Vir and the Army of Tenebrax

Interactive Media and Game Development

A Major Qualifying Project Report

submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

by Peter Lepper, Tyree Robinson, and Robert Smieja

Advisors: Mark Claypool, Brian Moriarty

Abstract

We set out to create a game that would act as a showcase of our individual abilities and talents. We designed the game based on the two different art styles of our artists. We created original 2D art and fully animated it, and coded the game in Unreal Engine 4 using Blueprints. We performed playtesting in order to evaluate our game, and ended with a well-polished project.

Table of Contents

1 Introduction 4

2 Design 5

2.1 Original Team and New Idea 5

2.2 Story 8

2.3 Gameplay 10

2.4 Level Design 11

2.5 Upgrades 13

3 Art 14

3.1 Models/Figures 14

3.2 Animation 16

3.3 Music 18

3.4 Sound Effects 19

4 Technical Implementation 20

4.1 Animation 21

4.2 Hero 23

4.3 Enemies 29

5 Evaluation 31

6 Technical Evaluation of a Scene in Unreal Engine 4 33

6.1 Platform 35

6.2 Procedure 35

6.3 Results 40

6.4 Summary 44

7 Conclusion 45

8 Works Cited 47

9 Appendix 48

9.1 Design Process and Re-Scoping 50

1 Introduction

For our MQP we set out to create a game that would showcase all of our individual strengths, our artistic abilities, our programming abilities, and provide a good portfolio piece.

In other to do this, we created a two dimensional side-scrolling platformer that included a simple combat system and a few “role playing” elements in the form of upgrades.

We combined the two different art styles of our artists in order to create four different villain characters, a fully animated main hero character with fourteen different sets of sprites, a combined total of nine original sound effects and music, and two original backgrounds.

We used Unreal Engine 4 to create the two levels of the game, including the animation system used to handle the animations and art for the enemies and the main player character, the upgrade system used in the game, the UI, and the AI used for the enemies.

For the rest of the report, we will review the design process of our game, beginning with the idea of the game, story and play mechanics, followed by the creation of the art assets and the technical implementation.

2 Design

2.1 Original Team and New Idea

We started brainstorming with the goal of making a short level for a well-polished and fun game that would serve as a good portfolio piece, showcasing art from both artists. We eventually agreed to combine the more cartoonish style of one artist with the more realistic style of the other, and to make that combination the basis of our new idea. Two art styles became two sides in a conflict; cartoonish became good, realistic became evil. Two worlds clashing, one invading the other.

We self-imposed several measures to deal with our limited time constraint. We would make 2D art instead of 3D, there would be no procedural generation in the level design, our game would only end up running for about five minutes, and contain a minimal set of features in order to create a well-polished demo.

We then hashed out a basic design for *Vir and the Army of Tenebrax*, creating concept art and a platforming tech demo. We settled on three basic minion types; the warrior, slow but powerful with heavy armor and a sword; the brawler, fast and aggressive but not as durable; and the wizard, physically weak but able to conjure and throw fire.

The original concept art for the villains took some inspiration from such games as *World of Warcraft* (Blizzard, 2004) and *Guild Wars II* (NCSOFT, 2012), but with our own unique spin, as seen in Figure 1 below. Tenebrax was a high fantasy world, and the minions were supposed to be faceless and interchangeable while still being imposing and intimidating.

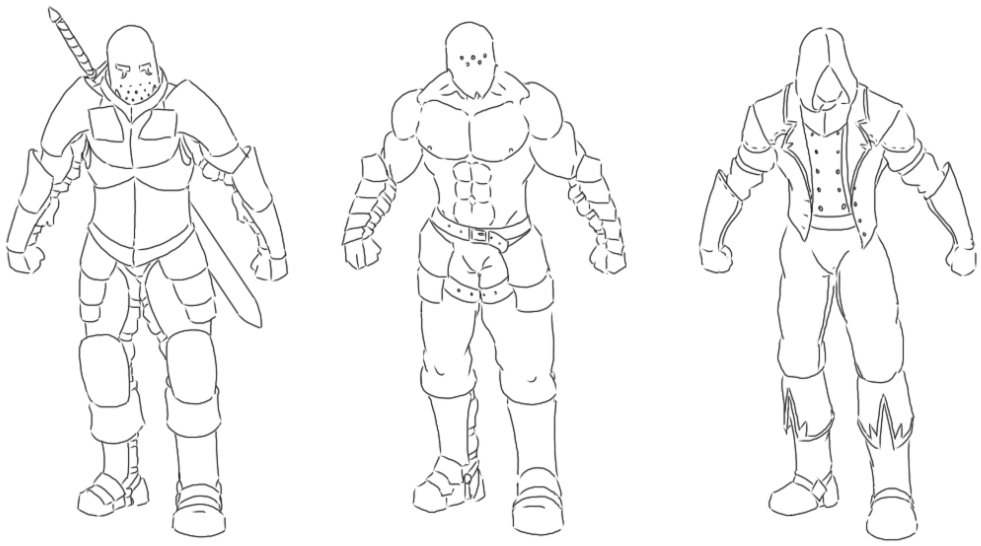

Figure 1: Early concepts for the three minion types which ended up becoming the Warrior, Brawler, and Wizard respectively.

The concept for the boss was based on the minions. He was to be the final challenge, and playing the game had to teach the player strategies to fight him: therefore, a fusion of the three enemy types would make the most sense. In his first stage, he fights like a warrior; in his second, once you have knocked his armor off, he acts as a brawler. Once his health drops below a threshold, he teleports away and behaves like a wizard. In each form, the boss is more powerful than the minion equivalent, thus presenting players with a similar situation but still challenging them. Concept art for the boss can be seen below in Figure 2.

Figure 2: Early concept of the boss.

Our idea for the game's introduction was a non-interactive "cut scene" movie. The player sees the world of Luxan at peace, when suddenly a portal is opened from which the villains of Tenebrax emerge. The cut scene was designed as a static image to save production time.

2.2 Story

Two worlds collide in violent struggle as the Highlord of Tenebrax leads his army of warriors, mages and brawlers into the peaceful realm of Luxan in search of battle and conquest. Most of the world's inhabitants prove easy prey for the trained killers, but one man vows to put a stop to the violence and save his world from enslavement.

Characters

- **Vir:** A citizen of Luxan, Vir is the main character of the game. He enjoys the peacefulness of Luxan, and hopes one day to become a motor bike sportsman. When his home is invaded, he will fight at all costs to save the inhabitants of Luxan and achieve his dream.

  The art style and gameplay of Vir are inspired by several characters from other games. His simple body shape and helmet were drawn from *Excite Bike*. The glowing visor on his head is reminiscent of Samus from the *Metroid* series. The enemy placement and weapon choices are similar to those found in the *Mega Man* series.
- **The Highlord:** Ruler of Tenebrax, a dystopian world of fire, death and endless war, the Highlord led his armies to conquest over his entire realm. With nothing left to conquer and bored of isolation, he turned to his cadre of mages and ordered them to tear open a rift between worlds so that he could spread his empire beyond the reaches of his own dimension. The Highlord is shown below in Figure 3.

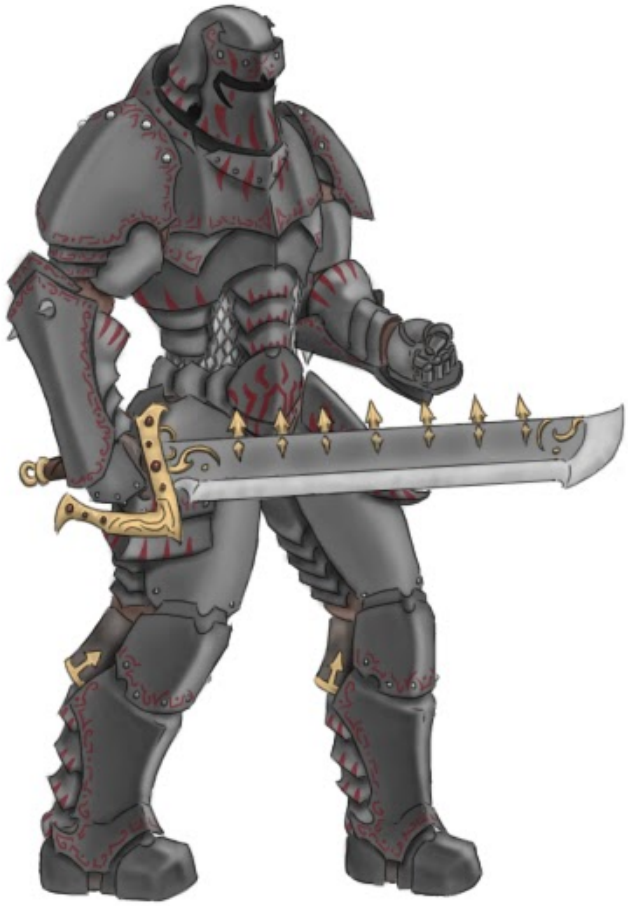

Figure 3: The Highlord.

A mighty warrior, The Highlord is skilled in all forms of combat employed by his troops, displaying peerless skill with a sword, destructive magic and his bare fists. Clad from head to toe in thrice-forged armor of dark iron, inscribed with blood-red runes of power, he carries a huge sword into battle but is not afraid to get his hands dirty if the situation calls for it.

The idea for our story was inspired by the conflict between the art styles of our two artists. The best way of using both styles in the same project, we decided, would be to have two different worlds.

Tyree's more cartoonish, abstract art style seemed more appropriate for the peaceful land of Luxan, as cartoonish drawings often appear friendlier, whereas Peter's more realistic style and specialization in fierce, threatening characters made him more suited to create the villains. Peter took charge of the Highlord and Tenebrax, while Tyree focused on Vir and Luxan. The two developed separately, but ideas were swapped and suggestions made regarding the world and story of the other.

2.3 Gameplay

The gameplay of our project is comprised of two mechanics, platforming and combat. The player has to navigate the hero Vir from platform to platform, making their way towards the end of the level in the top right hand corner of the map. The combat system lets the player attack in order to remove enemies, which appear as obstacles on the player's path to the goal. There is no scoring system, only a counter that keeps track of defeated enemies and rewards the player with upgrades, as described in Section 2.5.

