullet / Muzzle Fire Script** |
| using UnityEngine;  using System.Collections;  using UnityEngine.UI;  public class BulletFire : MonoBehaviour  {  // f is used to declare a float  public float timeBetweenBullets = 0.15f;  public GameObject projectile;  /\* Bullet Info - maxRound max bullets a weapon can carry.\*/  public int maxRounds;  public int startingRounds;  public Slider playerAmmoSlider;  int remainingRounds;  float nextBullet;  // Audio Info - This leave the dependency of the weapon to decide between which weapon sound will be prioritized.  AudioSource gunMuzzleAS;  public AudioClip shootingSound;  public AudioClip reloadSound;  // Use this for initialization  void Awake ()  {  nextBullet = 0f;  remainingRounds = startingRounds;  playerAmmoSlider.maxValue = maxRounds;  playerAmmoSlider.value = remainingRounds;  gunMuzzleAS = GetComponent<AudioSource> ();  }    // Update is called once per frame  void Update ()  {  // Directing a route for the gun moozle through the root of the directory of the player  PlayerController myPlayer = transform.root.GetComponent<PlayerController>();  /\* if the person is actually firing the weapon and the next bullet has surpassed it time to live  \* then the next bullet will be allowed to be shot. The Vector3 corrects an issue when the player is flipped the  \* direction of the bullets projectile is misdirected. \*/  if (Input.GetAxisRaw("Fire1") > 0 && nextBullet < Time.time && remainingRounds > 0)  {  nextBullet = Time.time + timeBetweenBullets;  // rot = rotation  Vector3 rot;  if (myPlayer.GetFacing() == -1f)  {  rot = new Vector3(0, -90, 0);  } else  rot = new Vector3(0, 90, 0);  /\* Quaternion surface level allows the axis of rotation to not become locked up during the gameplay  \* and is converted to Euler - there's a larger understanding beyond the implementation in the code.  \* Hint: It's a mathematical coordinate system \*/  Instantiate(projectile, transform.position, Quaternion.Euler(rot));  /\*  // Everytime you pull the trigger, a sound is produced.  gunMuzzleAS.clip = shootingSound;  gunMuzzleAS.Play();  \*/  // Simpler fix that prioritizes a function rather than producing one sound in specific to the action.  playASound (shootingSound);  // Players rounds is represented accurately by the AmmoSlider.  remainingRounds -= 1;  playerAmmoSlider.value = remainingRounds;  }  }  public void reload()  {  remainingRounds = maxRounds;  playerAmmoSlider.value = remainingRounds;  /\*  // Everytime you reload your weapon, a sound is produced.  gunMuzzleAS.clip = shootingSound;  gunMuzzleAS.Play();  \*/  // Simpler fix that prioritizes a function rather than producing one sound in specific to the action.  playASound (reloadSound);  }  void playASound(AudioClip playTheSound)  {  gunMuzzleAS.clip = playTheSound;  gunMuzzleAS.Play();  }  } |
| **Bullet Time Script** |
| using UnityEngine;  using System.Collections;  public class BulletTime : MonoBehaviour  {  public float range = 10f;  public float damage = 5f;  Ray bulletRay;  RaycastHit bulletImpact;  int bulletMask;  LineRenderer bulletStream;  // Use this for initialization  void Awake ()  {  bulletMask = LayerMask.GetMask(“Destructible”);  bulletStream = GetComponent<LineRenderer>();  bulletRay.origin = transform.position;  bulletRay.direction = transform.forward;  bulletStream.SetPosition(0, transform.position);  if(Physics.Raycast(bulletRay, out bulletImpact, range, bulletMask))  {  // enemy damage goes here  if(bulletImpact.collider.tag == “Enemy”)  {  enemyHealth theEnemyHealth = bulletImpact.collider.GetComponent<enemyHealth>();  if(theEnemyHealth != null)  {  theEnemyHealth.addDamage(damage);  theEnemyHealth.damageFX(bulletImpact.point, -bulletRay.direction);  }  }  bulletStream.SetPosition(1, bulletImpact.point);  } else bulletStream.SetPosition(1, bulletRay.origin+bulletRay.direction\*range);  }    // Update is called once per frame  void Update ()  {    }  } |
| **Following Camera Script** |
| using UnityEngine;  using System.Collections;  public class CameraFollow : MonoBehaviour  {  public Transform target;  public float smoothing = 5f;  Vector3 offset;  // Use this for initialization  void Start ()  {  offset = transform.position - target.position;    }    // Update is called once per frame  void FixedUpdate ()  {  Vector3 targetCamPos = target.position + offset;  transform.position = Vector3.Lerp (transform.position, targetCamPos, smoothing \* Time.deltaTime);    }  } |
| **GameManager / Cleaner / GameObject removal script** |
| using UnityEngine;  using System.Collections;  public class Cleaner : MonoBehaviour  {  // Use this for initialization  void Start ()  {    }    // Update is called once per frame  void Update ()  {    }  // if player fall into box collider, destroy the player.  // prevents player from falling forever.  void OnTriggerEnter(Collider other)  {  if (other.tag == "Player")  {  playerHealth playerDead = other.gameObject.GetComponent<playerHealth>();  playerDead.isDead();  } else Destroy (other.transform.root.gameObject);  }  } |
| **Ammo Pickup Script** |
