April 2011

# Obsidian: Development of a Multiplayer Game with Adaptive Enemy Intelligence

Douglas Paul Turcotte
*Worcester Polytechnic Institute*

Patrick Leland Knight
*Worcester Polytechnic Institute*

Richard Francis Pianka
*Worcester Polytechnic Institute*

Ryan Kenneth Chadwick
*Worcester Polytechnic Institute*

Sean Patrick Beck
*Worcester Polytechnic Institute*

Project Number: RL1-P210

# Obsidian

Development of a Multiplayer Game with Adaptive Enemy Intelligence.

Interactive Media & Game Development

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

Sean Beck, Richard Pianka, Ryan Chadwick,
Douglas Turcotte, Patrick Knight

Advised by:
Professor Robert W. Lindeman
Professor Britton R. Snyder

## Abstract

*Obsidian* is an Interactive Media & Game Development Major Qualifying Project striving to conceptualize, build, test, and release a prototype of a full game built in the *Unity3D* engine. The game challenges players to defeat a single enemy controlled by an adaptive AI system in a multiplayer game environment.

*Obsidian* is a concept piece for a larger game in which a team of agile players battle lumbering monsters in diverse environments. Players must use their speed, skills, and teamwork in addition to an assortment of spears, swords, and daggers to bring down the beast. They will also traverse an environment that encourages players to explore and find new possible ways to defeat the boss as they replay the game. Likewise, the boss will have his own set of goals and desires guided by his perceptions. Each game ends with the boss' defeat.

The project went through a fluid development process of four phases in which the team of three artists and two programmers designed, created, implemented, and tested their respective tasks. The team collaborated on nearly all aspects of the gameplay design. This process is documented through this report.

During the first phase of development, the team documented all facets of the game's technical design. Artists conceptualized their work through drawings, digital paintings, and 3D mock-ups. These designs and concepts were overseen by our advisors and underwent significant changes. The process involved in the creation of these designs is included in this document. At the end of this phase, we had all basic design elements completed and written into a design document.

The second phase involved the creation of all art and technical assets. During this phase, our team built game assets according to the design documents and concepts. At the end of this phase, all art assets were completed and organized for use in-game.

For the third phase, the team collaborated to combine all technical and art assets using the *Unity3D* engine into a playable form. Basic aspects of the gameplay such as combat and movement were also implemented during this stage. The final result of the consolidation of our assets was an alpha version of the game that was ready to be play tested.

The final stage consisted of testing and polishing the alpha version of Obsidian. Volunteers from the IMGD department played the game on campus and were asked to fill out a questionnaire related to finding any inferior aspects of the game. The information in these forms was organized and used in the final polishing process of our project.

# Table of Contents

# List of Figures

## List of Equations

# 1 Introduction

*Obsidian* is a multiplayer action/adventure game, set on a tropical island where three players cooperatively battle a single enemy. Players take the roll of tribal warriors seeking to kill a demon that has taken up residence on the island. They use a litany of weapons and skills in their attempt to defeat him, while the boss attempts to claim victory by fighting the players off.

The project team formed in D-term 2010, and began design and concept work over the summer. Due to having a double major on the team and to allow for extra time, the project was spread out over four terms. Ryan Chadwick and Patrick Knight formed the art team, Doug Turcotte and Rick Pianka did all the programming, and Sean Beck was a technical artist.

Artistically we strove for a realistic look, relying on a combination of hand-painted textures and photographic textures. Our work covered everything from hand drawn concepts, to models, to textures and shaders. The team endeavored to create art assets that could potentially be found in any modern game produced by an actual game studio.

The technical work consisted of two sides, the client and the server, due to the multiplayer nature of the game. The client work covered game development in Unity3D. The server work included implementation of a webserver and the enemy's artificial intelligence.

This document discusses the design and development process of Obsidian. The first three sections cover the design, art, and technology behind our game respectively. Finally, information on play testing is included along with appendices containing asset lists and design elements removed from the game.

# 2 Gameplay

In *Obsidian,* three players must work together to defeat the enemy that has taken over the environment. The enemy is controlled entirely by an artificial intelligence that adapts to the player's actions. When the game starts, players are dropped into a separate area from where they fight the enemy. Players drop down into the arena area, and must deplete the enemy's health. The end of the game will always result in a boss death, as the players cannot lose.

## 2.1 Player

Three players are allowed into each game, though the game is still playable with one or two. There are three separate types of player characters – melee, ranged, and utility. Melee players have the greatest amount of health, and wield a sword. Ranged players have the least amount of health, but wield a spear that can be thrown from great distances. Finally, utility characters are a balance of the two, and use a dagger. The controls shown in Table 1 follow the normal gaming conventions.

**Table 1 – Obsidian Control Scheme**

| Keyboard Key | Action |
|---|---|
| W | Run Forward |
| A | Strafe Left |
| S | Back Pedal |
| D | Strafe Right |
| Space | Roll Forward |
| Left Click/Ctrl | Attack |

Players have two resources: health and stamina. On each hit from the boss, the players will lose a portion of their health. To allow for longer games, players will regain their health slowly. Stamina is depleted as the player uses abilities as well as when they roll. Stamina recovers quickly over time.

## 2.2 Skills and Weapons

Players are given skill points at the beginning of a game and must place them into skill boxes. These skill boxes will represent passive bonuses to the player, bonuses that give the player small boosts to things like attack speed, health regeneration, amount of health, and amount of damage.

As players have access to multiple types of weapons, they will also be able to choose which elemental weapon they prefer. There are three types – fire, ice, and poison. Fire weapons cause more damage, ice weapons slow the boss down, and the poison weapons cause damage over time.

## 2.3 Enemy

The enemy ("boss") takes the form of a large bi-pedal creature. He is a great deal larger than the players, and causes large amounts of damage with each hit. Driven by his AI, the boss reacts to each interaction with the player based on past experiences. His goal is to kill each player.

Similar to the players, the boss has two resources from which he draws. He has a large pool of health, which is always shown at the bottom of the player's screen. Every time the boss attacks, he also uses some of his stamina. Both stamina and health regenerate over time, with stamina regenerating more quickly.

## 2.4 Level Design

The level layout has an effect on how each battle is played. There are two main areas for fighting the boss, both surrounded by heights for the players to scale. The heights give the player a haven from which they can plan their next attack or launch spears to draw the boss in.

Due to the bosses size, he can attack players at any height using a swipes and punches. However, at heights, the players are protected from the enemy's feet.

## 2.5 Winning and Losing

When a player dies, he is sent back to the area where he first spawned with half of his health removed. Each time a player dies, the boss regains a portion of his health. Thus, players must be careful not to die or else the battle can last for an extended period of time.

# 3 Art Development

The development of art assets began early on in the process. Starting in A-term, we immediately began concepting and designing the pieces necessary to create the game. This section will go over that process in detail, covering conceptualization, design, and how each model was brought into the game. It focuses on the boss, player, and environmental models.

Separate from the main models are the last three sections which go over animations, the user interface, and sound effects.

Autodesk's Maya, 3ds Max Design, Pixologic's ZBrush, and Adobe's Photoshop all played a big role in the creation of our assets. The following sections will go into how each tool was used.

## 3.1 Design

This section covers the design of our main art assets. Each asset follows important design choices made early in the process. Included is the design process for the boss, player, and environmental assets.

### 3.1.1 Enemy

The main antagonist in Obsidian, the boss, serves as the main focal point and goal in the game. Because of the importance and focus of the Boss character, careful consideration was put into his creation. The first two weeks of the project were spent conceptualizing his character which involved two main investigative tasks. The first task was to research previous creations with similarities in style, structure and purpose. We selected a range of games to pull from including AAA titles like *Shadow of the Colossus*, *Darksiders*, and the *God of War* trilogy. We decided early on to use the most modern technology and techniques in order to mimic the high production value of these professional titles produced by much larger teams on a much larger budget. In order to achieve this feat, we scoped our project down to include only one boss creature so that we could focus on making him as detailed as possible in both look and capability.

### 3.1.2 Players

After the overall theme and artistic direction of the game was decided, it was necessary to specifically tackle the challenges of designing and creating the various characters which would be actors in the game. For the avatars which would represent the players in the game world, we

decided to create three distinct classes which would work cooperatively in the game world. By design, these characters all use the same base model so that each character would feel like the others and so animations could be reused for each character class. Originally, wholly different character models were considered for each player class, but considerations for the time necessary to concept, sculpt, texture, and animate three completely original models made it necessary to rely on a single base mesh. However, while the base meshes of the models are identical, the accessories (armor, clothing, bags, and headwear) of each character are different so as to create unique silhouettes. These unique silhouettes are important for players to distinguish a particular character from across the game map during play.

### 3.1.3 Environment

For the environment, the team decided on a tropical island in the midst of a volcanic eruption. We also wanted two areas, one lush and green and the other dead and dark, to create juxtaposition. Players start in the green area, and then must hop down into the volcanic area which acts as an arena for the player to battle the boss. The main inspiration for the environment was tropical islands in the Pacific Ocean, as the team liked the ubiquitous black rock.

Due to the nature of the enemy, a demonic creature, the forest section has some fantastical elements. There are giant green and red mushrooms strewn across the environment, floating orbs of light emitted by tiny bugs, and blue flame torches.

The environment also needed assets to make for a more compelling experience, so the enemy was given a throne to illustrate that he is ruler of the island. The sharp angles and dark green obsidian were intended to complement the red, harsh environment.

## 3.2 Concept Art

The section covers any concept art used during the process of creating assets. These can be hand-drawn examples the team created or reference images from other games and artists.

