Proj Code: DMO- 4167

DEMON DISSENSION

A Major Qualifying Project Report:

Submitted to the Faculty

of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Anthony Sessa

Nick Konstantino

Brian Seney

Michael Metzler

Professor Dean O'Donnell, Major Advisor

Professor Mark Claypool, Co-Advisor

Date: April 25th, 2013

Abstract

Demon Dissension provides a strategic twist on the traditional fighting game experience to players and showcases complex game logic, networking, and fighting game design principles. Built entirely in the Unity engine and programmed in C#, *Demon Dissension* pits two players against one another in a battle for glory, challenging them to not only fight against the character in game, but the strategies being employed by the actual opponent. A team of two artists and two programmers took four terms to create a deep multiplayer battle experience.

Table of Contents

List of Figures

Special Thanks

The Demon Dissension team would like to thank Professors Dean O'Donnell and Mark Claypool for all of their advice and guidance over the course of the school year. This project would not have achieved the high level of quality it did without them. We would also like to thank Breeze Grigas for the art that he worked on for the game

1. Introduction

Fighting games are games where 2 players battle to the death utilizing a roster of characters with different abilities. *Demon Dissension* is a fighting game with RPG elements, meaning that it introduced the concept of a player being able to customize their character's battle attributes, in order to make it a more personal experience. *Demon Dissension* features single player versus an adaptive AI, local multiplayer play, and network play that has both manual and matchmaking methods of connecting players. The players have four characters to choose from, each with a unique skill set, but all share similar traits such as special and super techniques, dashing, jumping, and eleven separate basic attacks. The goal of the game is to simply reduce the opponent's Hit-points (HP) to 0, or just have more HP when the timer runs out.

Two programmers and two artists worked on the project for four terms. Anthony Sessa was assigned the roles of producer, sound design, and stage design and modeling. He kept the schedule of all the meetings and deadlines, as well as recording sound effects and implementing them for the characters and stages. He was also responsible with making sure the stages fit aesthetically with the characters, and were not so filled with detail that it caused the game to slow down in any way. Nicholas Konstantino was the lead game designer and lead artist. Nick created all the assets for all four of the characters, from concept art to finished, rigged models with full sets of animations, as well as made all the final design and balance decisions. Brian Seney was the lead programmer, and coded much of the game logic, menu systems, and useful in-game extras we did not even plan for, such as the debug attack editor, the adaptive AI, and the achievement system. Michael Metzler was the main network programmer, who over the course of the project created a playable online mode with a central, hosted server that cut down on lag

time fairly effectively. For a term, Breeze Grigas of Becker College designed a character and did some texture work to help the characters and the world come together more solidly.

1.1 Concept Development

The original idea we had pitched was combining a "beat'em up" game similar to *Streets of Rage*, where players would fight their way to the center of the level, collecting stat boosts along the way, and then finish the level by fighting each other like in a normal fighting game. The problem with this idea was not that the scope was too large, but rather that the entire idea of having to play a mini-game in order to play the main game was flawed. Playing that mini-game would eventually become a chore for the players who just want to start fighting right away, and then giving them the option to skip it will end up making all the work to make that "pre-fight" game wasted. We cut that feature in favor of using a stat menu.

1.2 Conceptualization

The fighting game genre has been really successful since their reintroduction to the mainstream with the release of *Street Fighter IV*, and each series has its own unique systems for combos, techniques, or gameplay. We tried to find a feature that those games had not yet done, a feature that could take the limited character pool we were forced into due to time and artist constraints, and retain the strategic depth that fighting games are known for. That solution ended up taking a customizable stat system and applying it to various fighting game stats.

We created a story stating that the players themselves are demons who summon these fighters out of their respective worlds and pit them in battle against one another for entertainment.

1.3 Gameplay Description

Demon Dissension is a fighting game, and a fighting game is a genre that is centered on executing a player character's attacks and skills more effectively than the opponent in order to reduce their HP to 0, or lower than the player's own HP before the time runs out.

Fig 1.1 A sample game screenshot

The players have 3 'bars' they need to pay attention to on the user interface: their HP (yellow), their Energy Meter (green), and their experience (EXP) Meter(blue). HP is reduced by attacks, but the other two meters are slowly filled when landing or receiving hits. The red area of the HP meter is health that is retained by blocking, which slowly regains over time.

While in combat, each player has 1 super move, 2 special moves, and 11 normal attacks - 4 standing, 4 aerial, and 3 crouching. The super skill can only be used while the player's energy meter is full, and it is used by pressing the button combination A+B. Special moves are used by doing a combination of directional inputs and either A, B, or C buttons. The rest of the moves are done by pressing on of the A, B, C, or D buttons either while standing, while jumping, or

during a crouch. The weakest and lowest priority moves, the "A" moves, can be canceled into B moves, and B into C, and so on, in order to create combo strings. Combo strings are attacks done in quick succession one after the next which is guaranteed to work on the opponent if the player has consistent timing.

Each character has similar defensive options, as well. Most importantly is the ability to block, which is done by holding back when the player is being attacked. Doing so will reduce the damage the player take and leave some "red health" that will slowly regain over time, unless the player gets hit. If the player blocks within a small frame window of being struck with an attack, the player does a "perfect" block that allows the player to do a quick counter attack if they were ready for it. The player can also press an attack button when an attack collides with their block to do a "push block", moving their opponent back out of range, in order to get more space to reset to a neutral position. Other defensive and movement options include walking by holding left or right, dashing by double tapping left or right, jumping by pressing up, air dashing by double tapping left or right while airborne, and crouching by holding down. The more advanced hybrid offense-defense option is the "Omni-Cancel", which is performed by pressing A+B+C buttons. For a cost of some of the player's meter, it will take them out of any attack or animation their character is currently performing, making it useful for unorthodox combos or getting out of dangerous situations.

Finally, the main attraction to our game is the level up system. When the player's EXP meter fills up, it prompts the player to level up by pressing A+B+C+D. Doing so will imbue the character with the bonuses the player had selected before the battle began, at the menu shown in Fig 1.3. The player will instantly heal any red health they have, as well as UI and character

specific aesthetic changes to make it visually apparent to the player and their opponent what level the characters are, as shown in Fig 1.2.

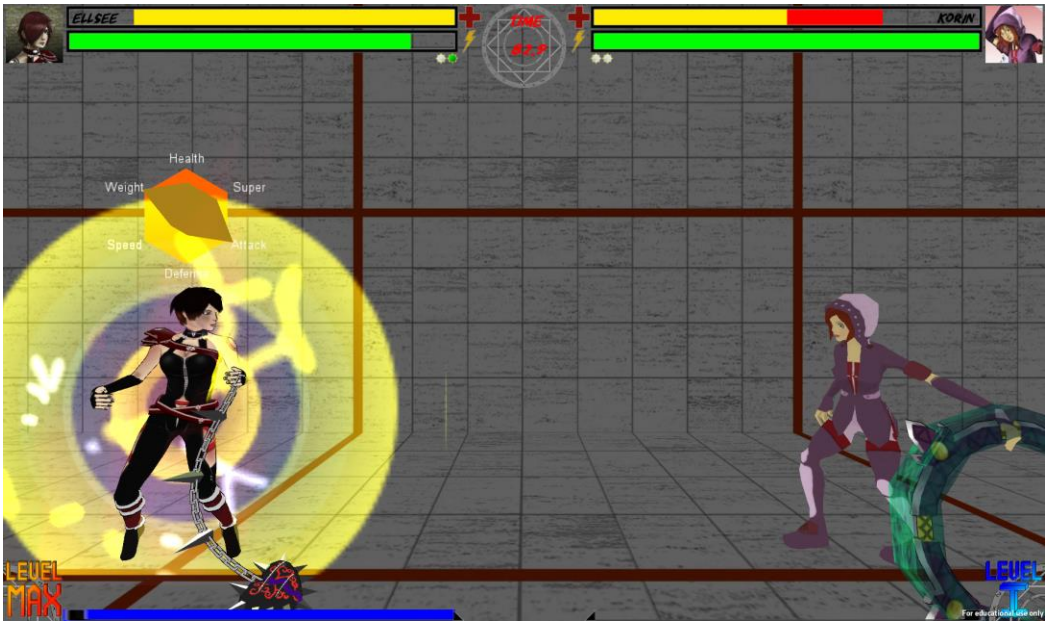

Fig 1.2 Screenshot showing the level-up mechanic in action

Fig 1.3 The stat select screen

The six stats the player has to choose from are:

- HP: Increasing this stat increases the character's overall health.
- Energy: Increasing this stat increases how quickly the character gains energy in battle.
- Attack: Increasing this stat increases how much damage the character's attacks deal.
- Defense: Increasing this stat increases the amount of health the character retains on block.
- Speed: Increasing this stat increases how quickly the character moves around the stage.
- Weight: Increasing this stat increases the amount of damage the character can take before dizzying.

1.4 Production

Our producer, Anthony, was in charge of keeping the group organized. He was responsible for scheduling all of the meetings as well as facilitating communication between the group and advisors. He was also responsible for ensuring the group was kept on task so milestones could be completed on time. Three major components went into ensuring a quality production job was achieved. These components included the scheduling of meetings, what actually happened at each of these meetings, as well as the production documentation.

Anthony scheduled an average of three meetings a week for the duration of the project. Once at the beginning at the week to discuss what had been done the last week, where that left us in our schedule, and what was projected to be done by the meeting with the advisors by the end of the week. The second meeting of the week was dubbed a "working meeting". Here, we all convened in the IMGD lab and spent an hour working on the project together. This facilitated

high productivity since we were all together and able to answer any questions that may have cropped up with ease. At the third and final meeting of the week, we met with our advisors Dean O'Donnell and Mark Claypool to present them with that progress and discuss our plans moving forward. These meetings were instrumental in determining whether or not we were developing a quality game.

All of our work was tracked on various Google Documents and weekly Tumblr blog posts were made to summarize our progress as a group. Some of these documents included a daily time sheet so we could track how many hours each of us was putting into the project, various design documents for characters or stages, as well as a to do list which had tasks we had to complete listed in order of difficulty and importance. Each of these was updated daily so Anthony could easily stay informed on the rest of the group activities and make any adjustments to milestones that may have been needed. Anthony also updated the Tumblr blog every Friday to provide an update to people outside of the project team on the progress of the game. These ranged in length from a paragraph or two to a much more substantial post if an entire term was being recapped.

2. Artistic Methodology

This section discusses the methods the artists went about to plan out, create, and eventually implement each of the art assets, so that one can observe the various thought processes, design decisions, and workflows experimented with, broken down by each major asset, and furthermore by character or stage when necessary.