| using UnityEngine;  using System.Collections;  public class ammoPickupController : MonoBehaviour  {  // Use this for initialization  void Start ()  {    }    // Update is called once per frame  void Update ()  {    }  void OnTriggerEnter (Collider other)  {  if(other.tag == "Player")  {  other.GetComponentInChildren<BulletFire>().reload();  Destroy(transform.root.gameObject);  }  }  } |
| **Health Pickup Script** |
| using UnityEngine;  using System.Collections;  public class ammoPickupController : MonoBehaviour  {  // Use this for initialization  void Start ()  {    }    // Update is called once per frame  void Update ()  {    }  void OnTriggerEnter (Collider other)  {  if(other.tag == "Player")  {  other.GetComponentInChildren<BulletFire>().reload();  Destroy(transform.root.gameObject);  }  }  } |
| **Player Melee Script** |
| using UnityEngine;  using System.Collections;  public class meleeScript : MonoBehaviour  {  public float damage;  public float knockBack;  public float knockBackRadius;  public float meleeRate;  float nextMelee;  int bulletMask;  Animator myAnim;  PlayerController myPC;  // Use this for initialization  void Start ()  {  bulletMask = LayerMask.GetMask ("Destructible");  myAnim = transform.root.GetComponent<Animator> ();  myPC = transform.root.GetComponent<PlayerController> ();  nextMelee = 0f;  }    // Update is called once per frame  void FixedUpdate ()  {  float melee = Input.GetAxis ("Fire2");  if (melee > 0 && nextMelee < Time.time && !(myPC.GetRunning()))  {  myAnim.SetTrigger("GunMelee");  nextMelee = Time.time + meleeRate;  // do damage - Return a list of objects that collides with the sphere.  Collider[] attacked = Physics.OverlapSphere(transform.position, knockBackRadius, bulletMask);  // Loops through attack colliders - the length of the attack is based of the enemy, the addDamage and the enemyHealth script.  // The effect are in regards to the enemies transform and position.  int i=0;  while(i < attacked.Length)  if(attacked[i].tag == "Enemy")  {  enemyHealth doDamage = attacked[i].GetComponent<enemyHealth>();  doDamage.addDamage(damage);  doDamage.damageFX(transform.position, transform.localEulerAngles);  }  }  }  } |
| **Explosion Controller Script** |
| using UnityEngine;  using System.Collections;  public class explosionController : MonoBehaviour  {  public Light explosionLight;  public float power; // Explosive Power  public float radius; // Radius of the explosion  public float damage; // Damage the explosion will cause  // Use this for initialization  void Start ()  {  Vector3 explosionPosition = transform.position;  Collider[] colliders = Physics.OverlapSphere (explosionPosition, radius);  foreach (Collider hit in colliders)  {  Rigidbody RB = hit.GetComponent<Rigidbody>(); // Looking for a RigidBody  if(RB != null) RB.AddExplosionForce(power, explosionPosition, radius, 3.0f, ForceMode.Impulse);  // Apply a force to a rigidbody inside the radius with Force Impulse  if(hit.tag == "Player")  {  playerHealth thePlayerHealth = hit.gameObject.GetComponent<playerHealth>();  thePlayerHealth.addDamage(damage);  } else if (hit.tag == "Enemy")  {  enemyHealth theEnemyHealth = hit.gameObject.GetComponent<enemyHealth>();  theEnemyHealth.addDamage(damage);  }  }  }    // Update is called once per frame  void Update ()  {  // Light goes from full intensity then decreases overtime.  explosionLight.intensity = Mathf.Lerp (explosionLight.intensity, 0f, 5 \* Time.time);  }  } |
| **Game Restart Script** |
|  |
| using UnityEngine;  using System.Collections;  public class restartScript : MonoBehaviour  {  public float restartTime;  bool resetNow = false;  float resetTime;  // Use this for initialization  void Start () {    }    // Update is called once per frame  void Update ()  {  if (resetNow && resetTime <= Time.time)  {  Application.LoadLevel(Application.loadedLevel);  }  if (Input.GetKey("escape"))  {  Application.Quit();  }  }  public void restartTheGame()  {  resetNow = true;  resetTime = restartTime + Time.time;  }  // Ability to close the application  public void quitGame()  {  Application.Quit();  }  } |
| **Shelter Door Script** |
| using UnityEngine;  using System.Collections;  using UnityEngine.UI;  public class shelterDoorController : MonoBehaviour {  AudioSource shelterDoorAS;  bool inShelter = false;  public Text endGameText;  public restartScript theGameController;  // Use this for initialization  void Start ()  {  shelterDoorAS = GetComponent<AudioSource> ();  }    // Update is called once per frame  void Update ()  {    }  // Prevents enemies from triggering door. inShelter prevents a player from triggering endGameText more than once.  void OnTriggerEnter(Collider other)  {  if (other.tag == "Player" && inShelter == false)  {  Animator shelterDoorAnim = GetComponentInChildren<Animator>();  shelterDoorAnim.SetTrigger("shelterTrigger");  shelterDoorAS.Play();  endGameText.text = "Found Shelter";  Animator endGameAnim = endGameText.GetComponent<Animator> ();  endGameAnim.SetTrigger ("endGame");  theGameController.restartTheGame();  inShelter = true;  }  }  } |
| **GameObject removal Script** |
| using UnityEngine;  using System.Collections;  public class TimeToLive : MonoBehaviour  {  public float aliveTime;  // Use this for initialization  void Awake ()  {  Destroy (gameObject, aliveTime);  }    // Update is called once per frame  void Update ()  {    }  } |