### 3.2.1 Enemy

In the early concept phase, we focused on creating an imposing figure that would fit stylistically in a dark, fantasy environment. We pulled ideas from games sporting demons, dragons, and all manner of large monstrous creatures in order to create an imposing and dangerous-looking enemy. Early concepts ranged from giant humanoids decked out in metal plate armor wielding

massive cleavers, to tree-like beings woven in vines, to muscle-bound demons covered in spikey protrusions. The last two can be seen in Figure 1. Ultimately we decided on a more demonic, powerful monster encrusted with volcanic obsidian spikes and armor. The Boss also has several face-like features on his torso and arms which would allow him to deploy magical breath-like attacks from several key points on his body. Giving the boss multiple faces allowed us to present several obvious weak points on his body as focal points for the player's attacks as well as opening up a range of alternate attacks to feed into his dynamic combat decisions.

**Figure 1: Early Enemy Concepts**

### 3.2.2 Players

Using the tribal island theme as a launching point, character concepts were created based on various reference images including those sourced from film, television, other games, and life photographs pertaining to tribal culture. For the player characters, very few concept drawings were created and most featured ideas for smaller accessories rather than the overall look of the character. The player character's design was mainly worked out in 3D in ZBrush to ensure a representative early prototype. Early models attempted a realistic style which later did not seem to fit with other generated assets, so a new design was created with a more stylized look.

### 3.2.3 Environment

The first step for conceptualizing the environment was creating overhead images that the environment could be modeled after. Figure 2 represents the earliest version of the concept done for the level.

**Figure 2: Early mockup of level**

The second iteration is displayed in Figure 3, which is the concept the level was built from. This concept image was intended to show the actual layout of the level such as where heights, lava, obstacles, and the arena itself would be. The lighter, slightly green area is the forest section where players spawn. The grey area to its right is the arena where the players fight the enemy, and it is split down the middle by the red lava. Finally, the area below the large pool of lava with an "H" shaped object is the throne. The throne was later moved into the middle of the arena to make it more visible to the player.

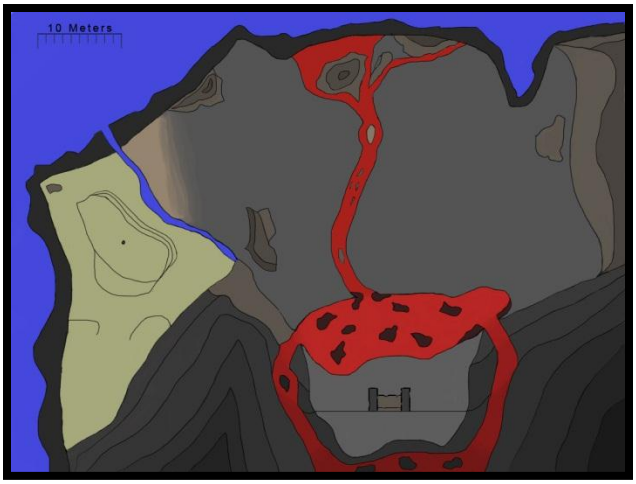


**Figure 3: Early design of layout.**

Figure 3 led to Figure 4, which is a screenshot of the very first iteration of the level built in the Unity3D editor. At this point, placeholders were used for all assets.

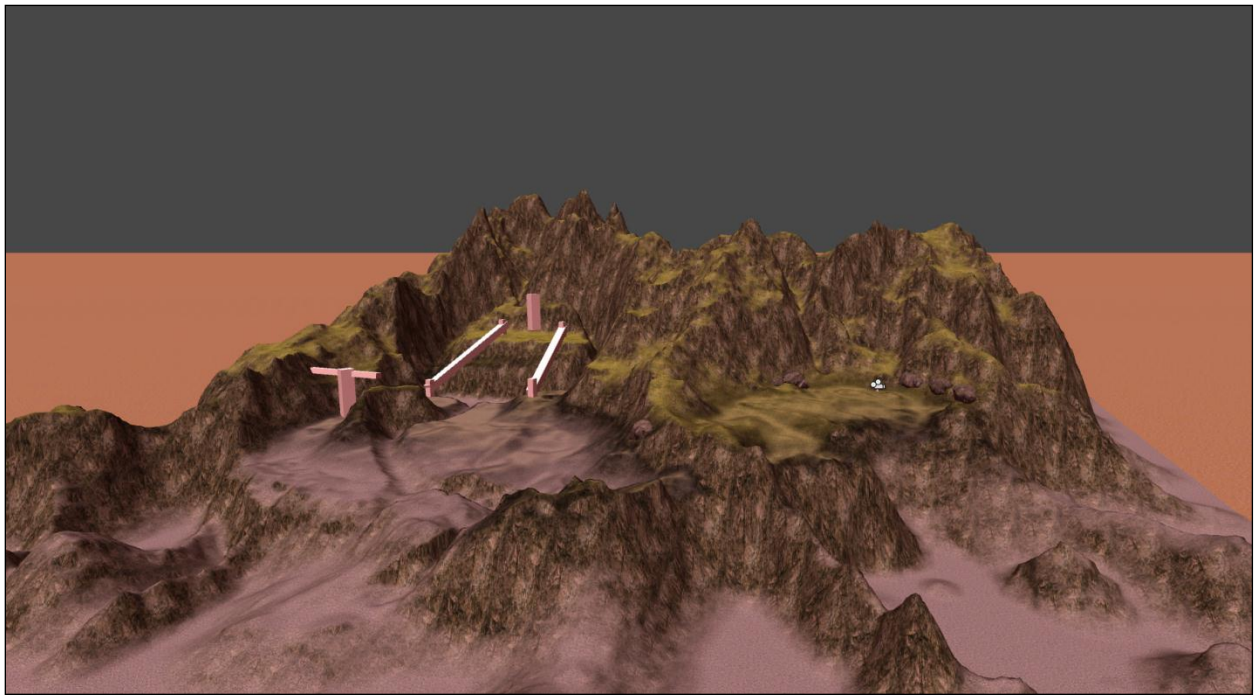

**Figure 4: First iteration of level.**

Environmental assets like rocks, trees, and foliage were not built from concepts to save time. The throne did have a basic sketch drawn, shown in Figure 5.

**Figure 5: Rough throne concept.**

## 3.3 Pipeline

The pipeline section covers how each asset was created and brought into the game.

### 3.3.1 Characters

After the concept was finalized, the boss was created in 3D. The base mesh was created using a 3D sketching tool called Zspheres in the digital sculpting program, ZBrush. ZBrush allowed us to sculpt the mesh using a very high resolution for detailing the boss model in order to give him lifelike features. Figure 6 shows his progression from a very rough model to the finished product. The initial model had very thing legs, as we wanted to have most of his size at the top to make him more intimidating in-game.

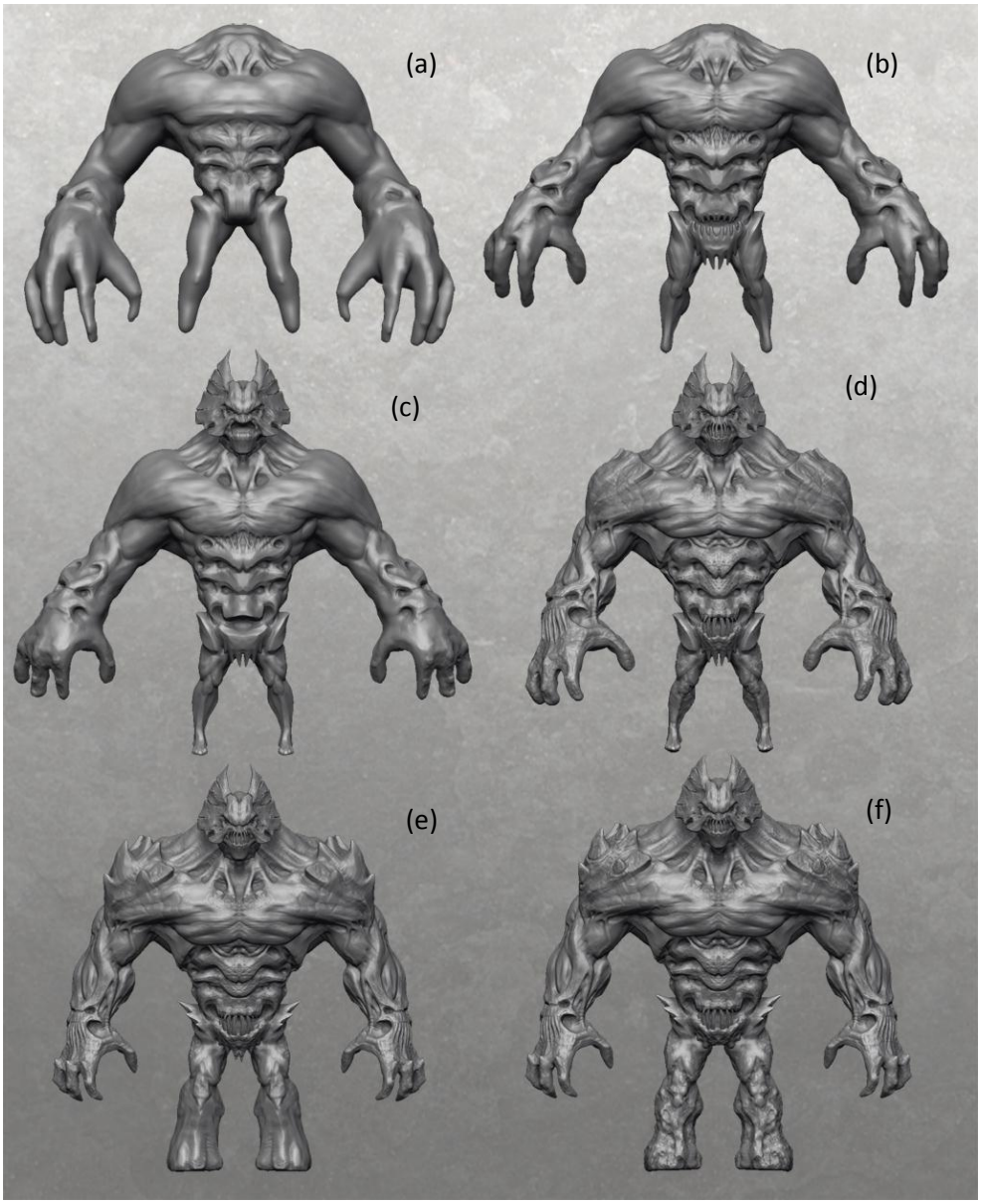


**Figure 6: Progression of enemy model. A through D are early versions with thin legs. F is the final version used in-game.**

After the sculpt was complete, the texture painting process began in ZBrush. An overall greenish hue, shown in Figure 7, was chosen for the boss because the environment had a more red tone. This contrast in complementary colors means that the boss is easily recognizable and stands out from the environment even from a distance.