The game contains three different types of enemies, and a boss enemy. The first enemy created for the game was the Warrior, a fairly simple enemy that guards platforms that the player needs to traverse in order to get to the boss. Warriors patrol on the platform they are standing on by walking from edge to edge. When a player approaches, they attack until they either die, or run away. The Warrior is able to take three attacks from the player, which is significantly more than other enemies, as a result of all of the Warrior's heavy armor, slow mobility, and simple AI. To compensate for the

ease that the player can avoid Warriors, each attack from a Warrior deals fifteen points of damage to the player.

The second enemy in the game is the Wizard, which has behavior similar to a “turret”. He floats in the air until a player approaches within a predetermined radius. He then begins to charge up a fireball and maintains that charge. If the player approaches even closer, the Wizard releases the charged-up fireball and immediately begins to charge another one to bombard the player with projectiles. Each fireball deals ten damage, since they are harder to avoid than a warrior's attacks but easier than a brawler's. The lightly-armored Wizard can be defeated in one attack from the enemy.

The Brawler is the third type of enemy. When the Brawler spots the player, he relentlessly pursues them using his high movement speed. Though armored even more lightly than the Wizard, his powerful physique lets him withstand two attacks from a player.

For our game, the health system for enemies consists of a number that represents the number of “hits” they can take. Each player attack counts as one hit. The player is able to regenerate health depending on the upgrades chosen. The total health regenerated per second is increased with each “good” upgrade, and the health gain system which initially starts at zero increases with each corresponding “bad” upgrade.

2.4 Level Design

We created two main levels; one for Luxan, where the player starts, and one for Tenebrax, which serves as the boss level, the arena where the player encounters the Highlord.

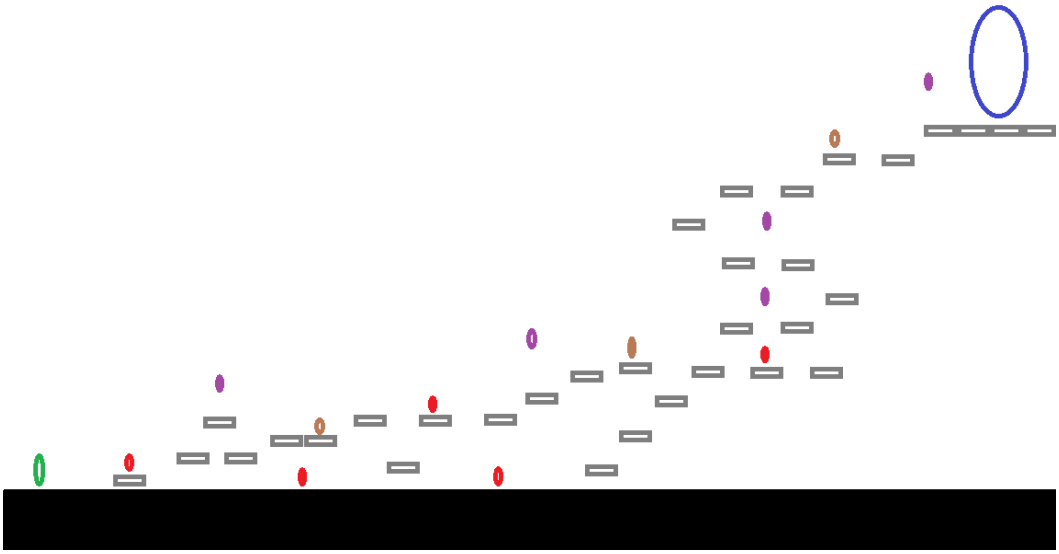

Figure 4: A rough draft of the main level.

We created a rough first draft in image format, color-coded to show the locations of enemies, as shown in Figure 4 above. This image was then sent off to Tyree, who implemented it and recreated the image in the game. The circles below correspond as follows:

- Large blue circle - This circle at the end of the level represents the end goal that the player must reach to progress to the final boss.
- Large green circle - This circle represents where the player starts.
- Red circles - These represent Warriors that patrol the platforms.
- Brown circles - These represent the placement of Brawlers.
- Purple circles - These represent the Wizards.

2.5 Upgrades

Vir is able to purchase an upgrade once he reaches a new tier, which is unlocked upon killing any five enemies. Each tier contains two upgrades, with a total of three tiers. The player is only allowed to choose one upgrade per tier, and each upgrade aligns either with Vir's home world, or with the world of the invading armies of Tenebrax.

The three upgrades that correspond to the "good" or Luxan side are themed around improving Vir's existing capabilities:

- Stronger Punches - Increase the amount of damage.
- Jump Height Increase - This is achieved through a double-jump.
- Increased Health Regeneration - While all upgrades provide some health regeneration for the player, this upgrade increases it significantly.

The three upgrades that correspond to the "bad" or Tenebrax side allow Vir to adapt and utilize the tools of the invading army:

- Sword - The sword extends the range of the melee attack.
- Chain - The chain allows Vir to grab distant enemies and pull them closer, with an example being the Wizard who typically floats out of range.
- Fireball – Shoot a fireball.

3 Art

Peter used Autodesk Sketchbook Pro, which offers features such as layer management and blending brushes. Sketchbook Pro can export images in many formats, including the PNG file format used by the game engine. This format was chosen for its high-quality lossless compression and support for alpha transparency. An example of Autodesk Sketchbook Pro is shown below in Figure 5, showcasing the several different layers that hold each “limb”.

3.1 Models/Figures

Figure 5: Autodesk Sketchbook Pro with the Warrior exploded to show parts.

Adobe Photoshop was used by Tyree to create the hero art. It has features similar to Autodesk Sketchbook Pro, including PNG export. An image of Vir being created in Adobe Photoshop is shown below in Figure 6.

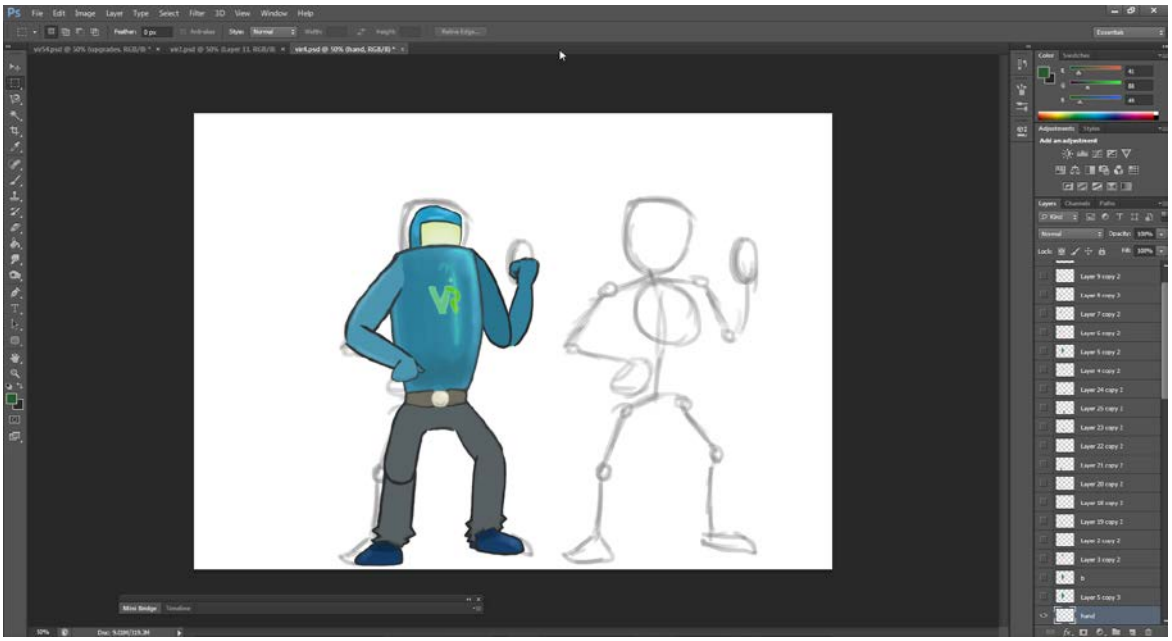

Figure 6: Adobe Photoshop with Vir in sketched pose.

Images were initially drawn with a Wacom Intuos Pro tablet. This device supports pressure sensitivity, which gave us the ability to control the width of brush strokes based on the pressure applied while using the tablet pen. After a basic sketch, we added bold lines to define the figure, then colored the art with flat colors, followed by shadows, highlights, and other details. We then combined the line-art and color layers, exported the resulting image and placed it in the game to see how it looked, repeating the process for each art asset.

3.2 Animation

We did some research on which program we should use to create animations for the characters in the game. We wanted a program that would let us individually manipulate body parts, similar to 3D art programs such as 3D Studio Max.

We settled on a program called Spine from Esoteric Software. It allowed us to animate characters by creating a “skeleton” and then attaching body pieces created in Sketchbook and Photoshop to individual “bones.”

When animating, we assigned key frames to the bones to specify the rotation, translation and scale of each body part over time. We also assigned key frames to certain bones to which multiple images where attached, allowing us to switch between different shapes. Example of this process can be seen in Figures 7 and 8.

Initially, we were going to export the animations using JSON, an efficient format that contains all of the parameters needed for a game engine to play an animation.

Unfortunately, we soon discovered that Unreal Engine 4 does not accept JSON files unless they are provided in a very specific format, and those produced by Spine are not compatible. To overcome this setback, we exported our animations as a sequence of PNGs, which Unreal is able to interpret as a traditional sprite sheet. This brute-force approach took away a lot of flexibility, since each frame had to be a static image instead of being composed in engine as a sum of the individual bones.