2.1 Art Direction

The art favors simplistic models, colors, and textures, taking reference from such games as *Jet Set Radio Future* (Fig 2.1), *Dragon Ball Z: Budokai*, *The Legend Of Zelda: The Wind Waker*. These games all make use of a toon shader in order to make the assets appear more like a cartoon. The decision to utilize toon or cel shaders in *Demon Dissension* was not made until Unity3D's lighting made it difficult to use other shaders to properly light the assets in a similar style.

Fig 2.1 A sample of characters from Jet Set Radio Future, as they appear in game.

2.2 Character Design

2.2.1 Concept Art

Ellsee's character design Fig(2.2) was inspired by the idea of creating a strong, empowered female character who can stand up to any man in the fighting arena. The idea of her being a "heart breaker" was taken to the next level by giving her a weapon in the shape of a locket, for the sole purpose of breaking her opponents. Some of her style and demeanor were taken from strong female characters in gaming, like Tifa of *Final Fantasy 7* and Laura Croft of *Tomb Raider*.

Fig 2.2 Finalized Concept Art for Ellsee.

Velle's character design (Fig 2.3) was inspired by fighting game characters like Ky Kisuke of the *Guilty Gear* series and Jin Kisaragi of the *BlazBlue* series, and then given a futuristic vibe. His weapons were designed to make his playstyle drastically different than Ellsee's. Not many of the decisions made about Velle's visual design were made during the concept art stages, and mostly made during modeling phases.

Fig 2.3 Velle's 'Final' concept art.

Penguin's design (Fig 2.4) was simple. What started out as a joke character gained popularity, so the team decided that in order to make it more over the top, the best way to make a penguin into a fighting game character would be to give him samurai armor, Heihachi-inspired hair, and a sword. He is designed to be a battle-worn and fearsome penguin warrior named Penguin, and has since become a mascot for the game.

Fig 2.4 Penguin's Final Concept Art

Korin's design (Fig 2.5) was inspired by the desire to create a second female character that would stand as an opposite to Ellsee. The goal was to make a character who was colorful, friendly, and lively. Her weapon and move set were inspired by the rhythmic gymnastics during the summer Olympics. Due to time constraints, Korin's concept artwork was tasked out to a student (Breeze Grigas) doing work for independent study project credit.

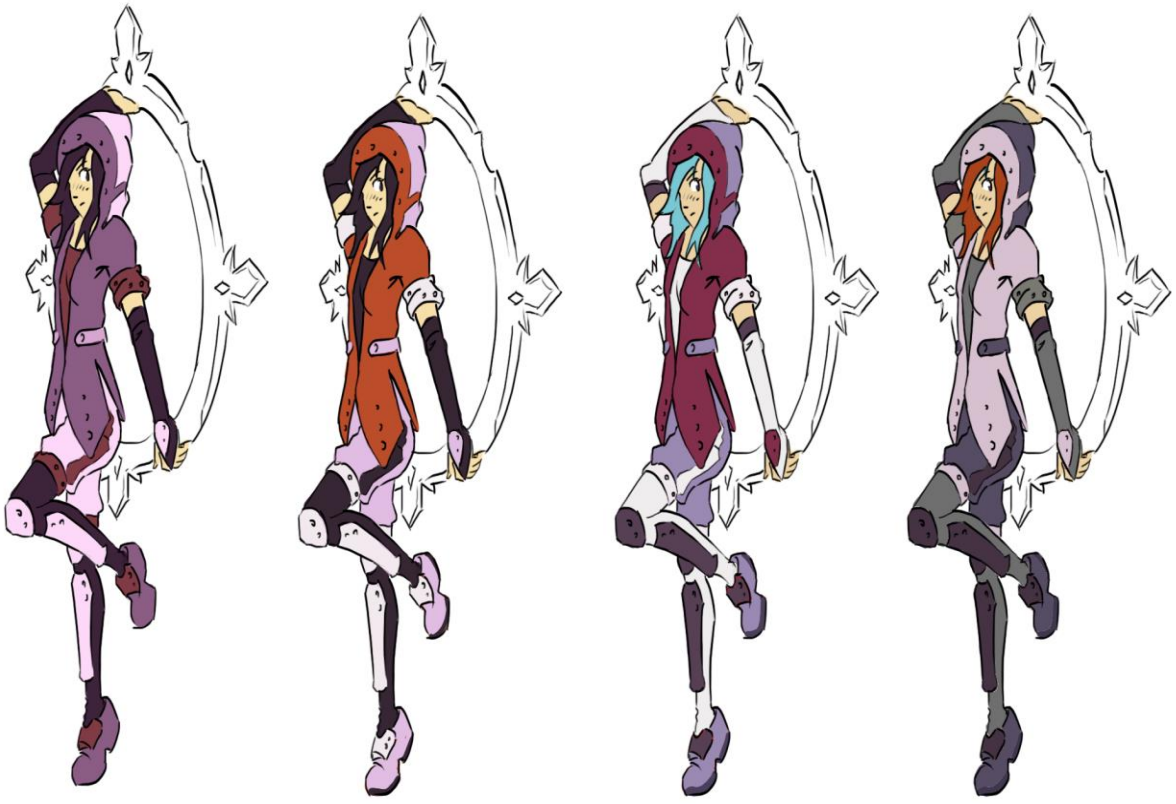

Fig 2.5 Korin's Final Concept Art

2.2.2 Character Modeling & Texturing

The modeling pipeline for the characters changed as we scoped down the assets required for a finished character. Ellsee and Velle had the most complex modeling process, which was done in order to create usable normal and texture maps during the modeling process.

1 Create a low poly base mesh in Maya, roughly shaped like the character in the concept art.

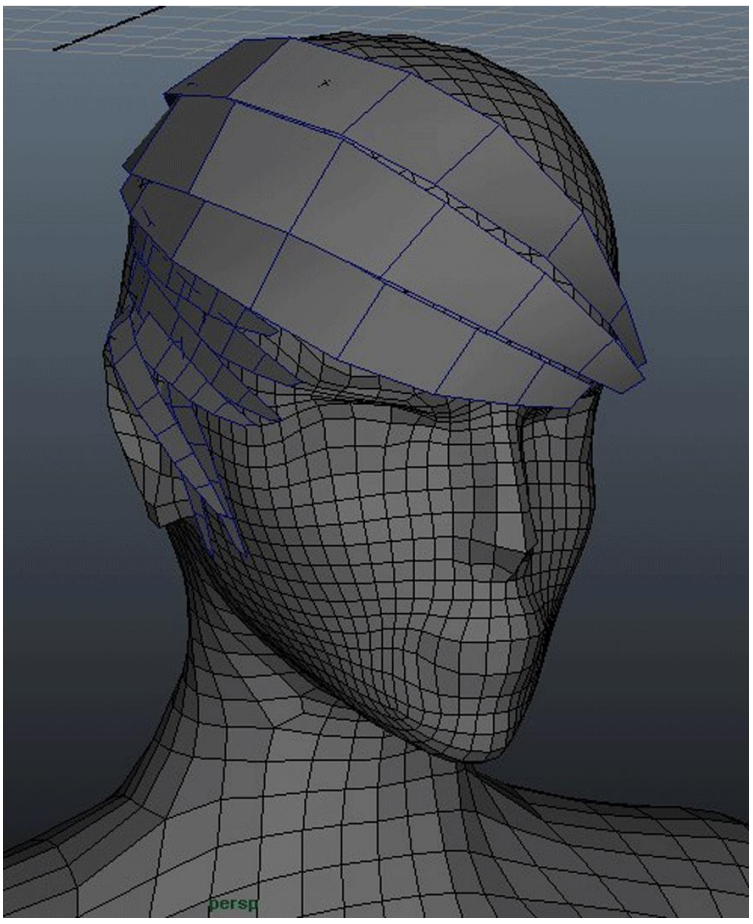

Fig 2.6 An example of a low poly base mesh

2 Import the low poly base into ZBrush, and sculpt in the fine details, work the proportions, and export the high poly mesh.

Fig 2.7 Ellsee's model, finished in ZBrush.

3 Bring the high poly mesh into Topogun and retopologize a new low poly mesh at about 15,000 polygons.

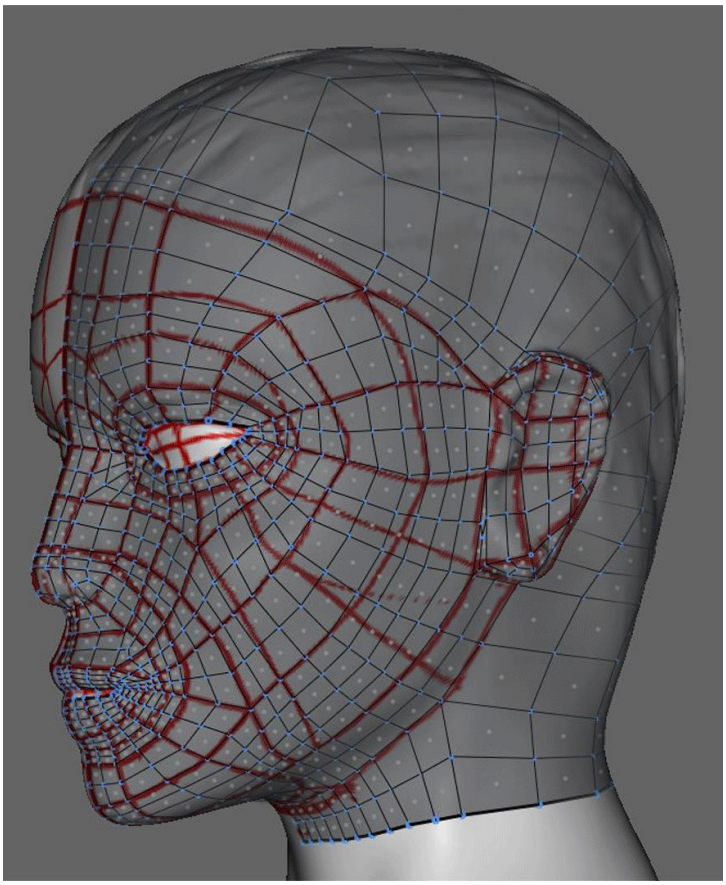

Fig 2.8: Ellsee's model undergoing the retopology process in Topogun.

4 UV Map the low poly mesh in Maya, then then create the textures for it in photoshop.

Fig 2.9: Ellsee's UV Unwrap and Texture.