**Figure 7: Enemy Texture Close up.**

The result of this process is the final product shown in Figure 8.


**Figure 8: Final boss model.**

Obsidian's player characters were also created using ZBrush. Through a series of sculpts and procedures which reform the model's mesh, or "retopologizing," we were able to construct the base mesh in pieces. This method allowed unequal distribution of the model's polygons, with more being devoted to areas requiring more resolution such as the face, and fewer being devoted to low resolution areas such as large surfaces or covered skin.

After creating the base mesh, work began to generate various assorted assets which would accessorize the characters to denote them as belonging to the melee, ranged, or utilities classes.

Items such as heavy leather armor and animal pelts for the melee class, a spear quiver and bandolier for the ranged class, and a hood and utility belt for the utilities class were all created as subtools by creating new polygon groups ("topology") and extruding them from the base mesh, and can be seen in Figure 9. This method allowed for rapid generation of assets which would fit the base mesh exactly and require little retopologizing to eventually produce low-resolution, mapped assets.

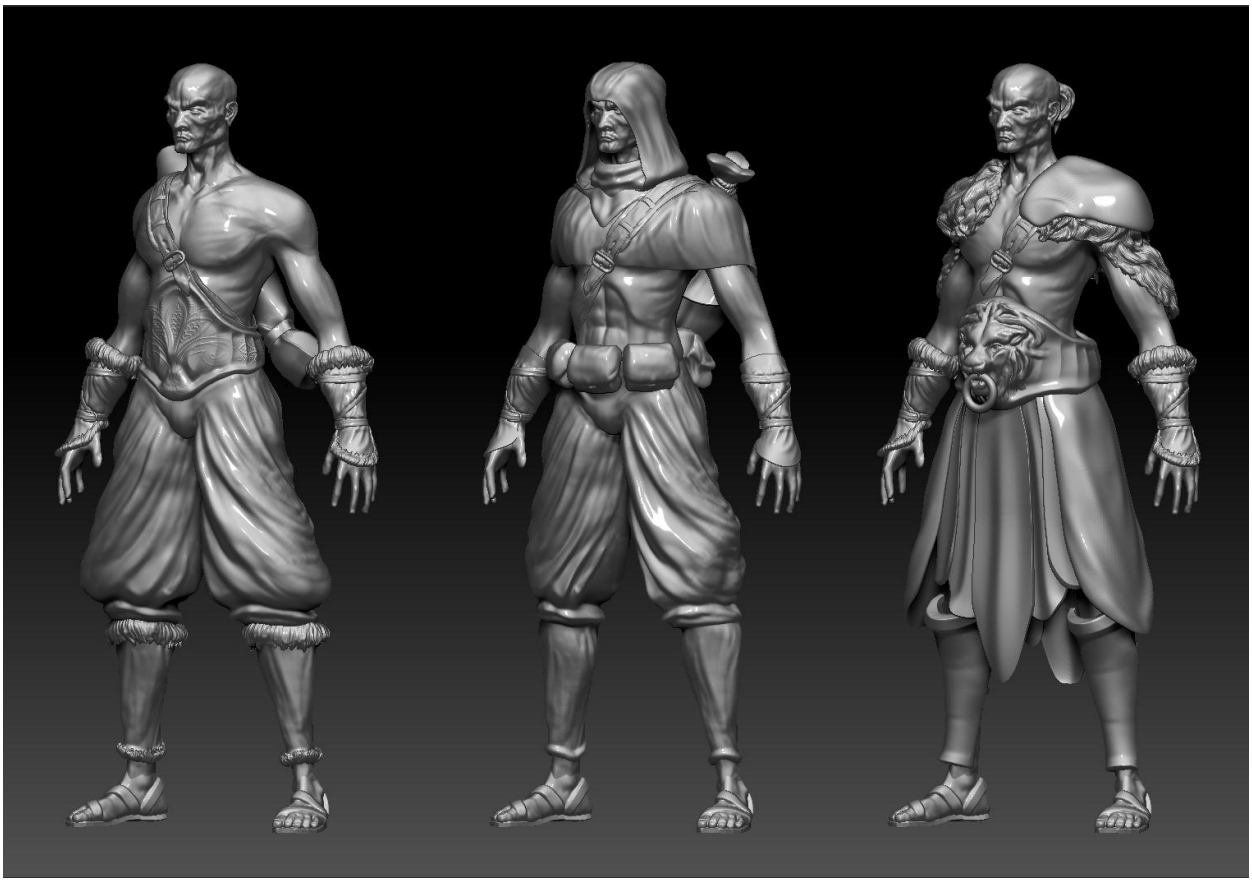

**Figure 9: High resolution player models untextured.**

Texturing of the player character models was handled almost exclusively in Zbrush using the "Spotlight" and "Polypaint" features which allow for a modeler to paint textures directly onto the 3D model. For things like fabrics, masking was used in coordination with polypaint to create striking contrast between folds. For objects like leather and metal which have very characteristic looks which may be difficult to render with paint, spotlight was used to project a texture directly onto the mesh. The result can be seen in Figure 10.

**Figure 10:Textured player models**

Once textures were created, normal maps and texture maps were generated for each individual piece of each player character. These maps were generated using Zbrush's UV Master plug-in in coordination with the proprietary normal map and texture map generation functions.

### 3.3.2 Environment

The environment can be broken into two main sections, objects built in the Unity3D editor and objects built with other tools such as Maya and ZBrush.

The terrain, including its textures, was the first aspect of the environment completed. Environmental assets were created once the basics of the level had been built. A full overhead image of the finished environment from within the Unity editor can be seen in Figure 11.

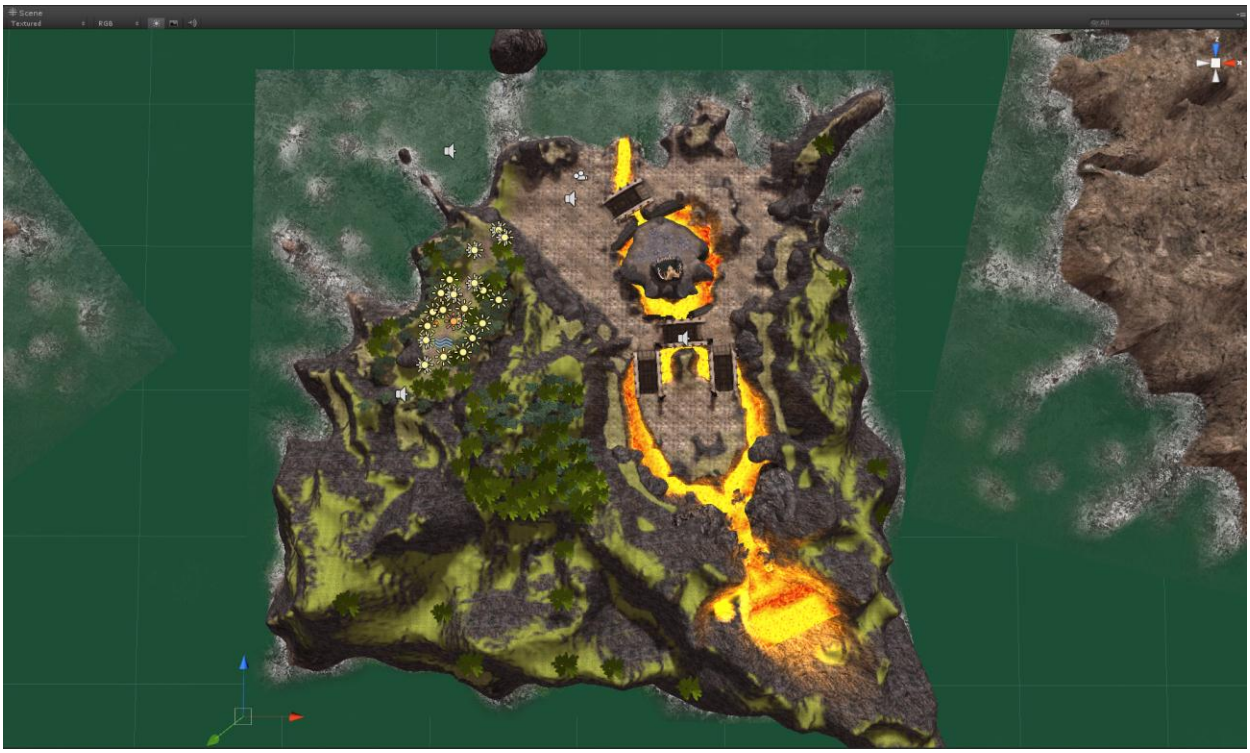

**Figure 11:Overhead view of finished environment.**

The following sections will cover the creation of the environment and all accompanying assets in detail.