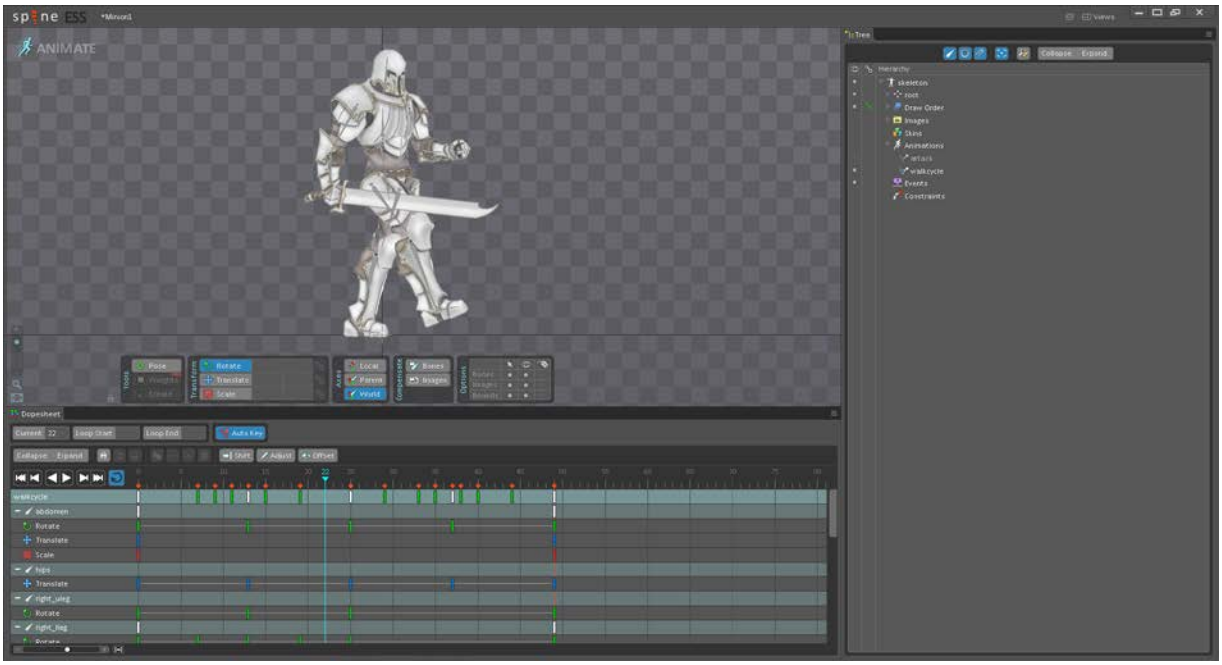

Figure 7: Animating the old Warrior in Spine

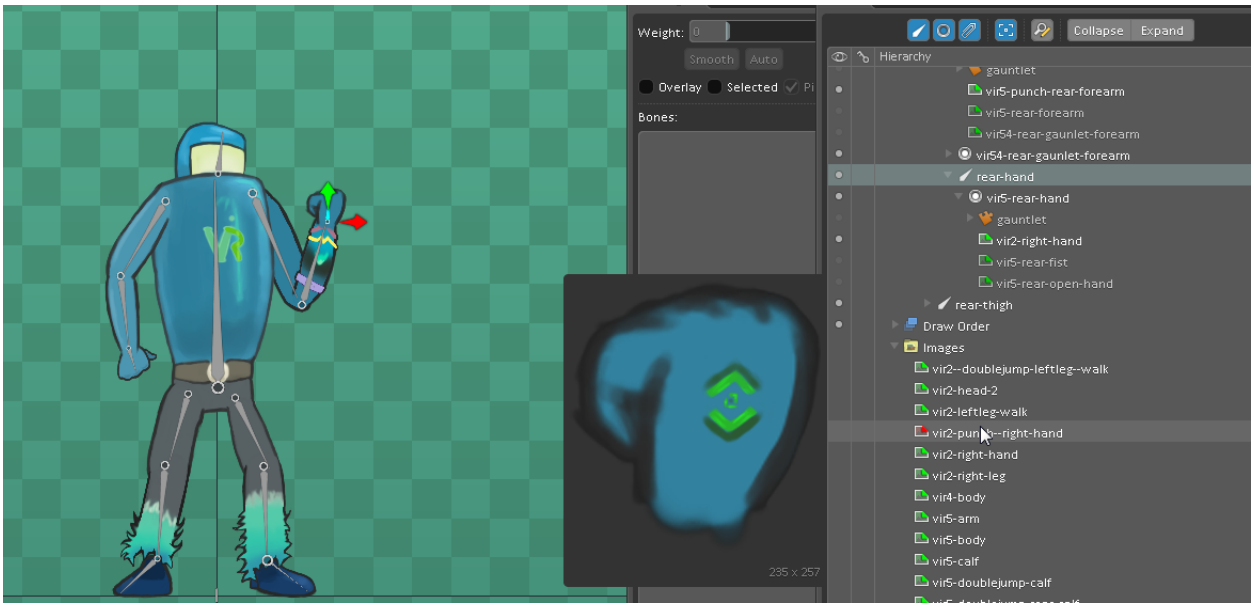

Figure 8: Adding image to hand bone of Vir.

Most of the animations that were created consisted of walks and jumps. The walk animations were looped so that the key frames of the character's pose at the beginning and the end of an animation would be the same. All animations were rendered to be

played at 60 frames per second to insure a smooth appearance in the game. The other animations created were the attack animations for each upgrade implemented in the game such as the stronger punches upgrade as shown in figure 8.

Most of the animations were at least 30 frames long. Posing each one sometimes required many hours over multiple work sessions. Refinements were made based on feedback from other team members and our advisors. In total, there are more than 144 animations spread across both artists in the project.

3.3 Music

The music for the game was created using Ableton Live 8.2. It is one of the more convenient programs of music production, as it provides access to a wide range of instruments which can be modified to create unique-sounding tracks. Music made with Ableton was exported into Unreal Engine 4 using WAV format.

A screenshot of a song being created in Ableton can be seen in Figure 9. In each row, there is a MIDI track with key presses for the instrument played.

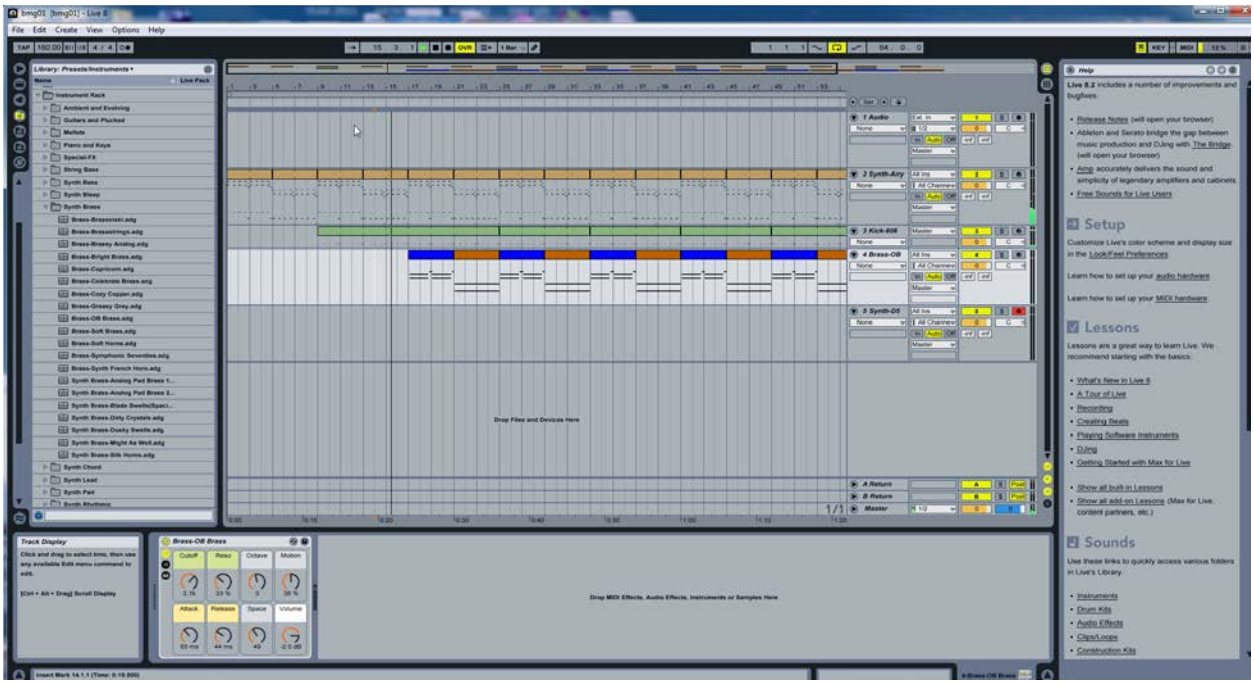

Figure 9: Ableton 8.2 with music sheets on multiple tracks, showing the process we used to create the music for the game.

3.4 Sound Effects

The sound effects for our project were recorded using a Blue Yeti microphone provided by the ATC. Reaper, an audio manipulation program, was used to edit, overlap and tweak the eight sounds we captured, adding effects such as equalization, pitch change and reverb as needed.

4 Technical Implementation

Our game is implemented in Unreal Engine 4. For programming in Unreal Engine, there are two primary options: create the code in C++ or in the provided visual scripting solution Blueprints. Blueprints is based on the Kismet visual scripting present in the previous iteration of Unreal Engine 3, with significant changes and improvements.

Compared to Kismet, Blueprints are now able to be separated from a level, meaning they are able to be re-used in various levels throughout a game and in the same level easily. A “Blueprint” is also able to inherit from a C++ class in the Unreal Engine 4 framework, allowing extension and the ability to create game logic.

The main benefit of Blueprints is the visual programming style, which allows more accessibility so that non-engineers are able to make changes. There are some drawbacks to using Blueprints, as complex procedures can become difficult to understand. Unreal Engine 4’s framework also is not entirely exposed to Blueprints, but the entire source code is exposed for manipulation in C++. For these reasons, C++ would be a better option to create that type of code and logic.