5 Put all of it together in Maya, and get ready to begin rigging.

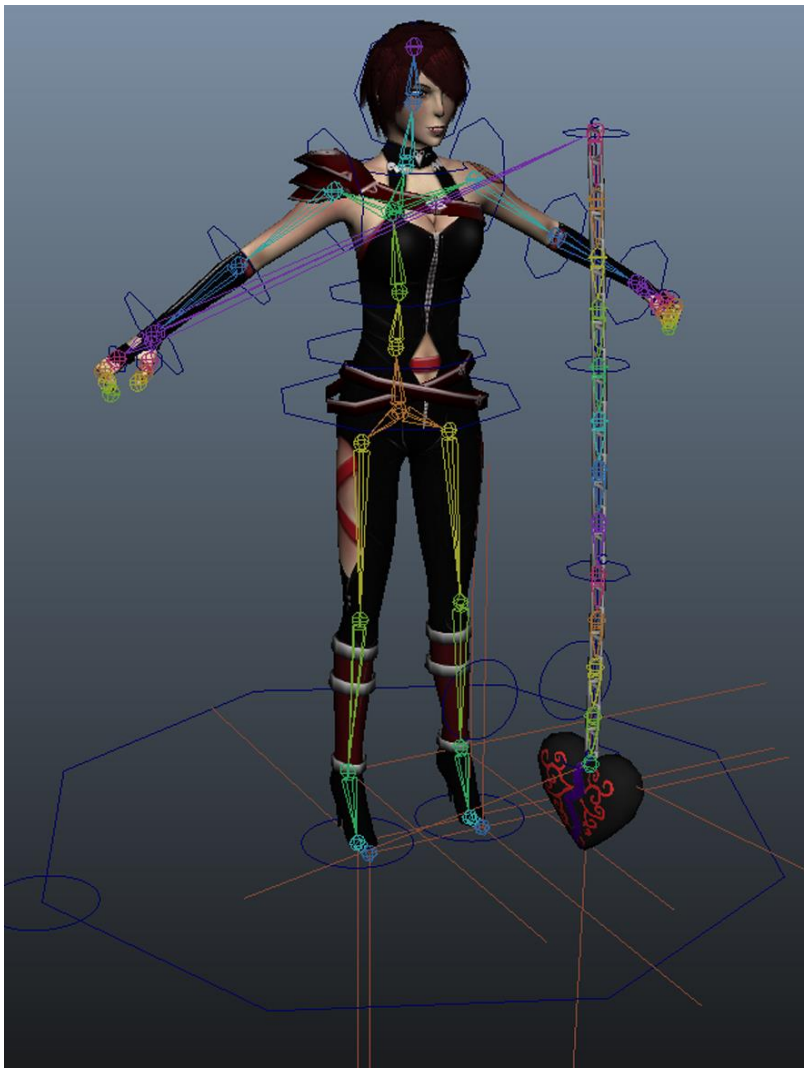

Fig 2.10: Ellsee's model, rigged and textured.

This process, although it afforded us a great deal of detail, took too long. Especially when there were at least 2 more characters that needed to be made, and several other animation, rig, and texture assets on top of all that. We had also decided that because of Unity's bug-prone

lighting, we were going to abandon the use of normal maps and begin the use of hand-painted textures. So when it came time to model Penguin and Korin, the process had been simplified to:

1 In Maya, create the low-poly mesh to total between 6,000 and 15,000 polys, essentially skipping all the steps in ZBrush from the previous method, and just working to make a more usable base model.

2 Use Face mode to select a region you want as a separate UV island, and go to Mesh -> Extract to make it into its own object, and much easier to UV. Do this as many times as you need to create a usable UV map.

3 Export UV map into Photoshop and begin texturing.

After creating Penguin and Korin in this method, it was determined that Ellsee and Velle stood out from them because of the differences in their model structure, and so they were revisited and touched up to be more like the newest models. Ellsee was also given planar hair maps by creating a single rectangular plane, assigning it a hair texture, and duplicated and resized and shaped over her head as necessary to get the desired hair style.

2.2.3 Character Rigging

This portion of the character creation process was by far the most difficult, but the most important thing to remember while rigging is that the more simple one can make the rig while still performing all the functions it is required to do, the easier it is to start creating the animations and easier to import into the engine.

Ellsee's rig required the most revision, due entirely to her weapon. After just giving it a 12 joint FK skeleton, it was determined that getting a realistic and fluid motion out of that set up took far too long than we had to spend per animation. The first real rig attempted to use the

"maya hair" physics to animate the middle sections of the chain between her arm and the end of the locket, but that functionality was removed in Maya 2013, so we had to move onto different options, eventually settling on SplineIK calculations. This way, we could just set a path using controls, and the locket would travel along that path. It did not give it a fully chain-like feel, but it was the best average we could manage with limited time and rigging knowledge.

Every character's rig used a simple, but versatile IK foot rig setup, allowing for ball roll and consistent floor contact. This was important because without IK functionality, it becomes harder to animate high speed movements in the legs and feet.

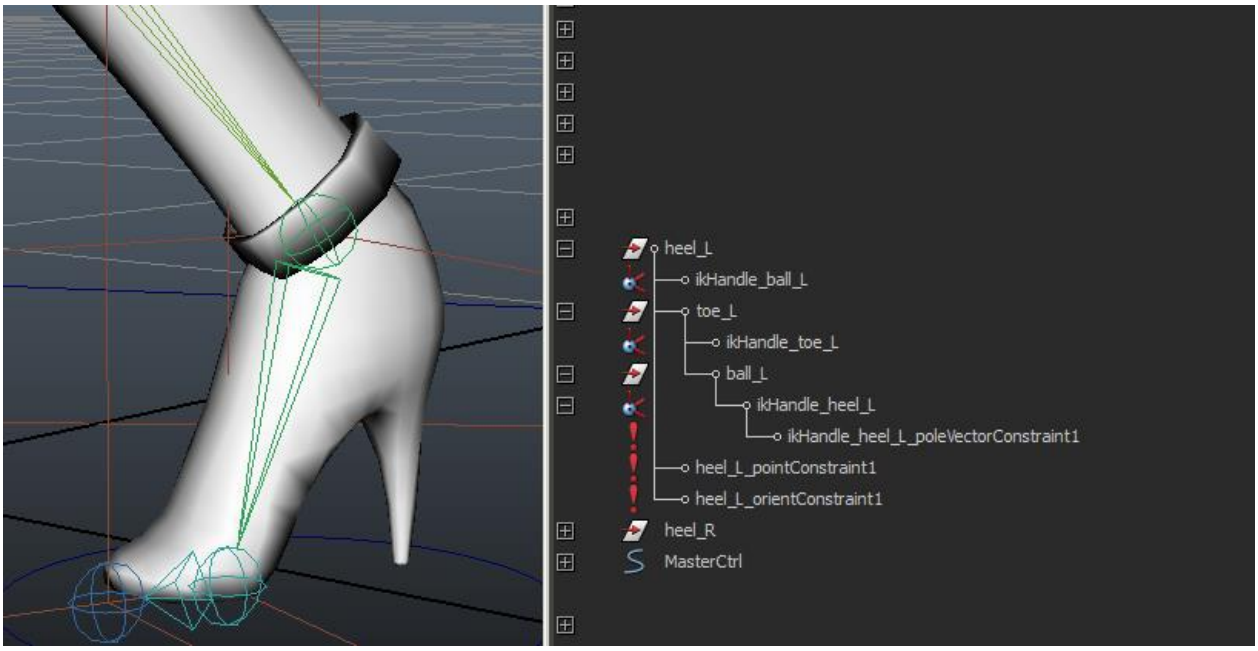

Fig 2.11: ikFoot Rig setup

Everyone else's weapons were either applied to them in game for functional reasons (Penguin, Velle) or just parented to a bone specifically for them (Korin), and worked right the first time.

2.2.4 Character Animations

The final, and most important, assets for the characters were their animations. There were roughly 35 animations per character, varying in length from 3 to 100 frames. Offensively, each character had 4 standing attacks, 3 crouching attacks, 4 aerial attacks, and a super. The stronger the attack, the more time it takes for the move to strike, so greater windup or bigger action was required. There were also all the mobility and defensive animations, like blocks, flinches, idles, dashes, walks, jumps, and more. For this portion a lot of reference was gathered from widely available fighting games and their frame data, such as *Street Fighter*, *Marvel vs Capcom*, and *Soul Calibur* series.

All the animation work was done in Maya, and special attention was paid to the perspective of the scene camera, viewing the animations from the same perspective that the in-game camera would be. If there was not enough time to tweak the animation to look good from all angles, then priority went to that camera angle. Unfortunately, with this being a 3d game, the camera in the actual game would shift slightly depending on the locations of the characters, and sometimes deform with the carefully crafted silhouettes, such as Ellsee's standing C attack (Fig 2.12).

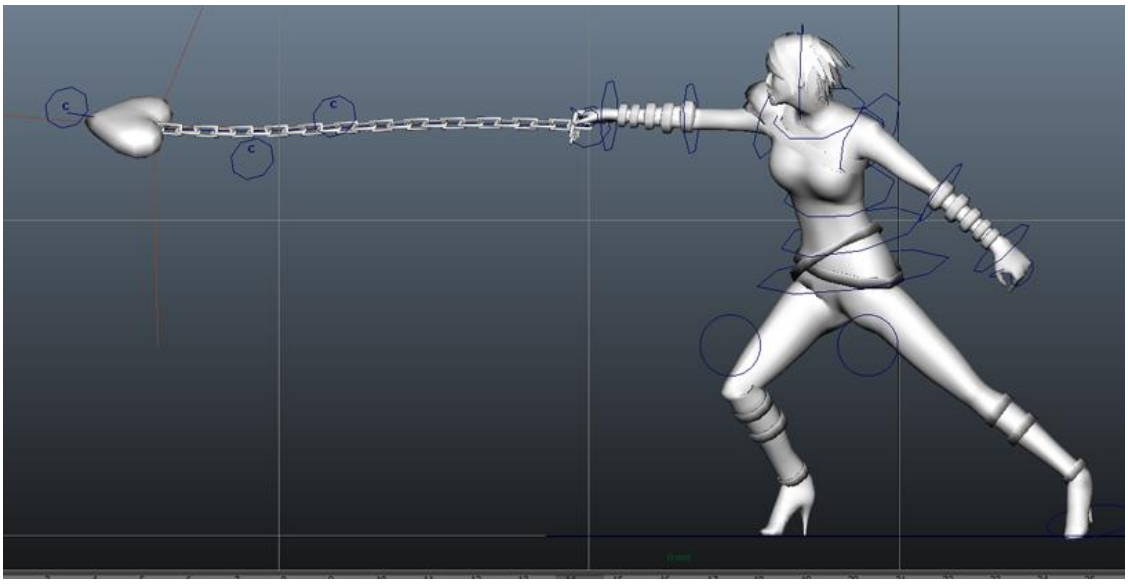

Fig 2.12: Ellsee in her "Standing C" attack.