### 3.3.2.1 Unity3D Assets

The terrain was built completely from within the Unity editor shown in Figure 12, using the terrain tools that come packaged with the editor. Supplementing the tools that came with Unity is the Terrain Toolkit. The Terrain Toolkit is a plug-in for the Unity Editor which allows the user to procedurally generate terrains, wind and water erosion effects, and apply textures across the terrain.

Our terrain was first blocked in with the basic tools, then the erosion and texture tools included with the Terrain Toolkit were used to add detail. For foliage, Unity utilizes a detail brush that allows painting of bushes, grass, trees, and any other small details to the terrain. Details painted with the brush are streamlined to give the best performance and also gives control over aspects of the foliage's color and size to give visual diversity. The foliage can be seen in Figure 13.

So as to better explain how the game was put together, this paragraph will go into the details of the Unity Editor. The bottom left pane in Figure 12 is called the hierarchy, and it displays all

game objects placed in the scene. The pane to the right of that is the project pane, and it shows all assets that are in the assets folder for the project. This includes all textures, models, animations, scripts, shaders, etc. The far right pane is the inspector, which is where all tools for the game are displayed for the currently selected object in the scene or hierarchy.

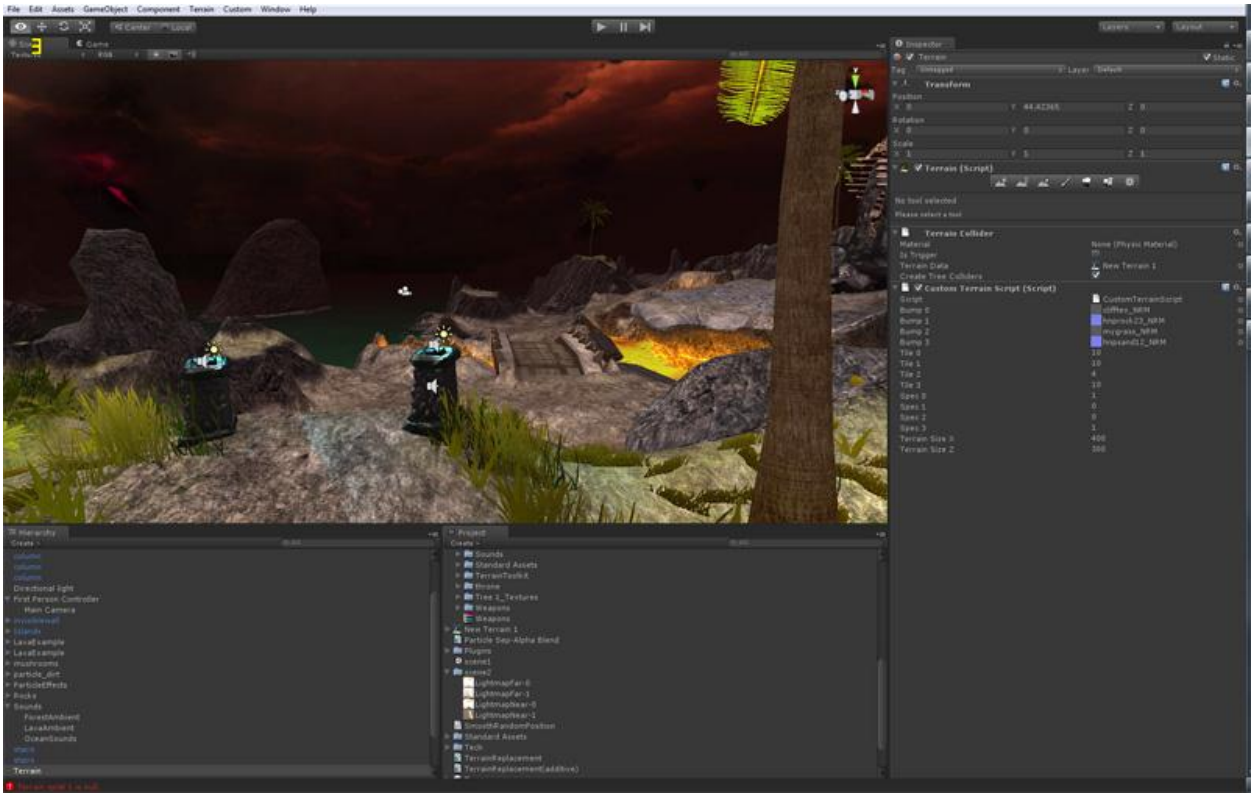

**Figure 12: Unity Editor**

Trees, shown in Figure 13, were created in the built-in tree creator tool which does a great deal of trivial work involved with trees. The shape and size of the trees, the location of branches and leaves, and what textures the trees use were user defined from within the tree creator interface. The end result is a tree that has levels of detail and billboards premade for performance gains and

can be simply placed using the detail brush mentioned earlier.

**Figure 13: Forest section foliage.**

### 3.3.2.2 Other Assets

The throne shown in Figure 14, throne base, and mushroom were created using a similar process to the player and enemy. Base meshes were created in Autodesk Maya and were exported to .OBJ files. Those OBJs were then imported into ZBrush to create a high poly version of the model and to paint normal and diffuse maps. Normally, a modeler would have to retopologize the models to create clean edge flows which ensure smooth animations. Since our environmental assets have no animations and thus do not require clean edge flows, the ZBrush plug-in Decimation Master was used to produce all the low poly versions of the high resolution models.

**Figure 14: Throne render from ZBrush.**

All plants, the bridge, and column were made solely in Maya and textured in Photoshop. Normal maps were generated in CrazyBump.

## 3.4 Animation

Animations play a very important role in our game, as smooth, creative animations are necessary to make gameplay seem believable. This section delves into why we chose our animations and how they were created.

### 3.4.1 Enemy

Once the enemy sculpt and texture were complete, he was exported into Maya, another 3D modeling and animation tool, in order to prepare him for animation. The rig for the Boss was a fairly simplistic bi-pedal skeleton outfitted with switchable IK and FK handles for the limbs. Because of the genetic artificial intelligence system that controls the Boss, he can theoretically support an infinite pool of actions to choose from in combat. Due to time and resource

constraints, we chose to focus primarily on the most used possibilities for actions. These animations include a walk cycle, an idle, and a collection of different battle actions, such as punches and stomps. By focusing the number of actions into a specific small subset of possible actions, we were able to concentrate on creating high-quality animations while still being able to highlight key features of the game.

### 3.4.2 Players

Animation of the player characters was done exclusively in Autodesk Maya. Animations were created by hand, posing the characters by manipulating points of articulation, creating key frames of extreme gestures, and letting the program interpolate in-between frames. Animations were created separately based on urgency beginning with basic motions such as running and attacking. Other motions included rolling, attacks for each weapon type, and side-stepping. When animations were complete, they were prepared by conversion to the appropriate file type while "baking" the animations to the model so the game engine would not need to calculate motion during runtime.

## 3.5 Sound Effects

As we didn't have a dedicated member to produce sound, we relied on purchasing sound effects from the online vendor AudioJungle.net. These include ambient sound effects for the lava, forest, and water, plus sound effects for actions in game. These range from grunts from the player to combat sounds.

Ambient sounds were attached to objects in-game through audio sources, an object built in to Unity3D. From there, volume levels and range of hearing were changed accordingly for each sound.

## 3.6 User Interface

Because of the multiplayer nature of the game, Obsidian requires online client in order for players to log onto the server, create new games, and connect to previously made games. Visually, the client user interface, or UI, seen in Figure 17, Figure 16, and Figure 17 was created to mirror the dark fantasy and volcanic themes in the game. The Obsidian client was designed to be as simple as possible in order to minimize time spent out of game. It provides a chat room seen on the left in Figure 16 and Figure 17 to discuss the game, a list of current games on the right in Figure 16 and a menu for creating or joining a game. Once the player has joined a game, they are

placed in a private chat room to discuss strategy. They also have the option to customize their character, choosing from three classes, warrior, hunter and trapper, with 15 unique skills for each class.


**Figure 15: Log-in Screen Interface**


**Figure 16: Server Lobby Interface**


**Figure 17: Game Lobby Interface**

The in game UI in Figure 18 was kept fairly minimalistic in order to maximize immersion. It shows the player's health and energy bars as well as two possible active skills dependent on character class choice. The other main feature on the in game UI is the enemy's health bar, which is always present on screen.

**Figure 18: In-game UI**

# 4 Technology

The technical decisions in this project are generally focused on Microsoft and Windows-based systems. The frameworks and software chosen include .NET, Unity3D, Photon, Simple Machines Forum and MySQL. A very high-level overview of the communication flow for the whole system can be seen in Figure 19.

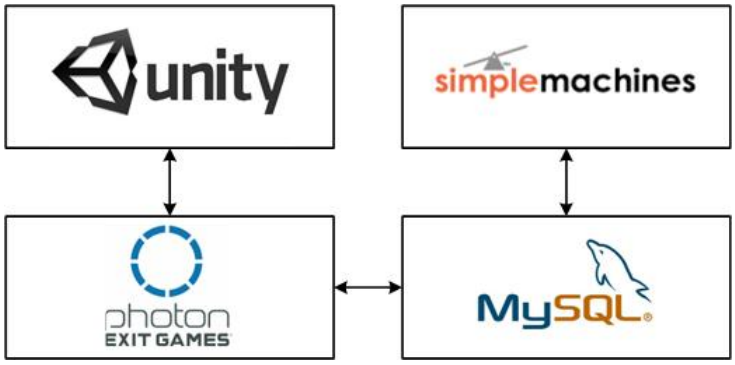


**Figure 19: Communication Overview**

The client side is implemented using the Unity3D engine using the C# language for scripting. The client's networking is implemented in a .NET library. This library uses the Photon framework for low-level networking and messaging. Additionally, the server is implemented in another .NET library that also uses the Photon framework for its networking and messaging. The server library is loaded by the Photon Control application, which runs on a Windows Server 2008 virtual machine.