Blueprints was chosen as the primary method of implementation due to the context search available when creating the scripts. An example of the context search can be seen in Figure 10. Using Blueprints also gave us an additional benefit of shortening the time required for code changes and testing, as while there was “hot reloading” with C++ the time needed to recompile the C++ was still significantly longer. “Hot reloading” without a full restart of the Unreal Engine 4 Editor was unavailable until version 4.5, which was released after we started. Blueprints were also chosen for greater compatibility with future versions of Unreal Engine 4.

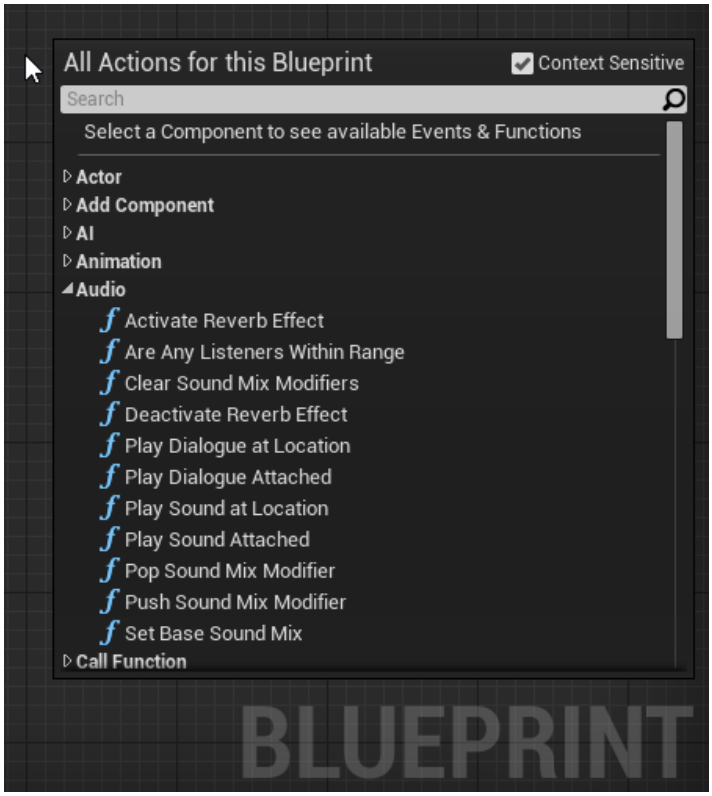

Figure 10: A list of functions provided by right-clicking in the Blueprints editor.

Unreal Engine 4's 2D support was introduced in version 4.3 in the form of Paper2D. Paper2D is the name given to the set of modules provided by Epic Games, the engine developer, in order to assist with creating games using 2D sprites instead of models. We utilized this subset of the engine extensively in order to render and manage the art assets in our game.

4.1 Animation

After an artist was finished with an animation and created the set of image files associated with that animation, the files were passed to our programmer, who imported the images into Unreal Engine 4 as "Texture" assets. After the import finished, the Texture assets could then be used to create "Sprite" assets, which could then be

combined into a “Sprite Flipbook” asset, which is Unreal Engine 4’s name for an animation sequence composed of 2D sprites.

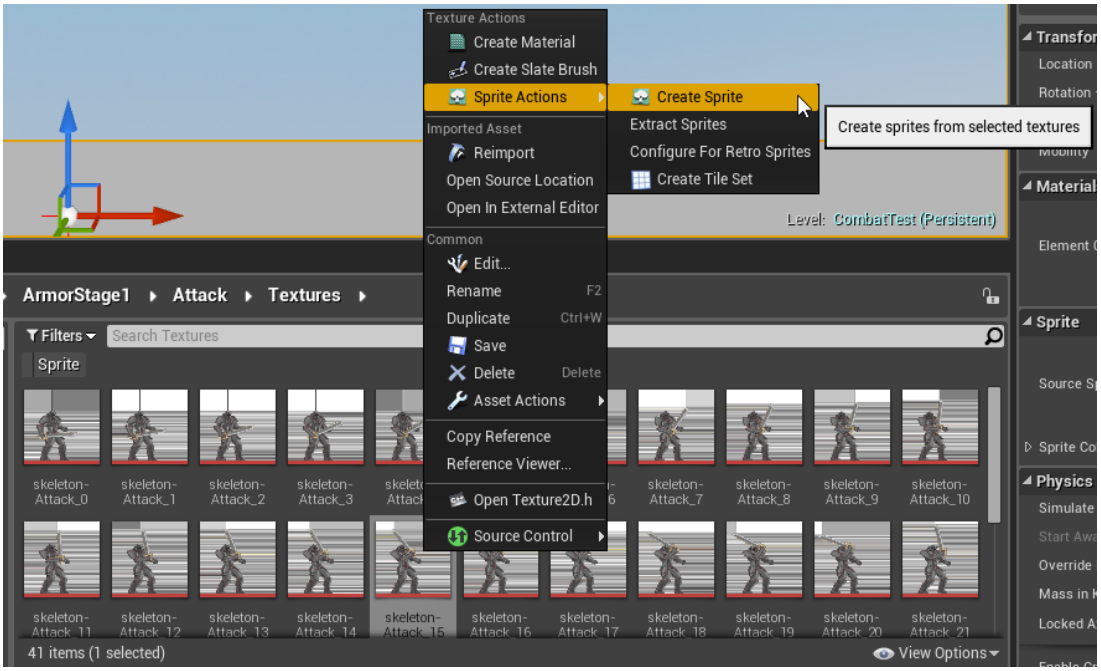

Figure 11: Creating the Sprites from the Texture files in Unreal Engine 4

Although Unreal Engine 4 provides a special type of “Blueprint” called an “Animation Blueprint” that acts as a state machine for all the different animations available for a skeleton, the Paper2D system did not have a system for managing transitions between different Flipbooks, so in the process of creating the main characters for the game, a system had to be created in order to manage the different types of animations each character would have.

The system that we used to manage the transitions between animations was created in Blueprints and is composed of several pieces:

- A list of animations which are supposed to be constantly looped, such as the idle animation for the hero, and are able to be “interrupted” and do not have to play out completely
- A data gathering function on the character’s current state, which sets corresponding variables
- A function that actually switches the animations, based on data provided in the previous two steps

In hindsight, if the system had to be recreated, we would implement it using C++ as the system became excessively complicated and unruly to manage in Blueprints.

4.2 Hero

Objects in the world capable of action are called “Actors”. A child class of “Actor” is the “Character” class, which is an actor that a player or an AI is able to control and has basic movement capabilities. Vir, the main hero character, is implemented in Blueprints and is sub-classed from the “PaperCharacter” class, which is a version of “Character” specifically made for 2D games in Unreal Engine 4. Vir is implemented through two Blueprints. One “Blueprint” acts as a parent “class” and contains all the code and abilities, with a second “child Blueprint” that contains all the configuration information. This implementation provided a system in which the main player character could be swapped between different Blueprints in the middle of game, with each “Blueprint” having different sprites for all the different animations. This design decision was made due to our initial plan to use a different set of sprites after choosing an upgrade, such that the visual appearance of the main character changed with the player’s decision. We ended up going with a different system, with the functionality to

switch between a different set of animations in one “Blueprint”, due to how the Upgrade system was actually implemented.

Movement for Vir is achieved using Unreal 4 Engine’s provided “Character Movement” component. This component handles most of the movement related logic. The portion of the logic that was required consisted of registering the buttons that would act as input, such as W for “Jump” and D for “Move Forward”. After the input is registered, an “Input Event” becomes available, and Vir was configured to process the Input Events by applying the desired movement direction and the force to the movement component using the “AddMovementInput” function.

Attacking for Vir is achieved by using two different collision components. One component represents the area covered by his punching animation, and was adjusted manually to cover that area. The other component covers the area of the sword attack animation. After the attack input from the player is processed, the corresponding animation is selected and played, and the corresponding collision component for the attack checks for any enemy Actors that are currently overlapping. The enemies are then sent a “Damage Event” that tells them which Actor is sending the event, and a float value that tells how much damage is being dealt. In order to prevent too many damage events from being sent in rapid succession, there is also a Boolean variable that is true after a damage event has been sent and while the attacking animation is still playing, so that one damage event is sent per punch.

While the Unreal 4 Engine provides a “Damage Event,” it does not provide a health system. The player’s health system consists of their current health, which is a float value that ranges from 0 to 100, a health “regeneration” system that adds to their

current health each second, and a health gain system that adds to their current health each time the player successfully deals damage to an enemy.

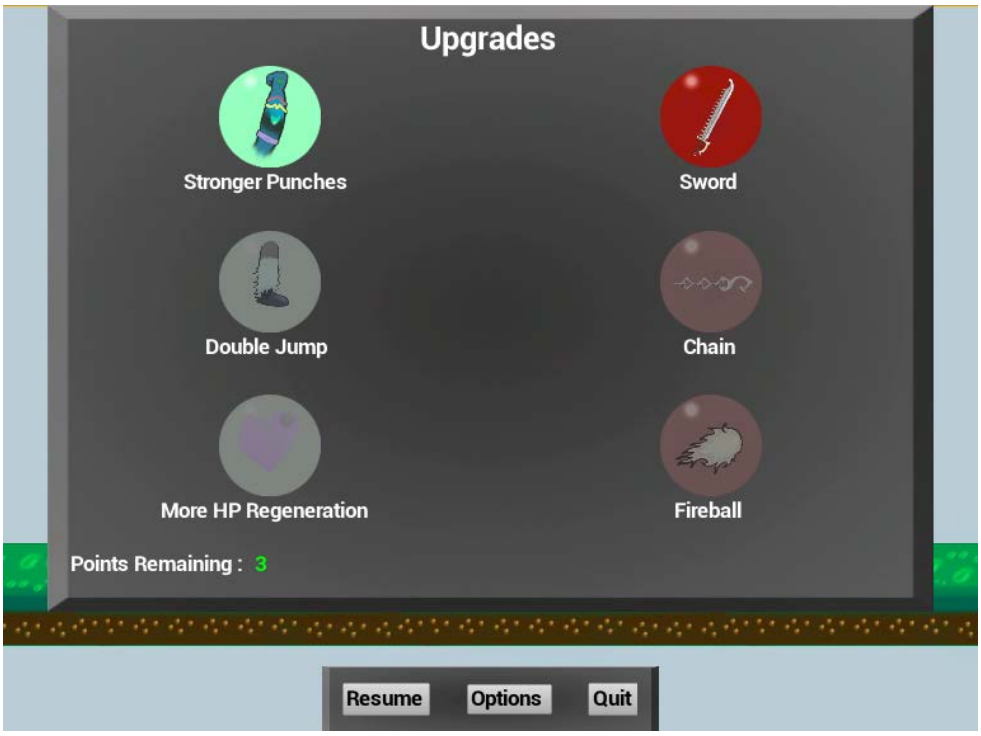

Figure 12: A screenshot of our upgrade screen created using UMG

A screenshot of the final upgrade screen can be seen in Figure 12 above. The upgrades on the left side correspond the Vir's home world, and the right side correspond to the realm of Tenebrax. The implementation of the upgrade screen is in Unreal Motion Graphics (UMG). UMG is one of the two UI modules provided with Unreal Engine 4, the other being Slate.