2.3 Stage Design

The stages for the game were designed to be diverse as well as fun to play on. In the final version of the game, we ended up with five different stages. These stages were "Arctic", "Urban", "Disco Floor", "Dungeon", and "Training Room". With the exception of "Training Room", each stage has a character associated with it.

Fig 2.13: Screenshot of Arctic in its final form

Arctic, the home of Penguin, was the stage that was started first. As a result, it was also the first stage completed for the game. It also went through the most revisions before getting to its final form. The stage features a main platform made of ice that is surrounded by smaller ice chunks as well as two small ice platforms with penguins and snowmen on them for added life. These inhabitants move around the stage as the fight is going on. Bigger icebergs float behind these smaller platforms as well.

Fig 2.14: Screenshot of Urban in its final form

Velle's stage, Urban, saw 2 major revisions over the course of the project. The original and first revisions of the stage were on the ground and took place on a streetside basketball court with cars whizzing by and pedestrians taking in the fight. The second revision brought the stage to completion. The revision ended up being necessary due to the fact that having a busy, bustling city down below caused our game to slow down whenever it ran on the stage. This final big change saw the fight moved to the top of a skyscraper with a city skyline in the background.

Fig 2.15: Screenshot of Dungeon in its final form

Ellsee takes up residence in the Dungeon stage. Since the title of our game has the word "demon" in it, we needed a stage that had a demonic vibe to it. This was the third stage that was completed for the game. Composed completely of brick, the stage is lit only by the torches that line its walls and pillars. The walls are decorated with swords & spikes and a mysterious fog rises from the ground to give the stage a dark and dreary feel.

Fig 2.16: Screenshot of Disco Floor in its final form

The Disco Floor is the final main stage in the game. We were not entirely sure what the stage was when it was first started since it initially consisted of just a few tables and chairs. The stage now consists of a bar with stools, a dance floor that lights up (playing the game of life in the process), and a set of lights with a disco ball hanging from the ceiling. It is the most colorful stage in the entire game.

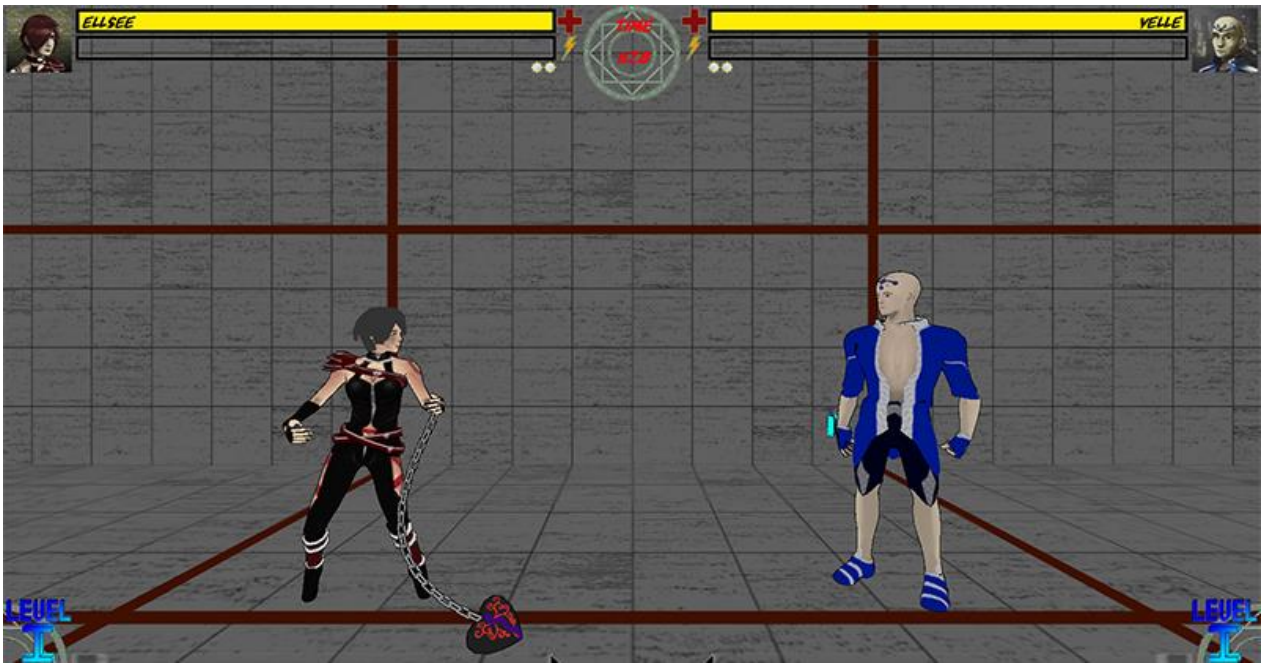

Fig 2.17: Screenshot of Training Room in its final form

After the four character stages, Demon Dissension also has a training room. While this "stage" can be selected to actually fight on, its main purpose is for our training modes. Whenever the player goes through a tutorial mode or a training mode, the gameplay takes place on this stage. As for the design, this stage is extremely simple. We created an old looking dojo texture and just repeated it several times.

2.3.1 Stage Modeling

The modeling process for the stages was pretty straightforward. After coming up with the initial concepts for a stage, the art team determined which objects would have to be modeled and made a list. All of the modeling was done in Autodesk's Maya software since the detail that Zbrush provides was not necessary**.**

2.3.2 Stage Animation

In order to breathe a little bit of life into some of the stages, some animation work was done to add another layer to each of them. The only examples of this that made it into the final game include the smaller penguins and snowmen in the Arctic stage. At different points in development, cars, helicopters, spectators, and even skeletons were animated with the intention of adding them to the stages. In the end, optimization was more important so these animations were scrapped in favor of making sure the game ran well.

2.4 Sound Design

2.4.1 Music

Music is one of the first things a player encounters that sets the tone of the game. All of the music in Demon Dissension was composed by the project team, and was created in Anvil Studio, a free MIDI creation tool. The music was composed for an ensemble of drums, electric guitar, bass guitar, and piano, and has a heavy rock feel. The rock style was selected as it meshes well with the nominally dark subject matter of Demon Dissension. Five music tracks were prepared for the game: one for each stage (the menu music is shared with the dungeon stage).

2.4.2 Effects

Sound effects play a very important role in a fighting game. Demon Dissension is no different. When it came to recording sound effects for the game, various punches, kicks, and particles sound effects were recorded using a microphone. This was accomplished by punching and kicking various pieces of furniture. After the initial recording process, the sound effects were brought into Pro Tools where they were mixed and mastered to sound more like the sound we intended them to before being brought into the game. Once in game, the punches and kick sounds are randomly chosen and played whenever a punch or kick lands on an opponent.

2.4.3 Voice Acting

The voice acting was a very necessary process for added a lot of life to the game and to the characters. First we would write up a small script consisting of all the different instances that a character would be speaking or making a noise, then we would send them off to the voice actors or actresses to review. Then we would meet in the recording room with the , record their lines, and then after a little remastering of the voices in Pro-Tools, the voice acting clips were ready to be brought into the game.

2.5 Asset Integration

Thanks to Unity3D's convenient art asset pipeline, the process of bringing in all the art assets was as simple as dragging and dropping. All of the character art was placed in folders to make navigation to specific assets easy. After it was loaded into the project, it was just a matter of letting the programmers write the necessary code and set up the prefabs for the characters so they had all their textures, hit-boxes, and physics controllers so that the characters were then usable in game.

3. Technical Methodology

This section of the paper discusses the technical implementation of our game. It is broken down by major subsections of the game, such as the control system, menu system, and the fight mechanics. Future MQP teams working on fighting-based games may find it useful to understand how the game was programmed and developed.

3.1 Engine Choice

Demon Dissension was developed in Unity, version 3.5. Unity is a modern, 3D engine that provides many capabilities vital to the game. Unity excels at importing art assets and animations, and as fighting games often have hundreds of animations, this was a benefit to the engine. Unity also has support for networking, which forms the underpinnings of our online system. Unity is also extensively documented, and both of the programmers on the project team had used the engine before. Having used the engine for four terms, it seems suitable for producing fighting games.

3.2 Controls / Input

Unity provides a simple and easy way to check for input, however, fighting games require additional controller support. For example, in many fighting games, there may be an attack that only activates if you sweep a quarter-circle around the control stick, or hold 'back' for a certain number frames. In addition, it is important for the controller to know the difference between whether a button is held down during a frame, or pressed on that frame. A Controls class was designed for the game that solves the above problems in a simple way. The next section refers to fighting game notation; described as follows:

- 1 - Press down and away from opponent.
- 2 - Press down.

- 3 - Press down and towards opponent.
- 4 - Press away from opponent.
- 5 - Idle (do not move the control stick).
- 6 - Press towards opponent.
- 7 - Press up and away from opponent.
- 8 - Press up.
- 9 - Press up and towards opponent.
- A - Light attack.
- B - Medium attack.
- C - Strong attack.
- D - Special attack.
- , - Used to separate frames: “8A” means press up and light attack at the same time; “8,A” means press up, and then light attack.

Any combination of the above is possible. For example, “6A” means press light attack while moving towards the opponent; “6,5,6” means tap towards your opponent twice, and “2,3,6,A” means a quarter circle towards your opponent, followed by a light attack.

3.2.1 Searching for Input

In the game, each character has a list of attacks, and each attack has an associated input string. For example, an attack might have “A” as an input, meaning when the light attack button is pressed, the attack is activated. Every frame, every attack in the characters move-list is examined; if a given attack’s input string was pressed, that attack is performed. This works perfectly well for input strings that consist of only a single button press, but for more complicated input strings, such as “2,3,6,A”, which span multiple frames of gameplay, a search algorithm is needed.

During each frame of the fight, input is polled and stored in an array - generally, about 0.5 seconds of input data is stored at any given time for both players. Each frame, the character's entire move-list is searched to determine which action, if any, was performed. During that algorithm, when the game requests a certain input string ("Did the player just press '2,3,6,A' "?), the input manager searches through previous input. To do this, it checks through the input log to ensure A was pressed that frame. If A was pressed, it checks to see if 6 was pressed a few frames before pressing 6. If 6 was pressed, it checks for 3, and so on, until either the whole string was found to be pressed, in which case the function returns true, or a different string was found, when the function returns false. This capability makes other tasks easier: for example double tapping right has the player dash to the right. Instead of checking if the player was recently walking right for a dash, the input can be checked for the string "6,5,6" (right, center, right), which represents the same action, but can be done in a single line of code. Although there is overhead with searching for every available action every frame, the algorithm minimizes it by returning false cases quickly. For example, if checking input for '2,3,6,A,' the algorithm first checks to see if A was pressed that frame; if A was not pressed, the function returns false, and continues onto the next input string without checking any other input.