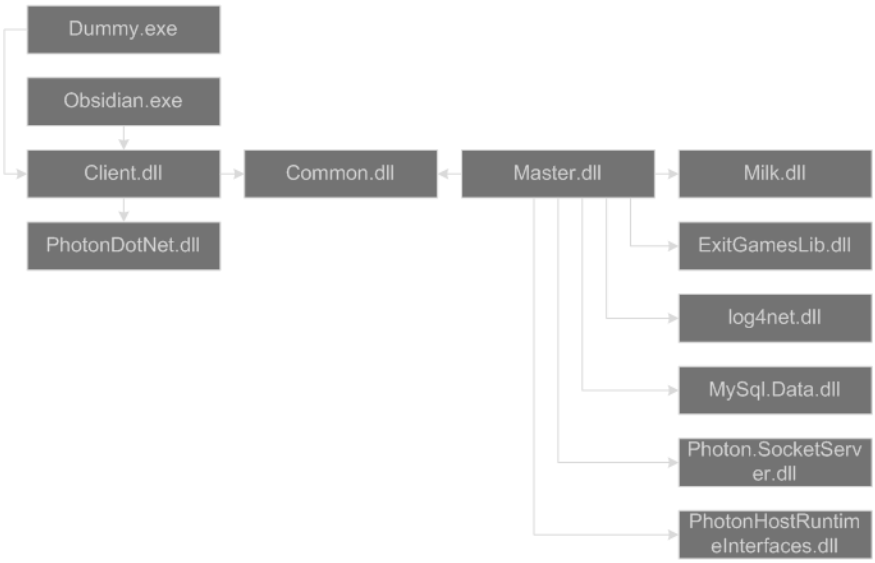


**Figure 20: Dependency Chart**

The artificial intelligence is all implemented in yet another .NET library, which is referenced by the server library. The full dependency chart is visible in Figure 20. The server library also integrates with a Simple Machines Forum instance on a LAMP stack. The MySQL instance houses all of the forum data as well as the persistent game data. Together, all of these components form the technical portion of this project.

## 4.1 Game Client

The client is composed of eight main modules. These modules are Input, Interface, Network, Character, Sound, Physics, Engine, and Scene. Data flow between these modules is also displayed in Figure 21.

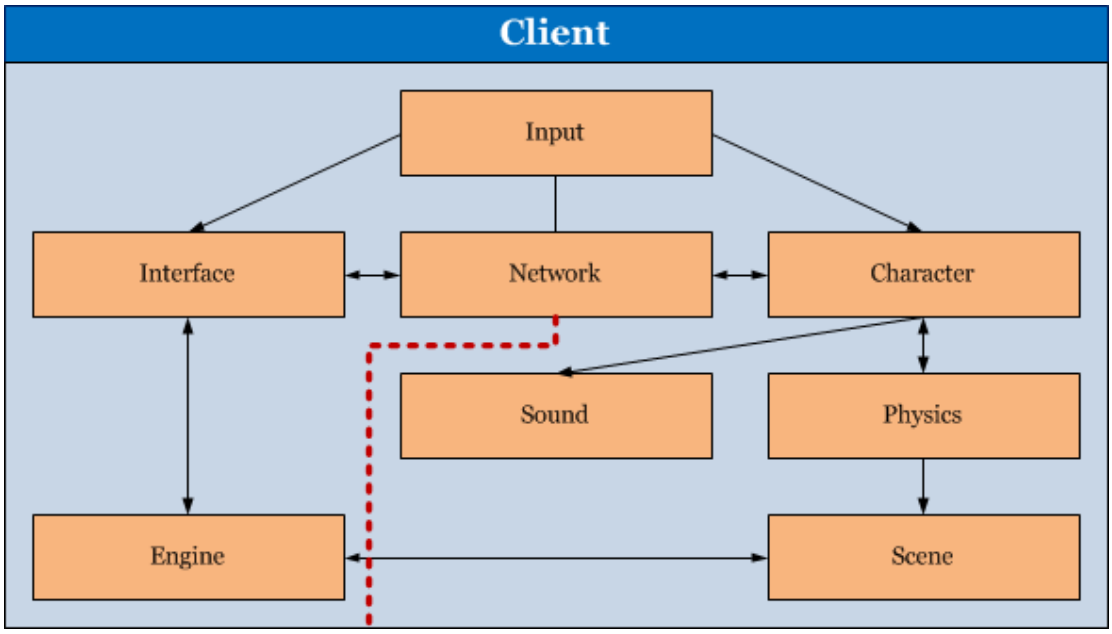


**Figure 21: Client Overview**

The Input module handles all user input from the mouse and the keyboard. User input is interpreted in reference to the user's keybinds, and appropriate game messages are dispatched. The Input module sends input event data to the Interface, Network, and Character modules.

The Interface module is all of the GUIs (HUD, various menus) in the game. The Interface module sends UI interaction event data to the Network and Engine modules. The Interface module receives input event data from the Input module, GUI state data from the Engine module, and display information from the Network module.

The Network module is the interface the client uses to communicate with the game server. It handles incoming and outgoing messages. The Network module sends display data to the Interface module and character state data to the Character module. The Network module receives UI interaction data from the Interface module, character state data from the Character module, and input event data from the Input module.

The Character module handles the visual display of character position and actions, the combat relationship between characters, and resource management. Character includes the boss and the players. The Character module sends character state data to the Network module, character event data to the Sound module, and character state data to the Physics module. The Character module receives input event data from the Input module, character state data from the Network module, and physics interaction data from the Physics module.

The Sound module handles all sound management and output. The Sound module receives character event data from the Character module.

The Physics module handles all physics interactions. The Physics module sends physics interaction data to the Scene and Character modules. The Physics module receives character state data from the Character module.

The Scene module is the internal representation of the game world. The Scene module receives data from the Engine and Physics modules.

## 4.2 User Input

The structure for the Input module is shown in Figure 22. Input from the mouse, keyboard, and controller are read by the input handler. The input handler then forwards the appropriate information to the UI, the network interface, and the character interface. Core input devices are the mouse and the keyboard. The “WASD” standard for movement is used for the keyboard, along with <Space> for roll and <Left Click> for attack. Moving the mouse rotates the chase camera and turns the character. The input handler is composed of two parts: the Unity3D input interface and an internally simulated Input State. Unity3D has an input interface which reads from hardware input devices and provides this data for reading by scripts. The data read from

Unity3D's input interface is used to update the internal Input State. The Input State is used to track input changes, and simulate the Input States of other players in the game.

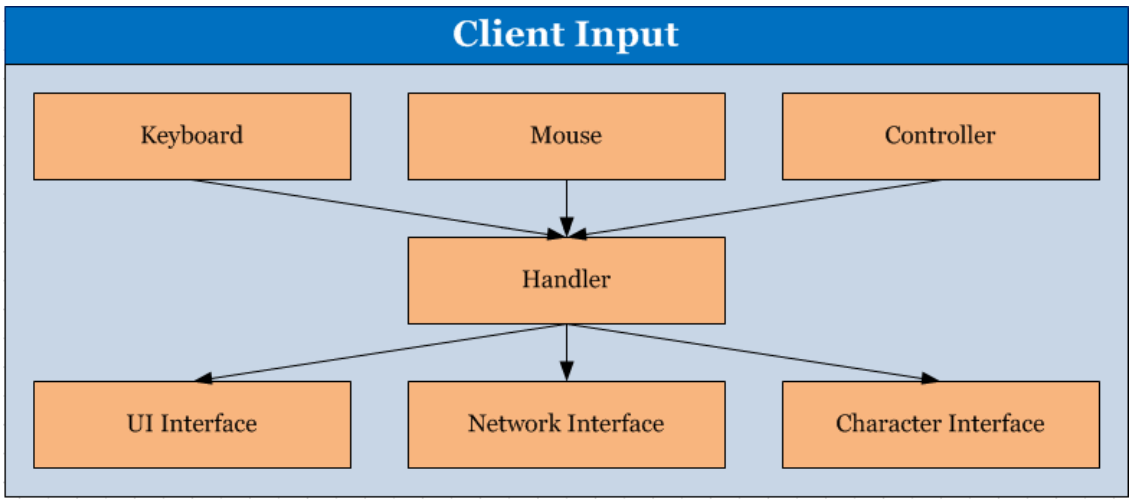


**Figure 22: Client Input Overview**

### 4.2.1 Input → Interface

The Input module sends data to the Interface module as shown in Figure 23. Mouse interactions from Unity3D's input interpreter communicate with the interface's menus and buttons. The Input State is not part of this interaction; input data is sent directly from the input interpreter to the UI.

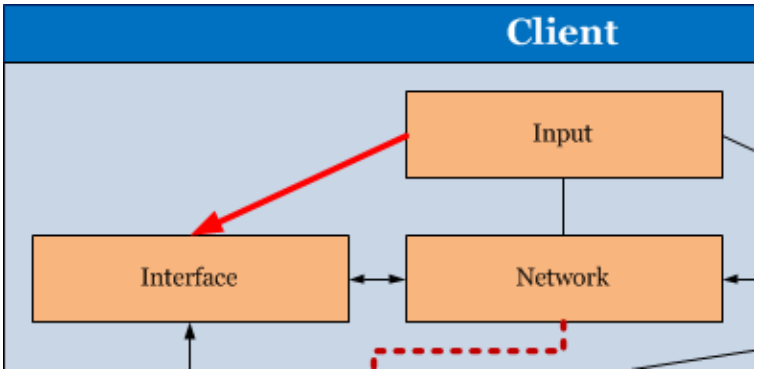


**Figure 23: Input to Interface communications**

### 4.2.2 Input → Network

The user keyboard and mouse interactions from the input interpreter that affect the player's character position, orientation, and actions are sent over the network to be relayed to other clients to allow all clients to display the character. This is shown in Figure 24. Data that is read from Unity3D's input interface is sent to the Input State. If the data is different from the current values

in the Input State, the updated data is sent over the network. The benefits of using the Input State structure are discussed in the next section.