Slate is a UI programming framework that is primarily used for the Unreal Editor, and is the main framework for all of the Unreal Engine 4. To interface with and use Slate, a programmer needs to extend the core widget classes using C++. After creating a custom widget class, the programmer creates an instance of the new widget and adds it to the viewport of the player's screen.

UMG is a visual UI design tool intended to help create in-game UIs. UMG is based on Slate and extends it to create a library of basic widgets, such as checkboxes, sliders, and buttons. The main interface for creating a UI widget in UMG is a graphical interface in the Unreal Engine 4 editor, called the "Designer" interface. A UMG widget also allows for the creation of Blueprints code, so that widgets can read data from various variables, and "bind" or tie events to functions. For example, a Health Bar can read the value of the player character and scale accordingly, and an Upgrade Button can be tied to a function that alerts the player character that an upgrade has been selected. A screenshot of the early stages of our Upgrade menu can be seen below in Figure 13.

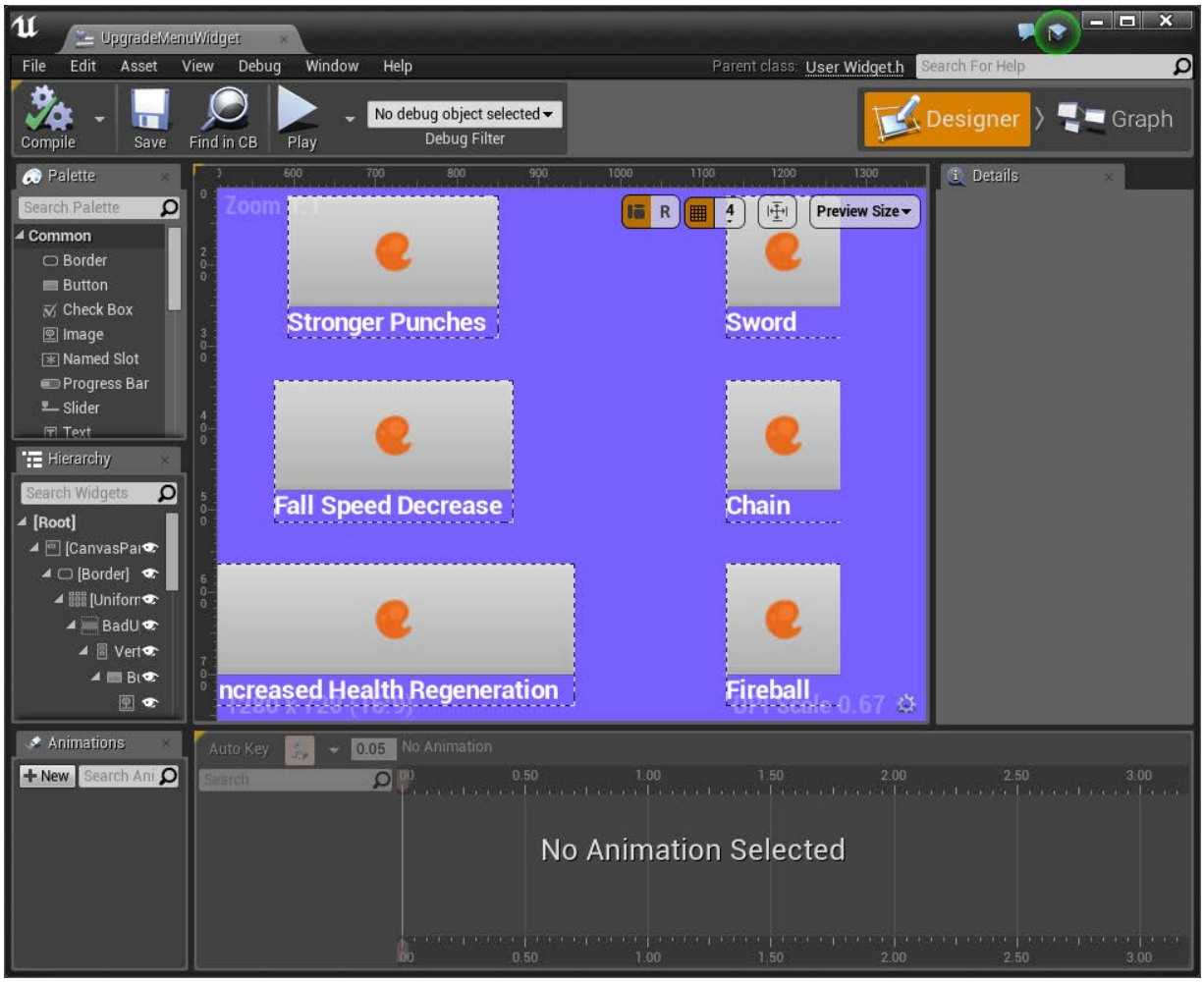

Figure 13: “Designer” interface for Unreal Motion Graphics,

showing our Upgrade Screen.

Our implementation of the upgrade screen utilized these features to set Booleans in the main hero object. These Booleans were used to determine whether the code for upgrades should be used, using standard if statements. The actual upgrades that the player can chose from on the “Luxan” side, or the left side, are as follows:

- Stronger Punches - This is achieved by using a different floating point value for damage if the Boolean for this upgrade is set.
- Jump Height Increase - This is done using a simple counter and if statement that checks to see if the jump button has been pressed twice, and if it hasn't it calls the "Jump" function.
- Increased Health Regeneration - Health generation is accomplished by using the Unreal Engine 4's OnTick event, and scaling the amount in accordance with the amount of time occurred since the last tick.

The upgrade options that correspond to the "Tenebrax" side are as follows:

- Sword - The sword is implemented using a secondary collision area, similarly to how the punch is implemented. The hero character reads the Boolean and selects the appropriate area to use, along with the appropriate animation.
- Chain - The chain is the most complicated upgrade from a technical perspective. The chain is created on the Unreal Engine 4's "BeginPlay" event which triggers on the game start, but after character creation. In this event, the chain links are created and are set to be hidden. When the player presses the button to use the chain, the OnTick event runs a method called "UpdateChain" which sets the visibility of the chain links. This sets the visibility to visible, from the first link to the last one, and then back to hidden from the last link to the first.

- Fireball - The fireball upgrade creates a new “Fireball” actor that uses Unreal Engine 4’s built-in projectile motion component for objects to shoot forward until the fireball either collides with an object or reaches the end of its timed lifespan. The actor is created from a predetermined offset, and uses the hero’s rotation to determine which direction it needs to move in.

4.3 Enemies

The enemies are implemented using the same “PaperActor” that the main character Vir is sub-classed from. The enemies all share a base class “Basic Enemy”. The reason for this is that they all share the same damage processing code, and using a single base class for all enemies provides a common class that can be used to check if an Actor being encountered is an enemy.

While Unreal Engine 4 provides an event that is triggered upon an Actor hitting the edge of the ledge, there were unresolved issues with the event never being triggered. As a workaround, our own version of detecting when the Warrior was on the edge of ledge was created. This checks the current velocity of the Actor, and upon detecting it has reached zero and that the Actor has stopped moving, our “At Edge of Ledge Detection” event is triggered. This causes the force that is being applied to the Warrior to change and apply in the opposite direction, and updates the movement animation to face the correct direction.

The Warrior achieves the attacking in a similar manner to how the main player character Vir achieves his melee attacks, with a collision area in front of them that the Warrior checks for the presence of players, and if any are found, plays the attack

animation and sends a damage event to the player. Like the man character, Warriors also use a Boolean to prevent sending more than one damage event per attack.

The second enemy, the Wizard, is a stationary character that shoots fireballs, limited to one every five seconds. To shoot a fireball, the Wizard creates a new “Fireball” actor at a predetermined offset from his current location. The fireball is given a rotation that orients it towards the player, then proceeds to travel forward in that direction until it either reaches the end of its lifespan, or collides with an object. During a collision, it checks to see if the colliding object is a player, and if so deals damage, then destroys itself regardless of what object it hit.

The third enemy, the Brawler, uses edge detection in a way similar to the Warrior, but instead of turning around the Brawler proceeds to jump and continue chasing. Due to how our “Edge Detection” works, the Brawler also jumps when he hits the edge of the platform, and then jumps up on the top of the platform to continue his pursuit of the player. When the Brawler finally catches up to the player, he proceeds to attack the player using a quick punching assault, using a Boolean to prevent additional damage events before the animation finishes playing.

When the boss is loaded, he starts off by playing an “Evil Laugh” animation and sound. When the sound finishes playing, an event is triggered that begins the normal boss AI. The boss functions as all three different enemies, using an enumeration that determines his current state. The boss switches between the different behaviors after taking a certain amount of damage at each stage. For example, after the Boss has taken ten points of damage while behaving like a Warrior, he then updates the variable that represents his current state to become a Brawler. The basic implementation of the

boss is similar to all the previous enemies, which the main difference being a switch statement being at each “Event” such as the “Edge of Ledge” and “Tick” events, so that depending on the boss’s current state, the correct code is executed.