3.3 Gamepad Controllers

Fig 3.1: Examples of Fight Stick and Gamepad input devices.

Fight games traditionally use arcade sticks to control the onscreen characters. They consist of a large ball handled joystick that can fit into a palm instead of just a finger, along with a few larger buttons for the other hand. Arcade sticks are relatively expensive, so we don't expect everyone to have one. Instead we decided to focus on support for game controllers. We tested and developed the support with Microsoft Xbox 360 Controllers for Windows.

Unity returns input from axis on a scale from -1 to 1. Having two axes on each stick, we are able to determine the angle the stick is pointing. This angle can be mapped to the 9 number grid systems we developed. We are able to set flags this way about what directions the stick is pointing, as well as mark if they input direction has changed from the last check.

Unity provides built in support for joysticks, but we encountered a few issues. First, when the controllers are plugged into the PC, Windows will recognize them and install the proper drivers. Windows then assigns the controllers a number based off the order they were plugged in. However, Unity does not seem to recognize Windows' ordering. The solution we came up with was to assign out own decision of who was first player and second player,

independent of both Unity and Windows. On the title screen, the first controller to select a menu option would become who we considered to be first player.

3.4 Fight Mechanics

Even though fighting games only consist of two characters fighting, they often have very complex and deep mechanics, and *Demon Dissension* is no exception. This section describes the technical capabilities of the game that bring the fights to life.

3.4.1 Movement

Movement in our game is accomplished through the use of kinematic rigid bodies - a Unity feature that can make objects affected by gravity, but not directly affected by external collisions. This is useful, as the amount of knock-back an attack provides should be able to be set by a designer, as opposed to being inferred by the physics system. Constraints were placed on the character rigid bodies - they are only free to move in the X-Y plane, and their rotations are fixed. This is to prevent a character from being knocked in the Z direction by an errant projectile, and therefore no longer facing their opponent. In addition, if rotation was not fixed, a character could be spun around or knocked on their back like a turtle, and since this is not an intentional state, the character would be unable to get back up. Lastly, for a large portion of the design process, characters could collide with each other, however, this behavior proved unsuitable, as the characters could get stuck on one another, or jitter back and forth every frame as the collisions attempted to resolve. To solve the problem, we prevented characters from colliding with each other at all (a helpful Unity feature), and manually determined that if characters were taking up the same space, to push them apart until the distance was adequate.

3.4.2 Hit-boxes

In video games, collision checks are often processing-intensive. It would be impractical to check for a collision on each of a character's thousands of vertices, especially as the character

is animated and constantly moving. A common workaround is to place hit-boxes - collision boxes that follow the general shape of the character but do not match entirely - on each of the joints of the character. For example, a character might have a cube covering each hand that determines collisions as a helpful approximation. Characters in Demon Dissension have four types of hit-boxes that perform different actions. In the following diagrams, the green boxes surrounding hit-boxes designate which hit-boxes fit in the given category.

- Attack Hit-box - these hit-boxes are attacked to 'attacking' joints, such as fists, feet, weapons, etc. These hit-boxes can be turned on and off (see Attack Implementation); when they are on, they do damage to opponents they touch.

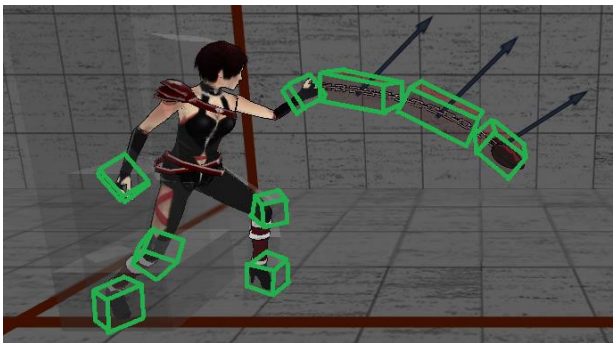

Fig 3.2: Examples of attack hit-boxes on Ellsee.

- Collision Hit-box - these hit-boxes take collisions. They are attached to parts of the body that absorb damage, such as the body, head, and legs. When they are hit by an active attack hit-box, they inform the character that it was hit.

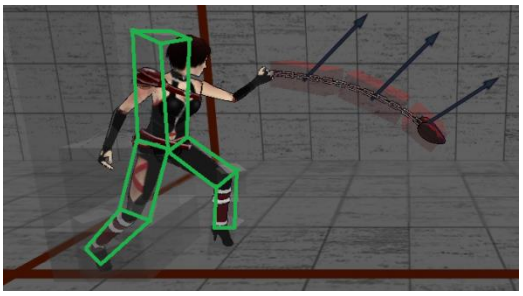

Fig 3.3: Examples of collision hit-boxes on Ellsee.

- Push-box - these hit-boxes are large, and encompass the whole character. When two push-boxes collide with each other, the characters are pushed away, thus preventing two characters from occupying the same space.

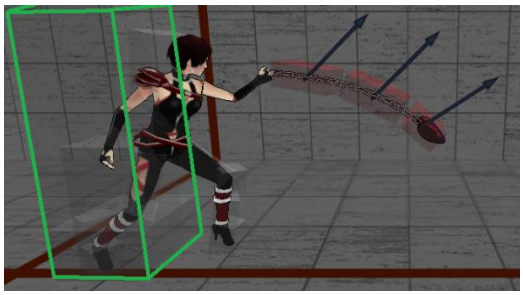

Fig 3.4: Examples of push-boxes on Ellsee.

- Bounding Box - these hit-boxes are small, and are placed at the feet of each character. The bounding box prevents characters from falling through the floor.

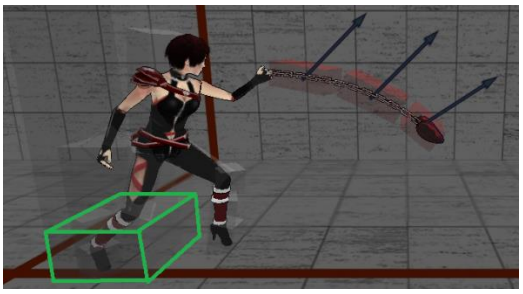

Fig 3.5: Examples of bounding boxes on Ellsee.

3.4.3 Attack Implementation

No fighting game would be complete without a wide array of attacks and capabilities for each character. Since attacks can vary wildly between characters, a system was needed to create all sorts of different attacks, many of which were not known when the system was designed. In Demon Dissension, attacks are represented as a linked list of Attack Events, which represent different occurrences that take place during an attack. Each attack event has a frame

representing the time it activates, and the action to take. For example, a sample attack might be this:

- Frame 15: Activate hit-box on right hand with a damage of 15.
- Frame 30: Deactivate hit-box on right hand.
- Frame 45: Null event.

When the attack activates, it begins on frame 0. For every frame it increments by one, until it reaches the given time of the next event, 15. On frame 15, it pops the first event off, and performs the action (activates the hit-box). It then continues on until frame 30, when it pops the next event off and disables the hit-box. It then continues until frame 45, when the last event is popped off, and the character is free to attempt another action. The purpose of the null event is to lengthen the attack; once a character attacks, they cannot move until the attack finishes or they cancel out of it. The null event therefore lengthens the time until they can move again, which synchronizes with the animation; for example retracting the player's fist from a punch attack.

There are several kinds of attack events possible in the game, as follows:

- Hit-box Event - enabling or disabling a hit-box with a certain damage and knockback value.
- Projectile Event - spawn a projectile at the given relative position and velocity.
- Throw Start Event - put the hit-box in the throw state (any character that is hit by it will be thrown).
- Throw Check Event - if the character attempted a throw and the throw did not connect, prematurely end the attack, since there is nothing to throw.
- Throw End Event - put the hit-box out of the throw state.

- Knockdown Event - put the hit-box in the knockdown state (any character that is hit by it will be knocked down).
- Special Event - an event that uses C# delegates (similar to function pointers) to determine the action to take. This could run any code, which makes Special Events a catch-all for special abilities only one attack will have, such as Velle's Dagger Return attack, which informs all daggers on-screen to return to him.
- Null Event - an event that has no behavior, but can extend the length of an attack as mentioned above.

3.4.4 Finite State Machine

In fighting games, the actions a character can take depend on their current state. For example, a character in the air cannot perform ground-based attacks, and a character who is stunned cannot move or block. Therefore, a system needs to be in place to keep track of a character's current state, and the actions they can perform. A finite state machine was used for this purpose. Each character has a current state which updates the character, and methods to change the current state of the character. For example, in the idle state, the code might look like the following:

```
if( pressed jump button ){

    Transition to State (aerial state)

}
```

The full list of states and their utility is as follows:

- Active Block State: the player goes into the active block state for a few frames any time they block - for example crouch block, reverse block, or aerial block. If they are hit in this state, it counts as an active block, and does no damage.
- Aerial Attack State : same as attack state, but in the air.
- Aerial Block State: same as reverse block state, but in the air.
- Aerial Dash State: same as dash state, but in the air.
- Aerial Knockdown State: same as knockdown state, but in the air.
- Aerial State: analogue to the idle state, but in the air. From this state the player can move forwards or back, use aerial attacks, or block.
- Attack State: when the player is using a ground attack. The player can only cancel into other attacks in this state.
- Crouch Attack State: same capabilities as the attack state, but for crouch attacks.
- Crouch Block State: when a player presses down to crouch. The player can perform ground attacks in this state, or release down to return to the idle state.
- Dash State: when the player double taps left or right on the ground, they enter the dash state. Same capabilities as idle state.
- Dizzy State: when the player is dizzied. The player can only cancel out of this state, or wait for it to end automatically.
- Hitstun State: similar to the dizzy state, the player cannot move for several frames after they get hit, unless they cancel out of it.
- Idle State: when the player is on the ground, or walking forwards. Allows the player to jump, crouch, move, or attack.
- Knockdown State: when the player is knocked down; same capabilities as dizzy state.

- Null State: before the match starts the player is in this state. This state automatically loops the idle animation, and cannot be cancelled out of.
- Reverse Block State: when the player holds back from the opponent. Same capabilities as idle state, but the player takes reduced damage.
- Super Null State: when a player is defeated, they are knocked down, and go into the super null state. This state has the same capabilities as the null state, but the idle animation is not played.