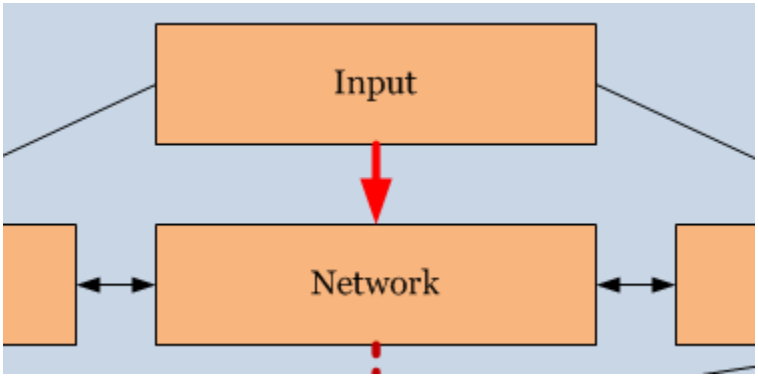


**Figure 24: Input to Network communications**

### 4.2.3 Input → Character

The user keyboard and mouse interactions that affect the player's character position, orientation, and actions are used to update the Input State, which is sent to the character handler as shown in Figure 25. This information is used to update the display of the player's character.

There are two key purposes for the Input State: first, it makes displaying other players in the game simple for each game client. The code for displaying the player's character is very similar to the code that displays other characters, since each reads from an Input State to determine actions that need to be displayed. Managing Input States in this way also results in a simple dead reckoning system. Each game client uses an Input State to extrapolate how to display the other players in the game. Until new commands are received for a player, it is assumed that their state has not changed. Thus, if the Input State indicates that the player is moving forward, it is assumed that the player continues to move forward until an updated value is received.

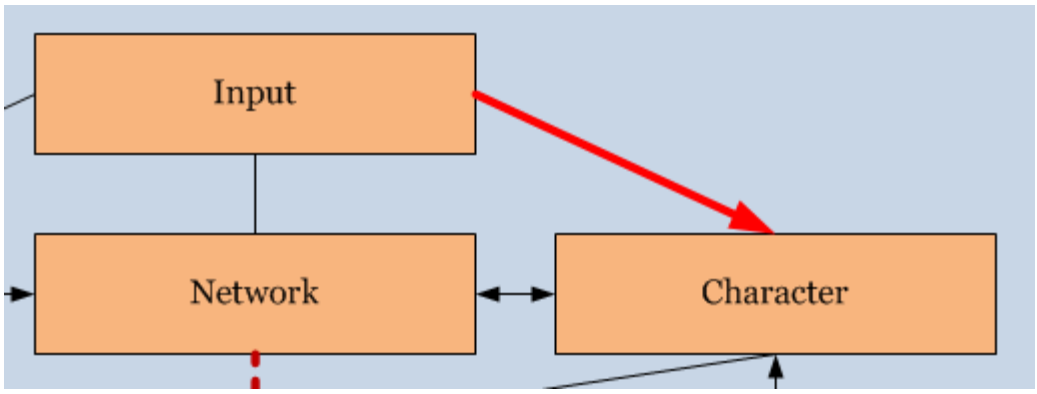


**Figure 25: Input to Character communications**

## 4.3 Interface

The Interface module consists of the game's UI screens. These are the login screen, the main lobby, game lobby, and HUD. This can be seen in Figure 26.

The login screen has fields for account credentials, as well as an option for connecting via the WPI network. A connection cannot be made with an external server from the WPI network. The solution for this is to host the server on the WPI network, either directly or through a VPN. The client waits to connect to the server until login credentials have been entered, and this connection option has been chosen.

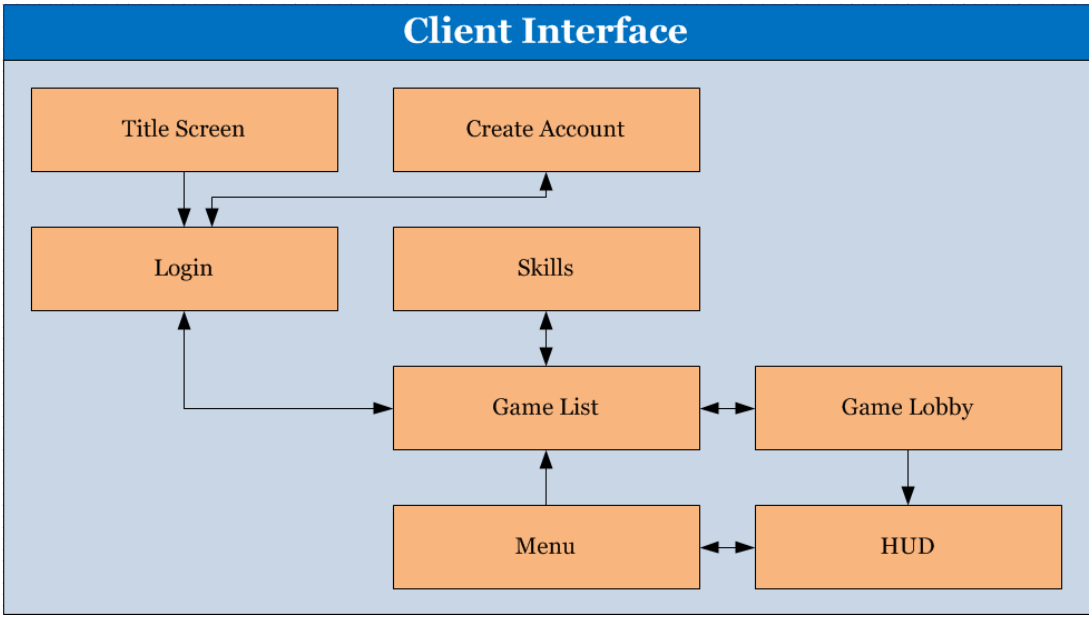


**Figure 26: Client Interface overview**

### 4.3.1 Interface → Network

The interface is responsible for telling the network interface to send various commands to the server. These commands include joining, leaving, and creating games, class selection, login, and game list queries. This is illustrated by Figure 27.

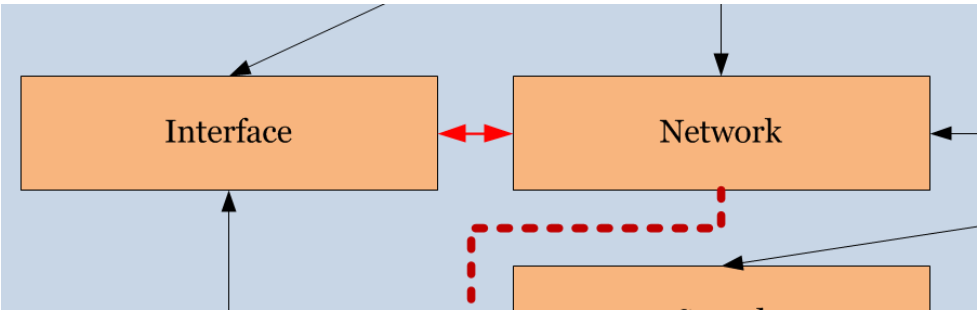


**Figure 27: Interface to Network communications**

## 4.4 Sound

The Sound module is composed of all audio sources in the scene, as in Figure 28. Many audio sources are used, to allow for 3D sound positioning and simultaneous sounds. The sound effects that are used are organized in an SFX class, and read by the appropriate audio sources.

There are environment audio sources for ambient sounds, as well as audio sources for the player and the boss. Most combat sounds are played by audio sources connected to the boss or the player. However, some combat sounds, such as when a weapon strikes the boss, are played by audio sources connected to particle emitters. When a character takes an action that leads to a combat sound, it is likely that the character has its own sound to play. Any given audio source can only play one sound at a time, so to play multiple simultaneous sounds, multiple audio sources are needed. Since a particle effect accompanies each of these combat sounds, it is reasonable to connect an audio source to it. The particle emitter, and therefore the audio source, is destroyed upon completion of its effect.

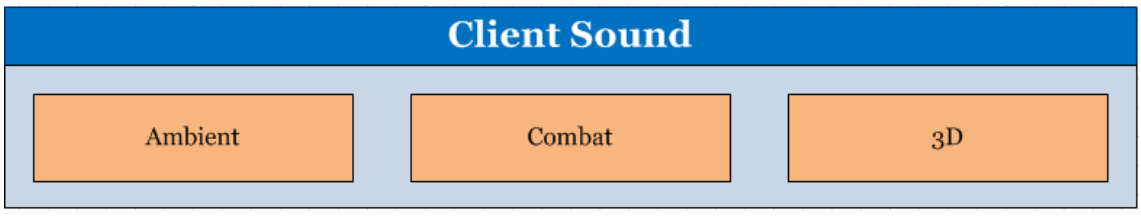


**Figure 28: Client Sound module overview**

## 4.5 Scene

The Scene module is composed of dynamic objects in the scene that are not handled by the Character module, visible in Figure 29. This consists of the camera, projectiles, lava hazards, and visual effects.

The Camera is a chase camera, following the player's movements. The camera aspires to stay behind the player, aiming at a point above the player's shoulder. The distance and vertical position of the camera are adjusted by mouse input. The camera always stays in front of an object that would otherwise obscure view of the player.