4.3 **Appendix – Definition of Terms**

.zip – an archive file format that supports lossless data compression.

.fbx – is a proprietary file format developed by Kaydara and owned by Autodesk since 2006. It issued to provide interoperability between digital content creation applications. FBX is also part of Autodesk Gameware, a series of video game middleware.

.mtl – (Material Library File) contains one or more material definitions, each of which includes the color, texture, and reflection map of individual materials. These are applied to the surfaces and vertices of objects.

Prefab – copy of a game object converted into a reusable asset – it shows up in the project folder and serialized as a file on disk. Prefabs can contain a hierarchy of game objects.

Collider – Components define the shape of an object for the purposes of physical collides. A collider, which is invisible, need not be the exact shape of the objects mesh.

Rigidbody - Fully simulated by the physics engine and can react to collision and forces applied from a script. They can collide with other objects (including static colliders) and are most commonly used collider configuration in games that use physics.

AI – (Artificial Intelligence) The capability of a machine to imitate human intelligence by process of an algorithm through a set of process of rules to be followed in calculation or problem-solving operations.

Lerp – (Linear Interpolation) Method of curve fitting using linear polynomials to construct new data points within the range of a discrete set of known data points.

Quaternion – A number system that extend the complex number. They were first described by Irish mathematician William Rowan Hamilton in 1843 and applied to mechanics in three-dimensional space. A feature of quaternions is that multiplication of two quaternions in noncommutative.

Euler – (Angles) Three angles introduced by Leonhard Euler to describe the orientation of a rigid body with respect to a fixed coordinate system. They can also represent the orientation of a mobile frame of reference in physics of the orientation of a general basis in 3-dimensional linear algebra.

GameObject – Fundamental objects in unity that represent characters, props or scenery. They do not accomplish much in themselves, but they act as containers for components a functional part of a GameObject.

Scenes – Contain the environment and menus of your game. Think of each unique scene file as a unique level. In each Scene, you place your environments, obstacles, and decoration, essentially designing and building your game in pieces.