3.5 Networking

Networking was one of the first features we had wanted to tackle when making our game. Games have been online for many years, but only recently has the fighting genre moved to support networked play. GGPO is used for a few commercial games, so we started examining that library. After understanding how the system worked, we decided it might be a little too complex for the needs of our game. GGPO does predictions about the opponent's moves, but we decided that this was unnecessary for our game and decided to focus on some of the other aspects.

3.5.1 Connection Methods

Networking in Unity is done by clients connecting to one instance of the game acting as a server. By entering the IP of a player who has started a server, another user is able to start a networked match. This however, is tedious, and requires that two players know how to look up their IP addresses. The alternate method is to use a master server. The master server is always running on a remote computer and has a static IP address. When a player starts a server, it registers the game to a centralized server. It stores the IP address and port of the server, as well as the name from the active profile from our account system. When a person requests a list of open games from the server, they are presented with a list of games that have open spots. The

player can then select the game with the name of the person they want to play against. When a connection is made, both the client and server can start selecting characters and level, and then they are loaded at the same time.

3.5.2 Level Loading

Loading into a different scene across all networked clients in Unity requires one library function call. However, to do this properly requires that each client stop processing messages while the level is being loaded. New messages that interact with game objects should not travel across the network. These problems were solved two ways. First, levelprefixes are used to prevent old updates from being sent objects into a newly loaded scene. Each time, the objects have a level prefix, and they will only receive updates if the incoming message prefix matches the objects levelprefix. Second, each message is locked into a group. We set all the logic for the game message to be in one group, and we could use a second group for other messages. This would allow us to still communicate between clients during loading, but not allow any message that affect game logic to be mixed in.

3.5.3 Keeping Games in Sync

Unity transmits information using Networkviews. Each Networkview has a corresponding NetworkviewID. Unique IDs are constant across all instances of the clients. Each time the Networkview syncs, variables that we chose are synced across the network. When sending, the variables are just passed through and assigned directly without modification. When received serialized data from the transmitted BitStream, all the variables except the transforms are passed through. Instead, the transforms are added to a queue. At this time, the current position of the remote player is interpolated to a point between the current and previous locations in order to keep the game looking smooth instead of teleporting. If we have not received any

new information, the location of the remote play is extrapolated based off previous data that has been collected.

3.5.4 Reducing Total Network Traffic

UDP is used on the back end for sending network messages. Unity provides two different settings for the way updates are pushed across the network. The first is the delta reliable compression. This method uses ACKs and NACKs to enforce packet reliability and order. It also looks at the previous information was sent, and only sends changes since the last time. The other method is the unreliable method. This sends the entire packet each time and does not force packet order or if the packet is even received. We decided to use the unreliable method since forcing packet order could cause a small bit of delay. The position of the players changes quickly enough that missing a packet is not a large enough problem.

Our program uses about 25kb/s. For our program, the difference in unreliable and delta compressed is negligible when comparing how much data is sent during a game. Animations are being synced across the network, and this uses the most network traffic in our game. The sendrate for our packets is 20, so a new packet is send every 50 milliseconds.

3.6 Artificial Intelligence

3.6.1 Overview

Fighting games excel in multiplayer gameplay – pitting human opponents together to determine who is superior. However, there are frequently times when a player would want to play the game by themselves, and Demon Dissension provides an AI to play against. Unfortunately, an AI with one level of skill may be too easy for some players and too hard for others. Therefore, multiple skill levels of the game AI would need to be created, however, this would be a large coding problem. It is not intuitively obvious what changes to an AI would

make it function better or worse, and the algorithm would be split among several large files, which would be hard to change. To circumvent this problem, a learning algorithm was devised that would enable the AI to improve over time. The algorithm keeps track of which tactics and moves would work best in certain situations, and can perform those more often overall. In addition, an AI could be tweaked in code to perform the best action all the time, or generally strong actions, or generally weak actions. Therefore, with the learned AI data, several different skill levels of the AI are implicitly created.

3.6.2 Learning Methodology

In order to create an AI that learns without human intervention, we needed to create a way for the AI to learn new information, and a format to store that information. For the AI to learn, it has a concept of what state it is in. The state of the AI is based on what state the character is in (ground, crouch, or air), what state the opponent is in (attacking, idle, or blocking), and how far away the opponent is in both the X and Y directions. Each state is mapped to a four digit number, for example State 0135 refers to a state where the player is on the ground (0), the opponent is attacking (1), the opponent is 3 units away in the X direction, and 5 units away in the Y direction. Each state also has a list of associated actions it can take, and the expected utility of that action. For example, the AI for state 0012 for Ellsee is as follows:

State:0012
Attack A#A#56.2201
Attack C#C#18.66991
Attack B#B#10.16989
Double 6#6,5,6#7.501622
6 20#6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6#7.308132
Attack D#D#-3.902193
7 20#7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7#-3.936279
QCF Special A#2,3,6,A#-4.084016
4 20#4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4#-4.156428
QCF Special C#2,3,6,C#-4.404858
Special Test#2,3,6,AB#-4.411794
Temp Throw Attack#6C#-4.415959

9 20#9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9#-4.827996
QCF Special B#2,3,6,B#-4.878638
8 20#8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8#-5.028845
Double 4#4,5,4#-5.318102
2 20#2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2#-6.672045

Each line besides the first corresponds to an action name, the input string for that action, and the expected utility of that action, with the '#' character as a separator. Therefore, the line "QCF Special A#2,3,6,A#-4.084016" means that the action is called QCF Special A, that to perform the attack the player presses "2,3,6,A", and that the action is currently rated -4.084016. Since the rating is not positive, it means the action was generally not beneficial for Ellsee in that state. The list of actions is ordered by utility; actions towards the top of the list are generally better actions. It should be noted that input strings with a large amount of numbers (for example, "2,2,2,2,2,2,2,2…"), have the AI press that button for the given amount of frames. In this case, that tells the AI to crouch for 20 frames before deciding the next action.

For the AI to learn, it performs the following algorithm. First, based on what the opponent is doing and how far away it is, it determines what state it is in. It also records what the health of the two players are before the attack. Then, it picks a random action from the state's list of possible actions, and performs it. After the action completes, the AI determines the utility of that action. The utility function is based on the health of both players (if the opponent took damage the action was good; if the player took damage the action was bad), and the total utility of the new state. For example, moving backwards could cause the character to switch states, and if that state was not as useful (for a short-range character), the algorithm would pick up on that. The algorithm would then update the utility of the given action to reflect the new data. Therefore, as the AI plays, it gradually learns which actions work better in each state, and can learn to perform those better actions more often.

Once the AI learns which actions are generally beneficial, it is removed from learning mode. In this mode, instead of picking a random action to test, it picks actions with high utilities, to make a challenging opponent. Additionally, in this mode obtained data is discarded, to keep the AI at the same skill level for players. This is to prevent players who play the game constantly from accidentally developing an AI they cannot beat, which would be frustrating.

An interesting problem was encountered in designing the AI systems relating to the number of states to use. In the final version, there are three possibilities for player state, three for opponent state, ten for x distance, and five for y distance. Therefore, there are 3 x 3 x 10 x 5 states, for a total of 450 states. In addition, each state has about 15 actions, for a total of 6750 actions. For the AI to be effective, it would need to reach each of those actions several times to refine how successful each action is. Assuming each action can be attempted in one second, and three iterations on each action would be necessary to refine the AI, it would take over five and a half hours minimum to train the AI for a given character. Although this is not unreasonable, increasing the state space slightly would increase the training time drastically. For example, increasing the number of player states from 3 to 5 would increase the total number of states to 750, and the training time to over nine hours. Other state types were considered, such as a state for player health and opponent health. However, even with just three discrete values for health (high, medium, low), the total number of states would be 1350, with a minimum training time of almost seventeen hours. This was considered to be unreasonable, and since the goal of the AI is to perform the most damage over time, having different behavior at different health levels was superfluous. The number of states was minimized to have the AI perform better actions with less training, instead of having a longer training time but more flexibility. After the AI finished learning, the data files were finalized and baked into the game.

3.6.3 Evaluation

A procedure was developed to test the effectiveness of the Learning AI. First, the AI for each character was created, and trained for between 7 and 8 hours against multiple characters. Each hour, the training was stopped, and a time-slice of the AI at that point was saved. Therefore, multiple versions of Ellsee's AI were saved from different levels of training. Then, a scene was created that would play the trained AI against a control, to see how much damage it would do over a period of ten minutes. Every fifteen seconds, the amount of damage the AI took and received would be printed to a file, which was then brought into Excel for analysis. For evaluation, five versions of Ellsee's AI were tested (0, 2, 4, 6, 8 hours of training) against Velle, with a controlled 0 hours of training. The given characters we tested were not important, as the AI should work equally well for every character. A graph was then created which plots the effectiveness of each AI (ratio of damage given to damage received) over time.

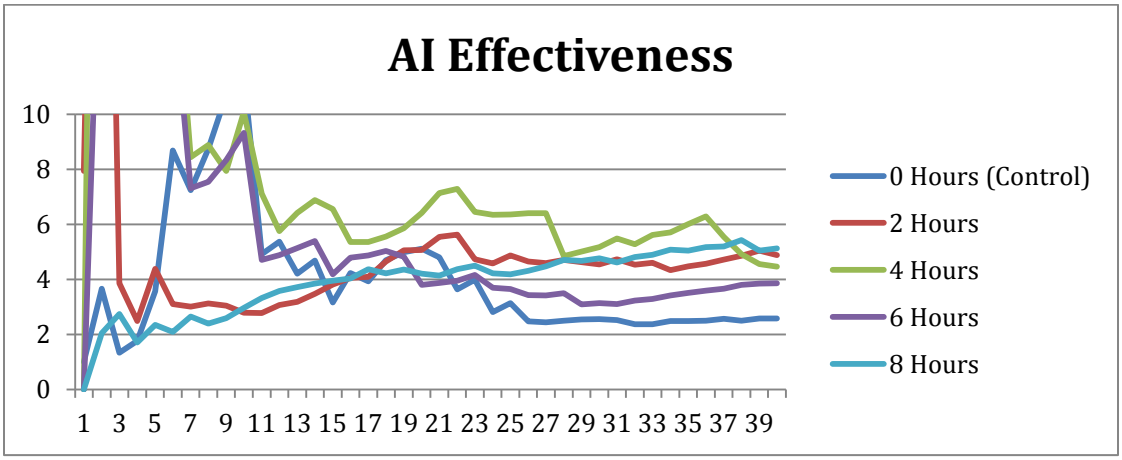

Fig 3.6: Graph of AI effectiveness over time.