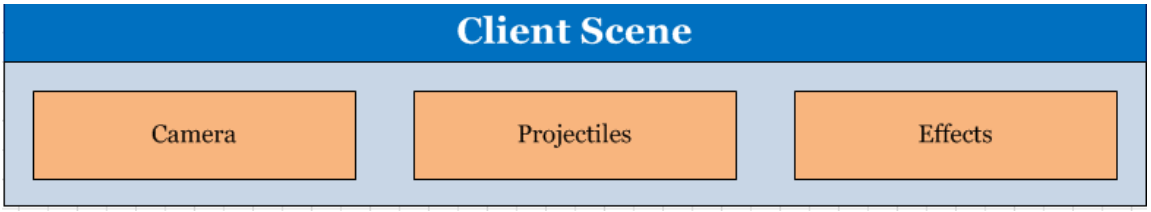


**Figure 29: Client Scene overview**

Projectiles in the scene are those created by players wielding the spear. Spears thrown calculate and follow their own trajectory, until striking a relevant collider, such as the boss or the environment. Each spear has a life of eight seconds after a collision. Upon collision, the spear is parented to the transform it collided with, so it remains in the same position relative to the parent transform. This results in a stationary spear when colliding with the environment, or remaining connected to a part of the boss when colliding with it.

Dynamic visual effects tracked by the Scene are particle emitters created when a player strikes the boss. These particles are destroyed upon completion of their effect.

## 4.6 Network

Network features can be divided into two categories: incoming from server, and outgoing to server.

### 4.6.1 Incoming messages

Incoming messages that the client network interface receives from the server are account verification, chat messages, the game list, the load game command, player data, and boss data. Player data includes Input State changes from other clients as well as their combat interactions. Boss data consists of path navigation, combat interactions, and attack commands.

An account verification message is received after a login attempt with valid credentials. Chat messages are received for the lobby that the player is currently in. The game list is returned from a game list query, if the player is seeking to join a game. The load game command is sent after the host of a game requests that the game begin.

Boss path navigation and attack data is received at the discretion of the server. Boss combat interactions are messages that are relayed from other clients.

Player Input State changes and combat interactions are relayed from other clients.

### 4.6.2 Outgoing messages

Outgoing messages that the client network interface sends to the server are Input State updates, combat interactions, login credentials, game list queries, join game requests, host game requests, start game requests, and boss data. The network modules can be seen in Figure 30.

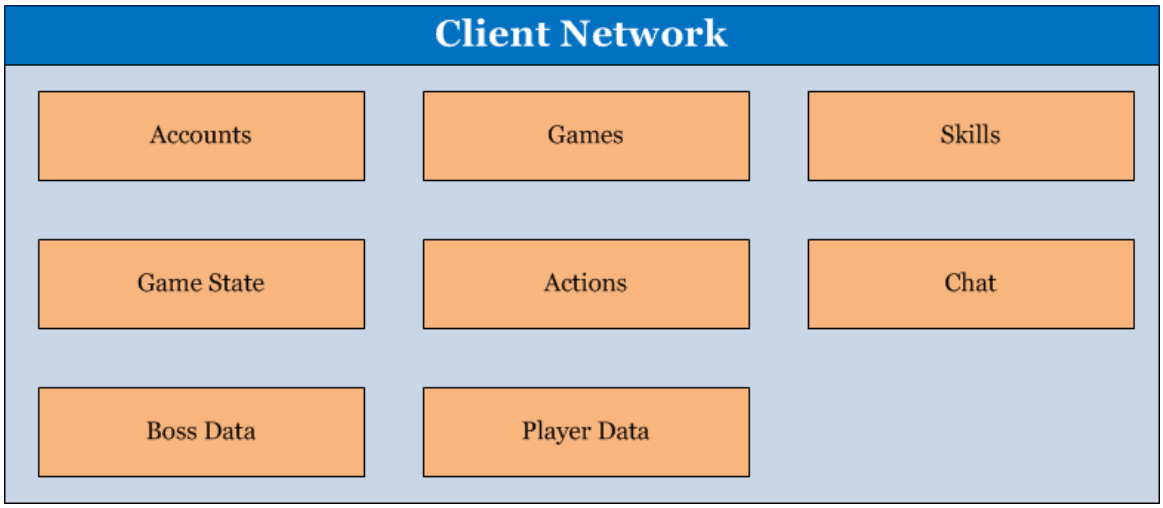


**Figure 30: Client Network overview**

Input State updates are sent to the server as they are received from the Input module. Combat interactions are sent to the server as they are received by the Character module. These messages are meant to be relayed to other clients for display of the player and syncing of game state.

Login credentials are sent after connection to the server. Game list queries, join and host game requests, and start game requests are sent as indicated to the Network module by the Interface module.

Boss data messages are sent if the client is the host client of the game. Boss data includes boss position and the state of animations being played.

### 4.6.3 Network → Interface

The client network interface forwards data from incoming messages from the server to the user interface as necessary, as shown in Figure 31. This data includes boss health data, game state data, login results, and game list data. The HUD in the user interface displays the boss' health according to the boss data received. Login successes and failures are shown in the login screen. Game list data includes the list of games, as well as success or failure of joining and creating games.

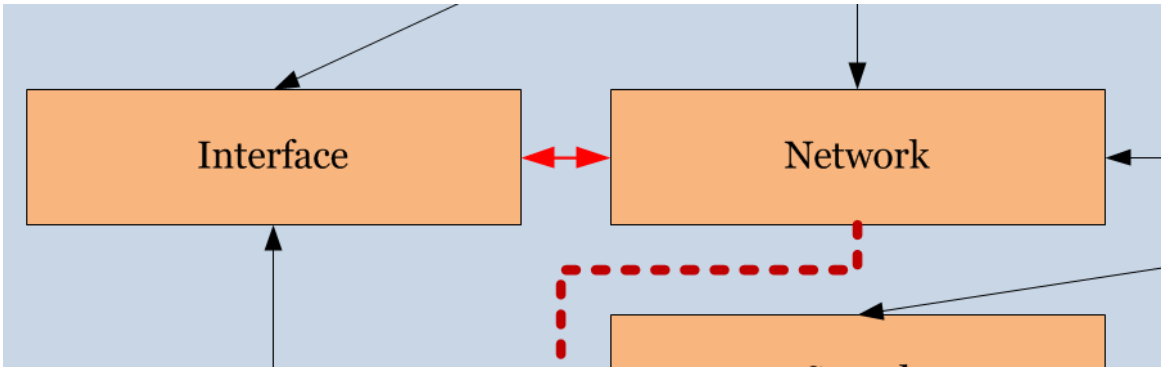


**Figure 31: Interface to Network communications**

### 4.6.4 Network → Character

The character module handles the display of characters in the game (players and the boss). The character data that is received from the server (boss data, game state data) is sent to the character handler to be processed and displayed. This information is the Input State of players, or the navigation and action data for the boss. Figure 32 illustrates this.

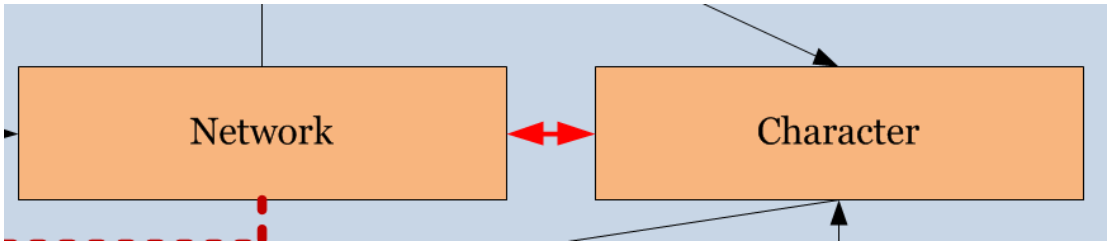


**Figure 32: Network to Character communications**

## 4.7 Physics

Core physics systems are unity collision physics and gravity. Collisions are frequently required for the characters and the environment. Desired physics systems are ragdoll physics and "force damage" processing of collisions. Ragdoll physics would be used upon a player or minion's death and while climbing the boss. "Force damage" physics would be a calculation made by the combat system that would alter the combat effect of the collision (effects such as damage dealt) based on the speed of the objects upon collision. These components are in Figure 33

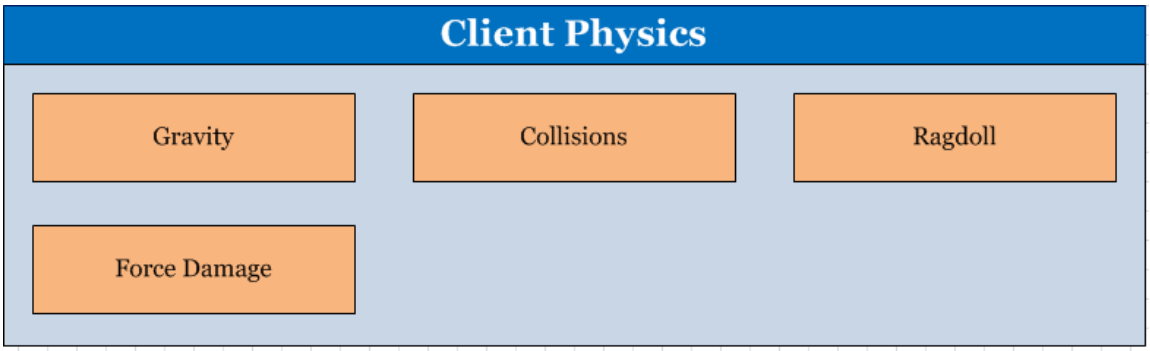


**Figure 33: Client Physics overview**

### 4.7.1 Physics → Character

Most of the combat processing is done in the character handler. The character handler gets the necessary information for combat calculations from the physics, especially the collision system. The gravity system also affects the character handler, which helps enforce the gravity and its gameplay effects. Most combat colliders used are kinematic rigidbody trigger colliders. Kinematic rigidbodies provide the utility of rigidbodies without the explicit influence of Unity3D's physics systems. Trigger colliders are convenient and efficient colliders for use in specific collisions. This relationship can be seen in Figure 34

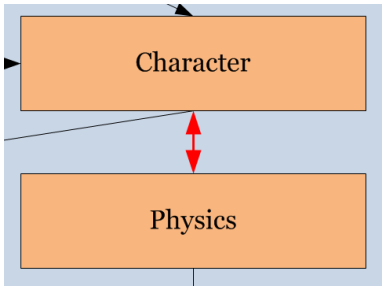


**Figure 34. Physics to Character communications**

### 4.7.2 Physics → Scene

Environmental objects in the scene are affected by the physics systems, as shown in Figure 35. The camera raycasts to detect when it needs to avoid the environment. Projectiles have colliders to detect collisions relevant to combat, or to detect environment collisions to trigger halting of the projectile.