5 Evaluation

In order to evaluate our success, we needed to be able to verify that we were able to do the following:

- Create a demo that could be completed in 5 minutes.
- Have an easily understandable control scheme.
- Have a set of balanced upgrades.

We conducted several playtesting sessions with friends to determine if we were able to meet the above criteria.

When conducting the playtest sessions, we recorded the overall time a participant needed to reach the boss level. We had one person recording video of the play session and another designated person observe and write down notes on the actions the player took, what they were confused about when they played, and recording any feedback the play tester had at the end. A screenshot of one of our playtesting sessions can be seen below in Figure 14.

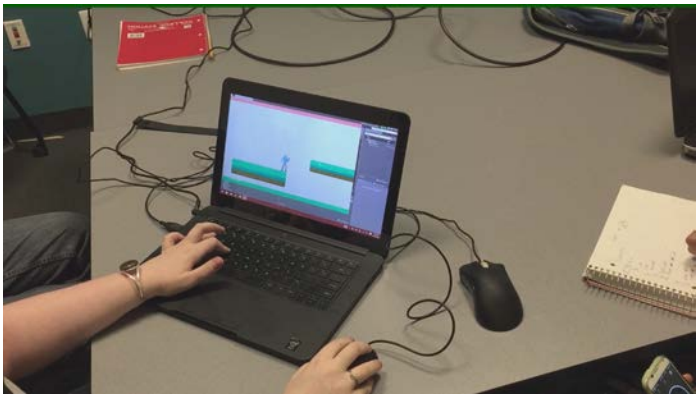

Figure 14: One of our playtesting sessions

From these playtesting sessions, the feedback we received was that the game was fun, although once a player had some of the upgrades, such as the double jump, they were able to skip past parts of the level and jump around enemies. The play testers were able to understand the control scheme, but gave us some feedback on how we were able to simplify it, such as using “F” for the fireball upgrade, instead of “Q”.

Due to time constraints we were only able to conduct sessions with four different play testers.

6 Technical Evaluation of a Scene in Unreal Engine 4

For the completion of the Computer Science requirement of our Major Qualifying Project, Robert completed a technical evaluation of the Unreal Engine 4 using the existing “Elemental Demo” available through the “Learn” tab in the Epic Games Launcher. A series of trials was performed using different play methods in order to determine what the impact of the play method would be on performance. The Elemental Demo was made available with the release of version 4.1 of the Unreal 4 Engine, close to when it was originally announced, as a demo that showcased all of the new features of Unreal Engine 4, with the camera moving through a pre-scripted sequence. The Elemental demo used was slightly modified to only play through the scripted sequence starting from 100 seconds in until 150 seconds, since from initial testing that segment provided the greatest change in framerate.

There are five different play methods that were explored, with three available options:

- Playing the current level in the Unreal 4 Editor Viewport
- Launching the current level in a separate Unreal 4 Editor Window,
- Launching the current level in a Standalone Game, which is a separate executable process.

The other play options that were explored were the two packaging options:

- Packaging with a “Development” configuration
- Packaging with a “Shipping” configuration.

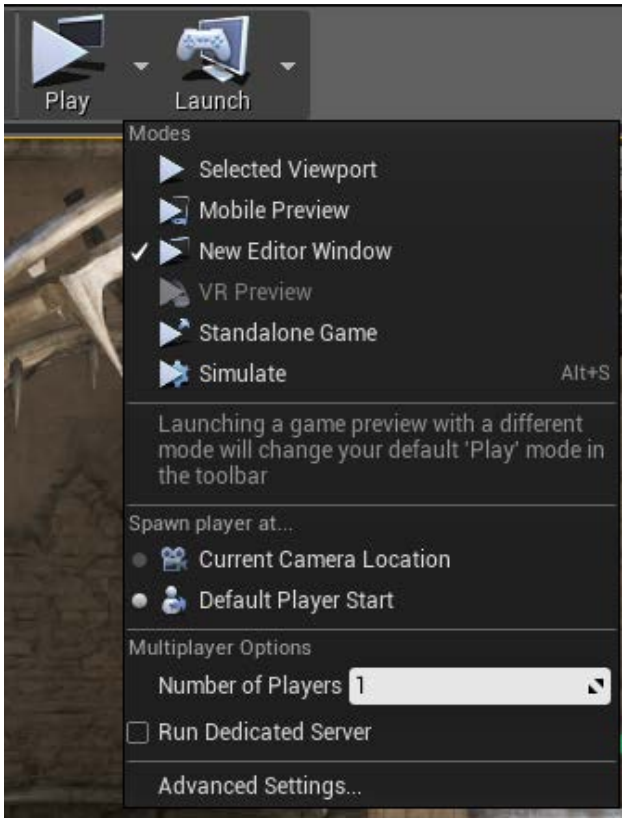

Figure 15: The in-editor play menu with the play options displayed.

The other play options that were available but not considered are shown in Figure 15, and are “Mobile Preview”, and “Simulate”. The reason for not using “Mobile Preview” was because the Elemental Demo was originally designed for non-mobile devices so using that method for asserting PC performance did not seem appropriate. “Simulate” runs the demo and allows the user to freely navigate the level, as such it would not actually run through the scripted camera sequence of the demo.

6.1 Platform

The hardware configuration used to conduct these trials was as follows:

- Windows 8.1 Pro 64-bit
- 8 GB DDR3-1600 MHz RAM Corsair XMS3 CMX8GX3M2A 1600C9
- Intel Core i7-2600 - 3.4 GHz
- EVGA GeForce GTX 780 Ti SC - 3GB DDR5
- Seagate 3 TB Hard Drive - 7200 RPM

The software used were as follows:

- Unreal Engine 4.7.5
- Fraps 3.5.99

6.2 Procedure

The follow procedures were used to configure the software before conducting the trials:

Fraps Setup:

1. Navigate to www.fraps.com and download and install Fraps.
2. Once Fraps has been installed, launch the program, and select the “FPS” tab at the top.
3. Ensure that the directory for the benchmarks to save in is valid.
4. Ensure that the “Benchmarking Hotkey” listed is valid.
5. Ensure that all three boxes under “Benchmark Settings” are checked and that the setup matches the configuration shown in Figure 17.
6. Ensure that the “Stop benchmark after X seconds” is checked and set to 50 seconds.

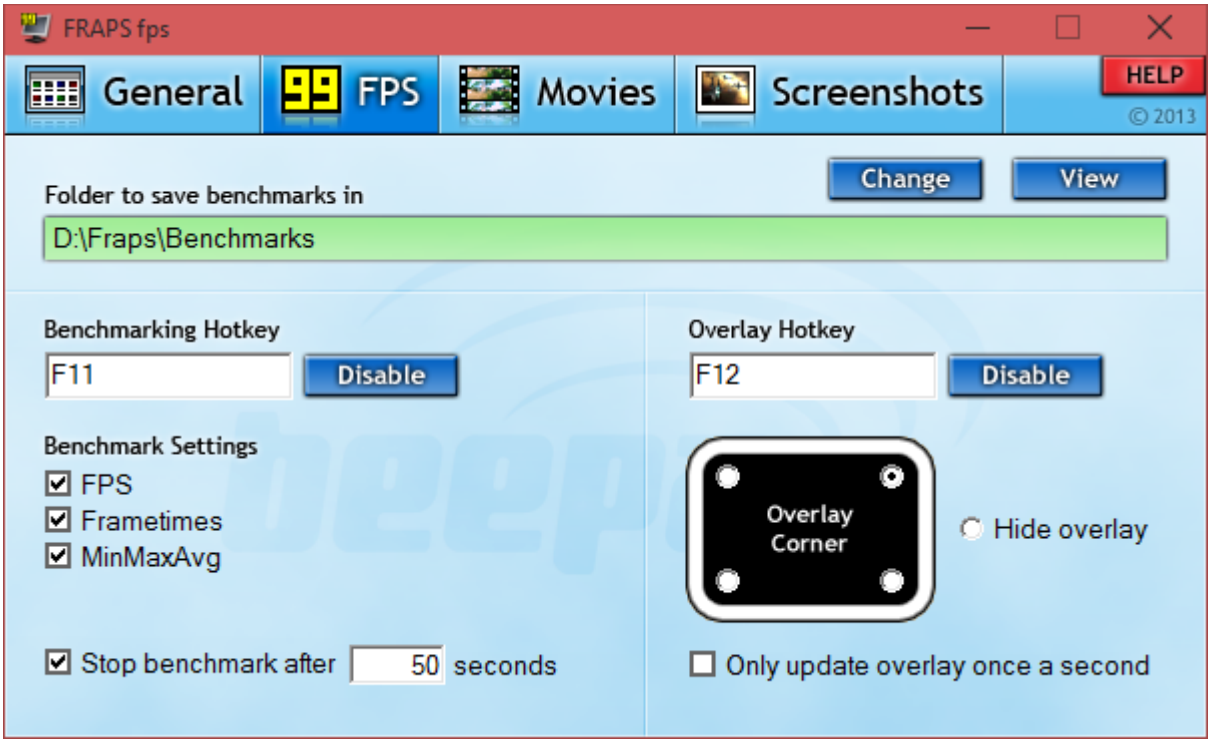

Figure 16: Screenshot of the settings used for Fraps

Packaged Methods Setup:

1. From the Epics Games Launcher, select the Unreal Engine 4 Project that was created from the Elemental Demo and open it.
2. From the Unreal Editor window, select File, Package Project, Build Configuration, and select the Development option.
3. From the Unreal Editor window, select File, Package Project, Windows, Windows (64-Bit) and select a file directory to which the packaged project will be saved.
4. Once that packaging has finished, repeat step 2 except selecting Shipping instead of Development.

5. From the Unreal Editor window, select File, Package Project, Windows, Windows (32-Bit) and select a different file directory to which the packaged project will be saved.