In the above graph, the value at each point represents the ratio of damage given to damage received; higher values indicate a stronger AI, since they will generally do more damage and receive less. After the values level out, there is a clear ordering for the AI; the control performed the worse, with a ratio of 2.58. The AI with six hours of training had a ratio of 3.85; four hours

had 4.46; two hours had 4.88, and eight hours had 5.12. The AI with the most training performed the best, but the middle three came out of order – therefore more training did not always lead to a stronger AI. This could be for several reasons. Since all of the trained AI performed better than the control, it could be the case that training AI works well for a few hours, but fails to continually improve with more training. Also, since both AI's behaved according to random chance, different runs of the testing scene would result in different data. As the ratios tended to level off around the five minute mark, the effect of random chance was diminished, and running the simulation for longer would create more definitive results. Overall, as all forms of training were stronger than the control (ratio of 3.85 for weakest trained AI; 2.58 for the control), the algorithm was successful.

3.7 Attack Editor

3.7.1 Rationale

During initial development of the game, attacks were created solely in code. The move-list initialization function would create all of the attacks, denote when hit-boxes turn on or off, and describe how much damage and knockback each attack does. However, this approach was quickly found to be unsatisfactory, since in order to modify an attack, the game had to be shut down, edited, recompiled, and restarted. The time spent restarting the game quickly added up, and small modifications to attacks, such as changing what frame a hit-box activates on, took much longer than necessary. In addition, programming attacks in code made them harder to change, especially for the artists on the team. Lastly, as the amount of attacks on each character grew, the amount of code for each character became large - several hundred lines, which is

generally not a good design practice. To solve these problems, a tool was designed in the Unity engine to dynamically create and modify attacks in real-time.

3.7.2 Implementation

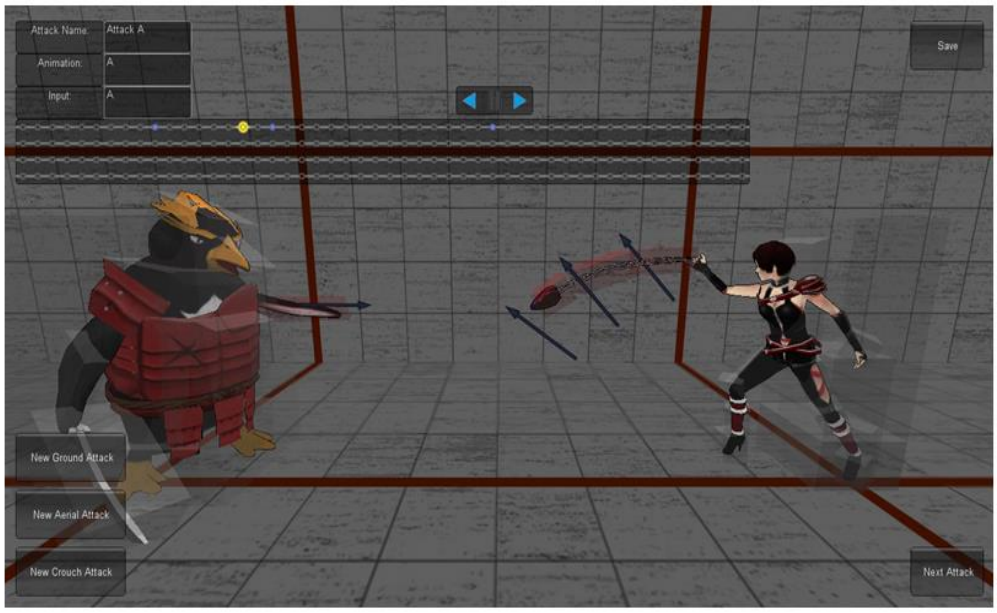

Fig 3.6: An example of the Attack Editor in action

In the above screenshot, Penguin and Ellsee are fighting in the Attack Editor. The red and white boxes that cover the characters are the hit-boxes that turn on and off during attacks. The arrows coming off of Ellsee's chain represent what direction Penguin will fly if he is hit by them. The four horizontal lines near the top of the screen represent a timeline of the attack; the timing of hit-boxes can be adjusted there. The Attack Editor provides capability to edit attacks in real time and view the consequences. The sequence of events on each attack can be adjusted: the length of the attack can be lengthened or shortened, or individual attack events can be placed on different frames. The amount of damage each attack does can be adjusted, as well as the amount of knockback each attack does. In order to implement this system, data from attacks had to be stored somewhere separate from code. To do this, the data from all attacks was saved in

XML. This solved two design problems with the game. First, it prevented each class file from becoming too large. Second, it allowed real-time editing of attacks. After the attack editor tool is used, the character's new move-list is saved back into an XML file. Then, when characters are initialized, they obtain the new, updated move-list from the XML.

3.7.3 Utilization

The Attack Editor system was used extensively in character creation. When the animation files for each character were created, the character was brought into the Attack Editor for initial placement of hit-boxes, knockback, and damage. Later in development, the tool was used to balance each character's move lists, so no character was drastically more powerful than any other.

3.8 Camera System

There are two different camera systems that were implemented in the game. The first camera system is used on the menus. For each collection of menu buttons, there is a camera point that points to each collection. When the camera gets a camera point to move to, it lerps position and actives the correct button collection.

The other camera system is the fight camera during a match. The camera adjusts itself dynamically during the fight. While fighters move away from each other, the camera zooms out to keep both characters inside the visible area. At a point, the camera locks and prevents any more backwards movement of the characters. The camera also has a minimum distance, and pans with the fighters, but does not zoom in any closer.

3.9 Menu System

The menu system in Demon Dissension went through several iterations. Initially, the menu system was created in code using statically-sized boxes; however, the approach was not very flexible. For example, a 400x200 button on the screen would be a fifth of the size of a large 1920x1080 monitor, but would take up more than half of the screen on a 800x600 monitor. Since the initial approach does not work well for monitors of different sizes, an addition was made that scales the buttons and text of the menu system based on the current screen size. For example, instead of saying "make a 200x100 button", one could say "make a button that is 0.2 times the width of the screen x 0.1 times the height of the screen." The second approach, while more flexible, is also hard to change, as any edits to the placement or size of buttons must be done by editing raw C# code. To solve the problem, the creation and placement of buttons was done in the game world, instead of in code. This allowed the artists in the group to adjust the positions, sizes, and colors of the buttons in an easy way, and code was only used to determine what happens when each button is selected.

There are several types of menu buttons in the game. The first allows the player to move up and down to different items in a list, such as in the main menu screen. The second allows the player to move in any direction; each button may have buttons to the left, right, top, or bottom. This type of button is used for the character select screen. The last type of button is a slider, and allows the player to move back and forth between two set amounts. Slider buttons are used in stat allocation, and selection of game options. Buttons generally use C# delegates to accomplish tasks; each button has a delegate associated with it that is called when the button is selected. For example, pressing the "Vs. AI" button in the menu calls a delegate function that brings the player to the character selection screen.

4. Playtesting

Throughout the project, we held unofficial testing sessions to make sure the fight engine and menus behaved as they should. This would happen whenever we added a new character, modified hit-boxes, damage, animations, or anything else. When testers from outside the team played *Demon Dissension*, they were shown the controls and encouraged to experiment with different characters, different moves, and different stat spreads after each round. This was effective for finding bugs because the players tended to just try to do everything, because they effectively had no idea what they were doing, and therefore were not playing very carefully. There was largest demo of *Demon Dissension* over the entire PAX East weekend, allowing over 200 unique players to get their hands on the game and test how easy it was to pick up and play, and gauge how fun they felt the game was.

However, playtesting could never be effective for balancing the game, due to issues with the varying levels of character asset integration and difficulty getting physics to work in a playable state. The issues with the physics did not make the game unplayable, but how far it was from where it needed to be would change how the game plays completely. In addition to that, when we finally our testers tended to not understand how to play our game on a basic level, or just not have any fighting game experience. One common problem we observed during the demo periods was when a player who is constantly walking into their opponent's weakest attack for the entire match complains that their opponent's character is broken; the data isn't all that useful. The second problem was that our characters were not ever completely finished, especially as we added more animation slots to characters later, the previously 'done' characters now needed more animations or attacks to match up, or there were outstanding bugs that had yet to be repaired. Most commonly was the problem that a portion of the fighting engine was not

available or complete yet, and so any balance data based play tests before its addition had the possibility of being rendered inaccurate by the increased amount of options a player had.

5. Post-Mortem

In a year-long, collaborative project, there are plenty of opportunities for unexpected challenges to arise. This section describes the successes and challenges of Demon Dissension's production, and contains suggestions on how to make future MQPs run smoother.

5.1 What Went Right

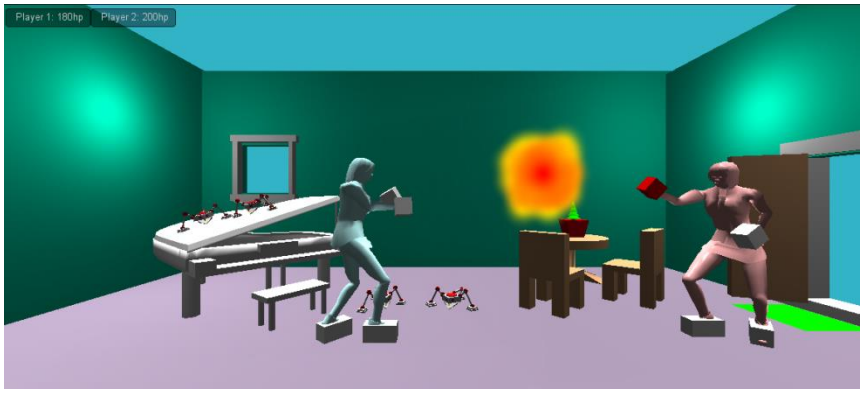

Fig 5.1: An early (May 2012) Screenshot of Demon Dissension

A major reason for our success was the amount of work we did over the summer before A-term. A test character model was created that could move, jump, and attack, and was implemented in the game before the MQP officially started. In addition, we had a playable version of the game during A-term – placeholder characters could move around and use a simple punch attack to damage each other (Fig 5.1). Since the game was playable so quickly, we were able to quickly add features every week, which contributed to the overall content of the game. Similarly, we scoped the game well (decided on four characters in A term, which was met exactly), which allowed us to easily evaluate our own progress towards milestones. Therefore,

when a milestone was easier than expected to complete, we could work on extra, unplanned features in the extra time (this allowed us to create the attack editor, and other features).

Unity was also a success for the project. The entire project team had used Unity in some capacity for the project, so the tool was familiar, and there was no learning curve necessary to get up to speed. In addition, Unity worked well with art integration; with the exception of a few hiccups, all the models, textures, and animations were imported seamlessly.

Our team also worked well together. Three of the team members had worked together on an IQP, and as a result the group dynamic started out in the right direction. Since we were comfortable working with each other, we were not afraid to mention things we did not like about the project, which overall improved the quality of the project. We communicated very well - a Facebook message chain was created in April-May of 2012, and any time we needed a quick response, a message could be posted there. As of April 2013, over 10,000 messages were posted to the chain, thus emphasizing the amount of communication present in our group.

5.2 What Went Wrong

There were several challenges, both technical and artistic, that came up over the course of the project. The physics system in the game was difficult to implement - we desired a system where players could quickly change the direction of their characters, but would also be affected by knockback. Early attempts at this resulted in characters that were too 'floaty', and that jittered back and forth when close to their opponent. In addition, since hitboxes were tied directly to the animations, the attacking joints needed to be directly lined up along the Z-axis, or they would miss the opponent. For example, if a player punches to the right of their opponent's head, the attack would miss, but from the perspective of the camera, the attack should hit. Several animations needed to be fixed to solve this problem. Collision detection was a big

problem during the first couple terms of the project as well - without a hack solution, player one could hit player two, but player two could never hit player one. The hack solution was eventually replaced by a more robust solution during C-term, when the correct way to handle collisions was found. Lastly, networking was a major part of our project, but deadlines tended to slip, and overall it took much longer than we anticipated. Since the networking fight was built on top of local fight, local fight had to be completed before networking could be started, which extended the time it took to complete.

Artistically, there were several problems that occurred during development. Ellsee's chain was difficult to animate, and took several iterations to get right. Additionally, her air C attack refused to work, and was not fixed until D-term. Penguin's sword would also occasionally get detached from the joint, and would end up outside his hand. To fix the issue, we imported a separate model of the sword, and attached that to the joint in Unity. From a design standpoint, it was determined in C-term that the stage artwork and the character artwork looked like it was done by two separate artists, which it was. To unify the art, a toon shader was placed on both the stages and the characters to give them a similar feel and brightness. In addition, the toon shaded characters had outlines which made them pop out of the background on all of the stages.

5.3 Lessons Learned

In a large project such as an MQP, scoping is vitally important. It is generally a more enjoyable and satisfying experience to add extra features to a smaller project than to take features away from a big project. In addition, planning is key; even though deadlines will not be met exactly as specified in the plan, it is a great organizational tool. Overall, it was useful to have a working prototype in A-term, as it helped us to find the 'fun' in the game. Other teams may find it helpful to have a prototype of their game done early for similar reasons.

6. Conclusion

The *Demon Dissension* team accomplished an effectively deep fighting game, a game where two players enter in combat with one another and battle to reduce their opponent's HP to 0. The development process was iterative on our early prototype, adding in character assets as they were completed and implemented. In the end, we had a fighting game with a unique and fully functional stat system, arcade mode, local and online versus modes.

Demon Dissension is playable with Keyboard or Xbox 360 controllers on its download page, www.demondissensiongame.com. There are always more characters to add, more animations to be made, sharper models and textures to create, and physics to rein in, at the current state this MQP is an entertaining competitive multiplayer experience that showcases what is possible with a balanced set of skills and team synergy.

Bibliography

Cannon, Tony. "Fight the Lag!" Game Developer Magazine. Sept. 2012: 7-13. Print.

“Fail-safes in Competitive Game Design: A Detailed Example” *Sirlin* 9 June 2012.

<http://www.sirlin.net/articles/fail-safes-in-competitive-game-design-a-detailed-example.html>

“Guilty Gear Accent Core Character Frame Data” *Dustloop* 19 May 2012.

<http://dustloop.com/guides/ggac/data/ac/select.html>

“Learning to Fight” Thore Graepel, Ralf Herbrich, Julian Gold. 1 Mar. 2013.

<http://research.microsoft.com/pubs/65639/graehergol04.pdf>

“List of Fighting Game Archetypes” *Douk.* 25 May 2012.

http://forums.shoryuken.com/discussion/156294?threads/list-of-fighting-game-archetypes.156294/

“A Simple but Versatile IK Foot Rig” *A. Price* 8 Oct. 2012.
http://accad.osu.edu/~aprice/courses/694/IKfoot/ikfoot.html

“Skullgirls Character Movesets” *Autumn Games.* 22 May 2012.

http://skullgirls.com/guides/Skullgirls_CharacterGuide_EN_X360.pdf

“Spline IK Setup in Maya” *Joe Harkins.* 11 Oct. 2012.

http://www.animationartist.com/2003/08_aug/tutorials/softbody_tutorial.htm

“Data Analysis for Ghost AI Creation in Commercial Fighting Games” Worapoj Thunputtarakul Vishnu Kotrajaras. 1 Mar 2013.

<http://www.cp.eng.chula.ac.th/~vishnu/gameProg/papers/wpj_vishnu07.pdf>

“Varga Hair Tutorial” *Paul Tosca*. 18 Jan. 2013.

http://www.paultosca.com/varga_hair.html

“ZBrush to XNormal Pipeline” *Eat3D.* Web. 22 Sept. 2012.

http://eat3d.com/free_zbrush_xnormal_pipe

Appendix A: Vision Document

Super Special Awesome Untitled Fighter 2013

Game: A 2 ½ D fighting game with a twist. Instead of just choosing a fighter and duking it out with your opponent, players have the ability to change their fighter's attributes before the battle, potentially altering the entire course of the match!

Mini-Game concept: Streets of Rage style level that each player starts on the opposite end of, and reaching each other as the "boss" in the center. **SHORT**. 45 Seconds long MAX. Needs to be fast paces, lots of explosions, fun to look at, or else we're just wasting the time of the people who want to play a fighting game. *Minigame should be made LAST. Until then we'll have a placeholder screen for placing the attribute points.*

Possible Engine 1: XNA

Pros: XBLA Port/Controller Support super easy. Net play might even be easier with it.

Cons: You programmers will have to work super hard and have a bare-bones engine ready by the time we're back in the fall or at least have a playable engine by B term if we'll have any hope of finishing the project in a satisfactory way. We don't know how difficult it will be to get models/animations/etc into it.

Info: http://msdn.microsoft.com/en-us/library/bb200104.aspx

Possible Engine 2: Unity ←←←←←←←←←←

Pros: Engine base is done. We just need to fill in all the scripts, physics, features, etc.

Cons: Some things like to act up. But I'll try to work with importing animations and stuff over the summer so I figure out what makes it go crazy and what doesn't. Net play COULD be easy with this, plus web browser integration is always a plus. Has incoming PS3/360 support, iOS/Android Support. For 360 support we need Microsoft to sign off on it, but I could TRY to use the GDC's Microsoft contacts to get that permission.

Info: Net Play- http://unity3d.com/unity/engine/networking

Microsoft Support: http://unity3d.com/unity/publishing/xbox

Coding Language: C/C++/C# (I'll leave this up to Brian and Mike)

Good article (thanks Nick) about making balanced AND DIVERSE gameplay in fighters, which we will stick to like white on rice as the backbone of our basic game mechanics to minimize the frustration of wild character imbalance.

http://www.sirlin.net/articles/fail-safes-in-competitive-game-design-a-detailed-example.html

EVERYONE WATCH THIS VIDEO AND AUTOMATICALLY UNDERSTAND MORE ABOUT [GOOD] FIGHTING GAMES THAN YOU EVER DID BEFORE:

http://www.youtube.com/watch?v=2I83GsQGg6U&feature=iv&annotation_id=annotation_585242

HOLY SHOOT NUMBERS ARE AWESOME:
http://dustloop.com/guides/ggac/data/ac/select.html

SCOPE:

NEED TO HAVE:

- Basic Fighter Engine
    - HP Bars
    - Timer
    - "Rounds"
    - Stat Distribution Screen
    - Hit Box system.
    - Offense Meter

    - Defense Meter
    - Character Select Screen
        - Character Name/Portrait
        - Random
    - Stage Select Screen
    - On-Hit Particle effects.

- Two (2) Characters with full move sets, animations, sounds, etc.
    - States: Idle, Walk, Run/Dash, Jump, Knocked Down (ground / air)/ hitstun, Blocking
    - "Full moveset" size to be determined after determining controls. Estimated ~15 moves.
- 1 Stage for characters to do battle in.
- Multiplayer
    - Support for Fightsticks/360 Controllers

WOULD LIKE TO HAVE:

- Netplay
    - Leaderboards/rankings/Win-Loss records
- Human Vs. AI
- 3-5 Characters
- 2-4 Stages
- Warm-up/stat grabbing minigame

IF THERE IS TIME:

- More characters and stages?
- 360 Support?

Appendix B: Sample Work Timeline

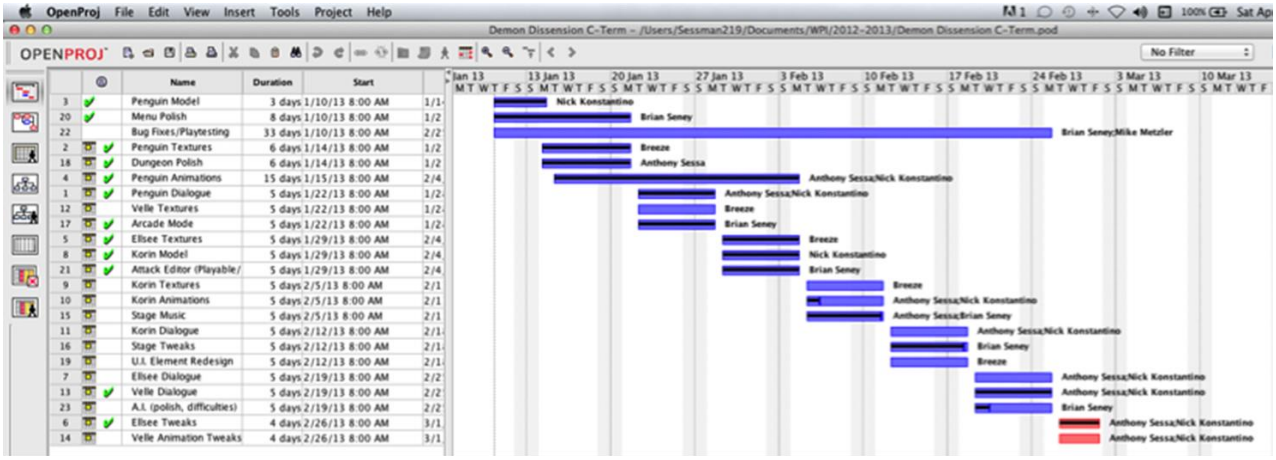

Fig 6.1: The timeline for C-Term

Appendix C: Blog

The developer's blog for Demon Dissension can be found online at www.handsomedevilstudio.tumblr.com