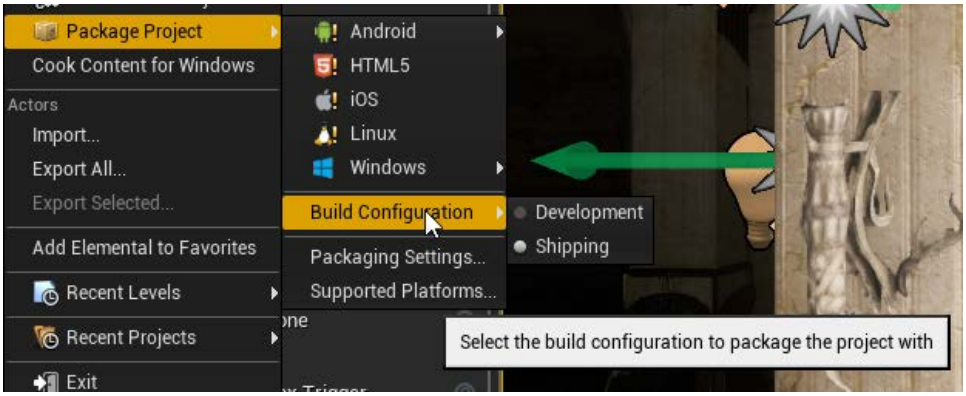

Figure 17: The package project options used in the course of the trials.

After downloading the Elemental Demo project from the “Learn” tab and creating the project, the following procedure was used to conduct the trials for the three in-editor play methods:

1. Perform the earlier setup methods mentioned.
2. Launch a play method using the Play button.
3. Let the play method finish running through the scene at least once to “warm up” the engine and to help eliminate any possible skewed data due to cached files.
4. Launch the level again using the same play method.
5. As soon as the level begins to play, press the previously set Benchmark hotkey for Fraps to begin recording the data.
6. After the benchmark finishes, open the previously set directory for Fraps and move all of the generated .csv files somewhere else with a label, such as “/Trials/Trial 1/Standalone Game/”.

7. Repeat steps four to six twice more for two more trials worth of data, and then repeat the entire procedure for each different Play method.

In order to get data for the packaged methods, the procedure for launching it is slightly different:

1. Perform the earlier setup methods mentioned.
2. Navigate to the directory in which the packaged demo was saved.
3. Open the “WindowNoEditor” folder and double-click the *.exe file inside to launch the game.
4. Let the game finish running through the scene at least once to “warm up” the engine and to help eliminate any possible skewed data due to cached files.
5. Launch the level again using the same method.
6. As soon as the level begins to play, press the previously set Benchmark hotkey for Fraps to begin recording the data.
7. After the benchmark finishes, open the previously set directory for Fraps and move all of the generated .csv files somewhere else with a label, such as “/Trials/Trial 1/Packaged - Development/”.
8. Repeat steps five to seven twice more to generate two more trials worth of data, and repeat this entire procedure again with the other packaged game.

For the last set of trials conducted, the following procedure was used to adjust the graphics options:

1. Package the Unreal 4 project using the “Development” configuration as in steps one to three above.
2. Navigate to %LOCALAPPDATA%/ElementalDemo/Saved/Config/WindowsNoEditor/GameUserSettings.ini” and edit the file to contain the following:

    [ScalabilityGroups]

    sg.ResolutionQuality=100

    sg.ViewDistanceQuality=3

    sg.AntiAliasingQuality=3

    sg.PostProcessQuality=3

    sg.ShadowQuality=3

    sg.TextureQuality=3

    sg.EffectsQuality=3

    This file must be edited in between each trial.

3. Gather results using the remaining steps, four through eight. After running through the first set of trials using the starting configuration, modify the configuration such that each different value for each option is tested, with the other options remaining at their original values. The graphics options and values that were explored were as follows:
    - Resolution Scale, “sg.ResolutionQuality” - 25, 50, 75, 100
    - View Distance, “sg.ViewDistanceQuality” - 0, 1, 2, 3
    - Anti-Aliasing, “sg.AntiAliasingQuality” - 0, 1, 2, 3
    - Post Processing, “sg.PostProcessQuality” - 0, 1, 2, 3
    - Shadows, “sg.ShadowQuality “ - 0, 1, 2, 3
    - Textures, “sg.TextureQuality“ - 0, 1, 2, 3
    - Effects, “sg.EffectsQuality“ - 0, 1, 2, 3

6.3 Results

One of the initial tests conducted was an attempt to narrow the scope of the demo to a particular scene that had many drops in frame rate when played using a “Standalone Game,” and to see what adjustments might be able to help prevent the drops in frame rate. Figure 18 below shows the current frame rate at each second along any given second.

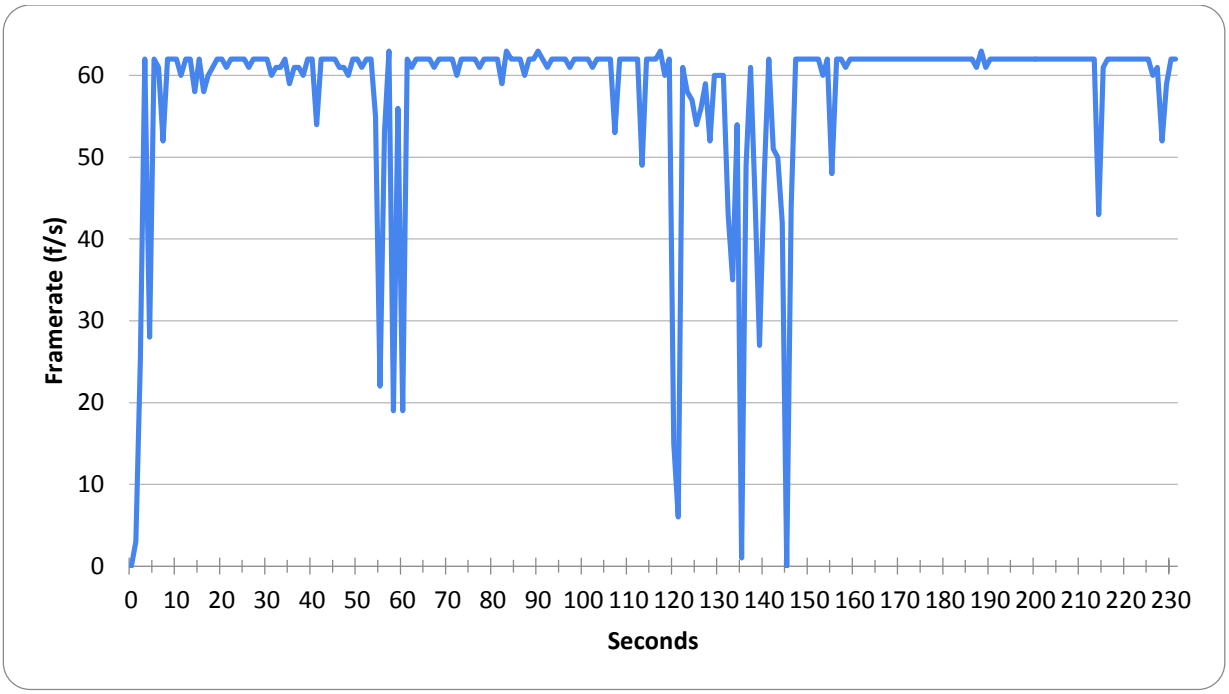

Figure 18. Framerate vs Second of the entire “Elemental Demo.”

From these initial trials, it was evident that the time period which had the most drops in framerate was between 100 seconds to 150 seconds, and so the scope of time that was examined in further trials was narrowed.

The next trials that were conducted looked at the different play options. After conducting all the trials, and analyzing all the data, the following charts were created to showcase the findings and conclusions that were drawn. Figure 20 below shows the average frame rate for each trial on the left axis, and compares it against the different play methods on the bottom axis.

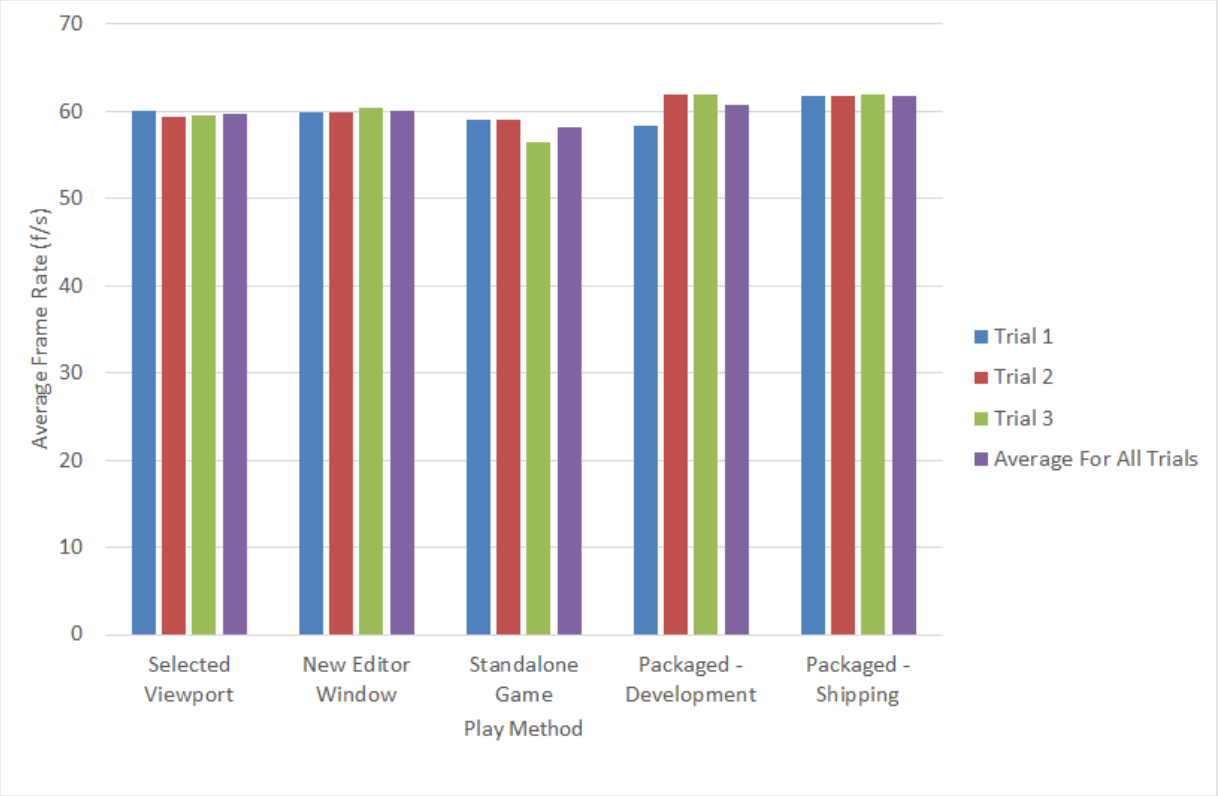

Figure 19: A bar chart showing the Average Frame Rate versus each Play Method and how they compare to the overall Average Frame Rate.

As shown in Figure 19, the average frame rate across all trials stayed roughly the same at around 60 frames per second, with the “Standalone Game” method showing the overall lowest frame rates, and “Packaged - Shipping” showing the highest frame rates.

After comparing the play methods available, the play method of “Packaged - Shipping” was selected as the method that would be used in conjunction with testing various combinations of the scalability options available to see if any of the options would impact the performance of the scene when adjusted. While other play methods were extremely close, this method is exactly what a player would experience on their own machine.

Figure 20 displays the average frame rate (f/s) of the three different trials, with each of the scalability options examined. Resolution Scale has a range of 25% to 100%, and the other options range from zero to three.

Scalability Options \ Value Used	0 (25%)	1 (50%)	2 (75%)	3 (100%)
Resolution Scale	61.99	62.00	62.00	62.00
View Distance	62.01	62.00	61.97	62.00
Anti-Aliasing	62.00	61.99	62.00	62.00
Post Processing	62.00	61.99	62.00	62.00
Shadows	62.00	61.98	62.01	62.00
Textures	61.99	61.98	61.99	62.00
Effects	62.01	62.00	61.99	62.00

Figure 20: Average frame rate across all trials. 3 is the highest value possible and 0 is the lowest

As can be seen from the table, there is only a minimal difference in frame rates between all the options, with variations ranging between -0.02 to +0.01 frames per second.

6.4 Summary

As can be seen from the results, using either the “Play In Selected Viewport” or “Play in New Editor” play methods proved to be the closest way to see how actual performance matches the “Shipping” configuration without packaging the entire game, which can be a time consuming process. Changing the graphics option also had little impact, which may be due to a variety of reasons, such as a frame limit that prevented the engine from rendering more than 60 frames a second. Another way to explore the impact of the scalability options would be to find a lower-end platform that these trials could be conducted on, since that type of platform is more likely to utilize the scalability options.

Potential errors may be due to the reset of the configuration file whenever the game was restarted, in which the settings were overwritten by Unreal Engine 4 before the trial was run. A potential fix for the configuration file reset is to configure the demo to allow a custom configuration of options before it executes, or seeing if the overriding of the configuration can be prevented. Another option to fix this error is to use one of the non-packaging methods and use the in-editor scalability options. In order to fix the frame rate capacity issue, there may exist a setting that allows it to be disabled.

7 Conclusion

We created a short 2D side-scrolling platformer game in Unreal Engine 4 using Photoshop, Autodesk Sketchbook Pro, and Spine for creating the art. Creating the art in both Photoshop and Sketchbook Pro went smoothly. Spine facilitated creating animations for the characters in the game after importing the body parts for the character and binding them to the skeleton we created for that character. We used Ableton Live to make some sound effects and background music in a short period of time.

The testers were able to understand and finish the game within our expected timeframe. Some had difficulty being able to tell when they were taking damage, while others found the control scheme confusing. Although we were not able to address all of the playtesting suggestions due to time constraints, we were able to implement some, such as changing the controls.

Our decision to build a 2D game in Unreal Engine was not optimal. Even though Paper2D has been out for a while in Unreal Engine 4, when we started it was brand new and is still marked as “Early Access Preview”. The system for managing 2D sprite animations is also significantly sub-par compared to the “Animation Blueprint” system available for 3D models.

Coding a game entirely in Blueprints also did not work out as expected. While Blueprints is an extremely useful tool and easy to use, it seems that the performance suffers as the Blueprints grow in size

Most of the above issues were consequences of doing a project in a relatively new engine. As an example, during the course of our project, the engine went through several major updates, and the documentation was frequently underdeveloped, missing, or out of date.

Despite these problems, we believe we ended up with a well-polished game that showcases our individual abilities.

8 Works Cited

"Portable Network Graphics." PNG () Home Site. http://www.libpng.org/pub/png/

"JSON Tutorial." JSON Tutorial. http://www.w3schools.com/json/

"Esoteric Software." Spine: Videos. http://esotericsoftware.com/spine-videos

Sieprawski, Brandon. "Unreal Engine 4.5 Released!" Unreal Engine 4.5 Released! https://www.unrealengine.com/blog/unreal-engine-45-released 14 Oct. 2014. Web.

"Slate UI Framework." Unreal Engine. https://docs.unrealengine.com/latest/INT/Programming/Slate/index.html

"UMG UI Designer User Guide." Unreal Engine. https://docs.unrealengine.com/latest/INT/Engine/UMG/UserGuide/index.html

World of Warcraft. Blizzard Entertainment. Nov. 23, 2004.

Guild Wars 2. ArenaNet. Aug. 28, 2012.

Excitebike. Nintendo. November 30, 1984

Super Metroid. Nintendo. March 19, 1994

Mega Man. Capcom. December 17, 1987

9 Appendix

Original Idea

The original idea was for a first-person, 3D exploration and survival game with a focus on story, set in the aftermath of a volcano eruption in California. It followed the last member of a resistance group fighting back against Gabriel, the self-made tyrant of what city was left semi-intact and the leader of one of two gangs that constantly fought over the area -- the Devil's Hounds, the gang with the most resources and best organization, led by Gabriel, who treats those living in his domain as a resource to be exploited; and the Crimson Storm, a gang of bloodthirsty marauders only barely held together by their brutish leader, Ozhog.

Figure 20: An early concept drawing of Gabriel.

As these two gangs fought over the city, and the private military contractors who had been sent in to keep order shoot first and ask questions later, the player must survive and either escape or die.

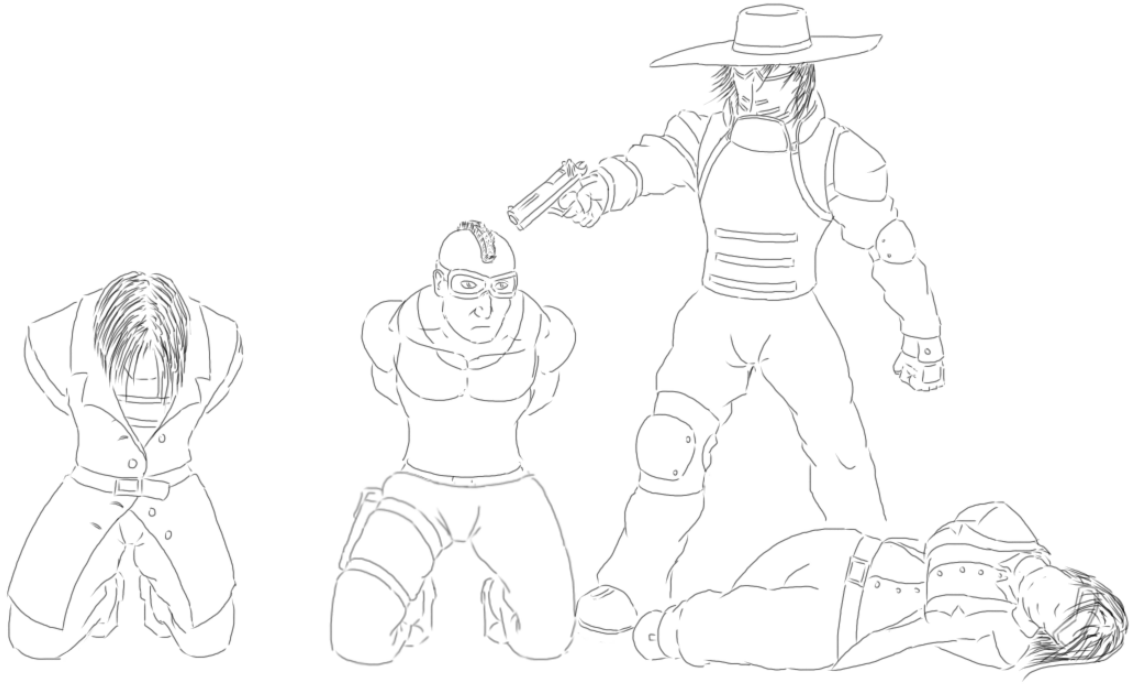

Figure 21: A concept for a cut scene from the original game idea.

The goal of this game was to take down the two gangs and the PMCs by killing the leaders of all three, after fighting through the rest of their gang, including mini-bosses with names and backstories of their own. Taking over bits of the city would confer bonuses, and the player might get some allies to fight beside the player along the way.

9.1 Design Process and Re-Scoping

Our original idea for the game was far too large to be practical. It involved combat, stealth, scavenging for supplies, character progression with three complete skill trees, and a large open world city with both gangs to fight, a series of bosses, full voice-acting, and multiple endings. Needless to say, this would be an ambitious idea for a full studio with a few years to work. For a team of four who had one school year, it was impossible.

So we re-scoped. The project became story-focused, and nearly all of the combat, stealth, and everything else was dropped. We focused on the cinematic experience, a story told through environment and cut scenes. The mechanic for this story would be “active” objects that would trigger flashback cut scenes, such as a knife, a radio, and an old newspaper.

The goal became simply to find the player character’s friend, a scientist who used to work for Gabriel, collect some spare parts and fuel, then repair a helicopter and fly away. When player got to the end, however, it would be revealed that it was Gabriel misleading the player all along.