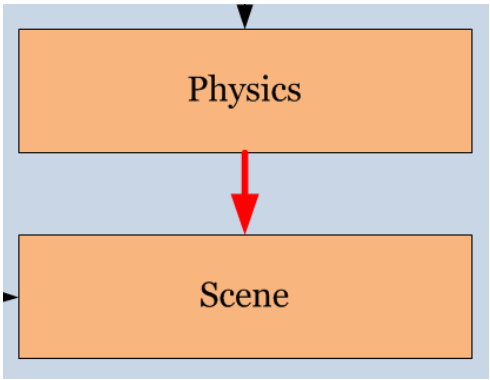


**Figure 35. Physics to Scene Communications**

## 4.8 Character

The subsystems of the Character module are shown in Figure 36. Core character features are kinesthetics, combat systems, and player vitality. Kinesthesis management involves management of character animations, for both the boss and the characters, and animation blending as appropriate. The bulk of the combat system calculations take place in the character module. Boss weak spots are processed here, as well as player ability actions and effects, and interpreting the outcome of collisions between characters and abilities. Player vitality consists of the health, stamina, and death status of each player. Required character module features are boss disabilities and player death consequences. In addition to weak points, the boss has senses that can be disabled, affecting the AI system. The character module enforces the consequences for player death.

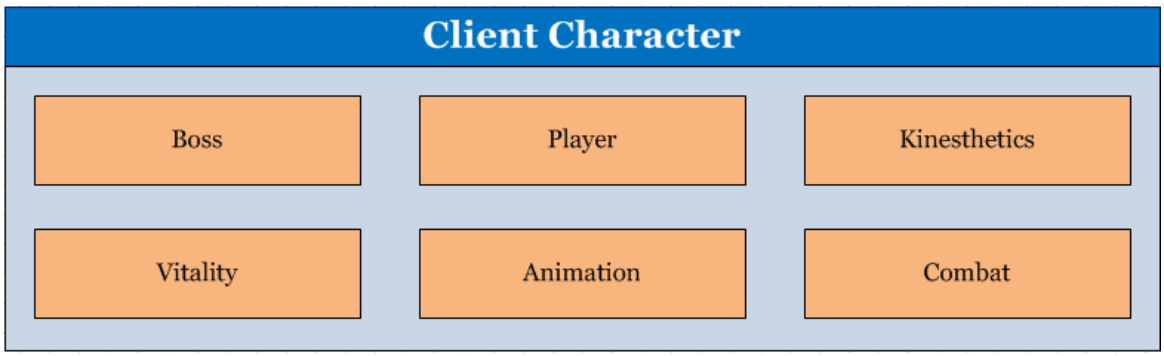


**Figure 36. Client Character overview**

### 4.8.1 Character → Network

The character module frequently sends data to the network interface as shown in Figure 37. Each client handles the combat results that involve its own player character, including damage dealt by the player, and damage inflicted upon the player. This information is sent to the network interface, so it can eventually be shared with the server and the other clients.

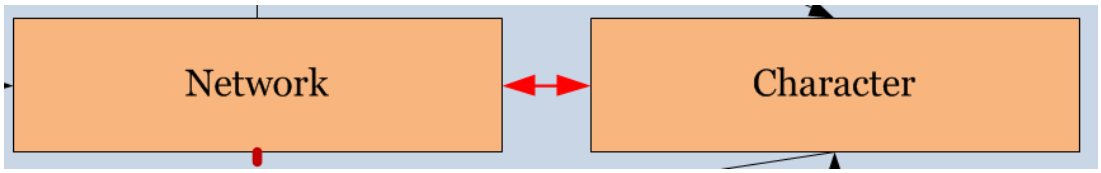


**Figure 37: Character to Network communications**

### 4.8.2 Character → Sound

The character module handles playing the sounds caused by the character as shown by Figure 38. Ability sounds that need to be generated are communicated to the sound module.

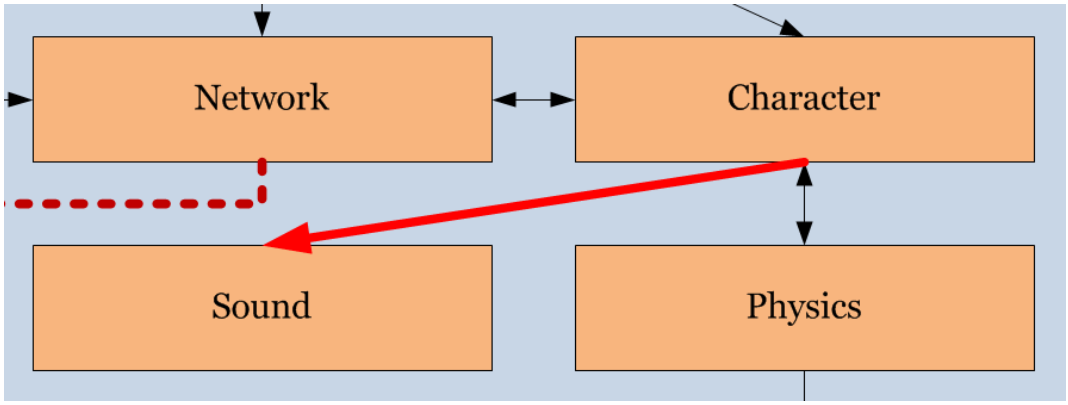


**Figure 38: Character to Sound communications**

### 4.8.3 Character Combat

The combat system is flexible to allow for balancing after gameplay testing. Each character, both the players and the boss, have values associated with outgoing and incoming damage, stamina totals and recovery, and health totals. These values are initially pre-determined. For player characters, these initial values are adjusted based on the player's class. The boss' values are adjusted based on the total number of players. This allows for difficulty adjustment that will appropriately tune the battle to the players.

## 4.9 Master Server

The multiplayer aspect of the game operates through a standard client-server network topology. The game client has a network interface which communicates with the master game server. Each player running the game client must be connected to the master game server in order to create, join and play games with other players. The master game server in turn has a number of interfaces that allow it to communicate with other the server modules, unbeknownst to the players. These other modules consist of a relational database instance, the artificial intelligence controller and a basic web server. The relationship can be seen in Figure 39.

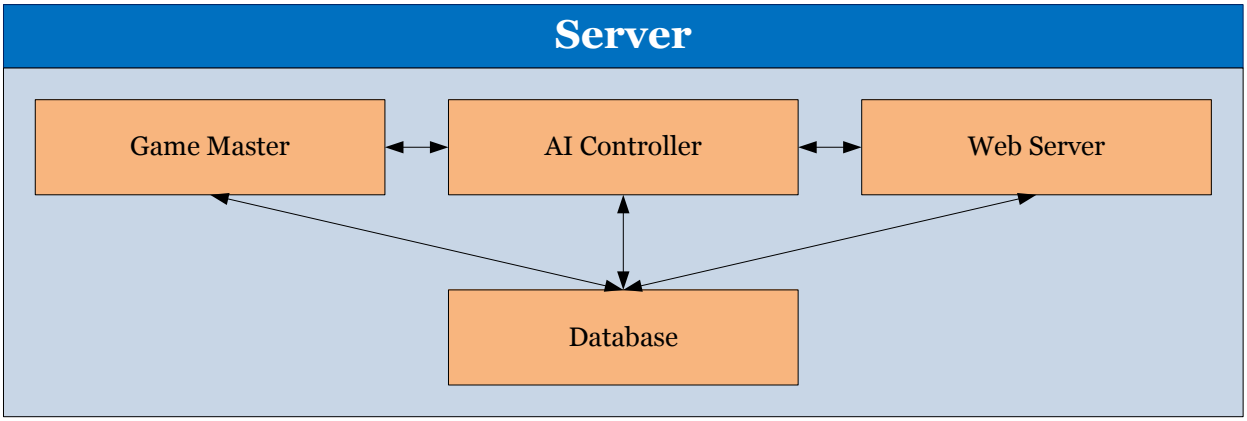


**Figure 39: Server-Side Modules**

The databases allow information to persist over time. The artificial intelligence controller computes a series of algorithms to determine the behavior of non-player characters (i.e. the boss). The purpose of the web server is to provide support forums and allow the developers to perform maintenance on the server modules. Players interact with the master game server through their game client and if they so choose, with the web server through their browser.

Within the master game server itself, there are a few subsystems, shown in Figure 40. First, there are the interface. The master must communicate back and forth with the game clients, the database instance, the artificial intelligence controller and the web server. Second, there is the subsystem that controls the game list which enables players to create, join, leave and start games.

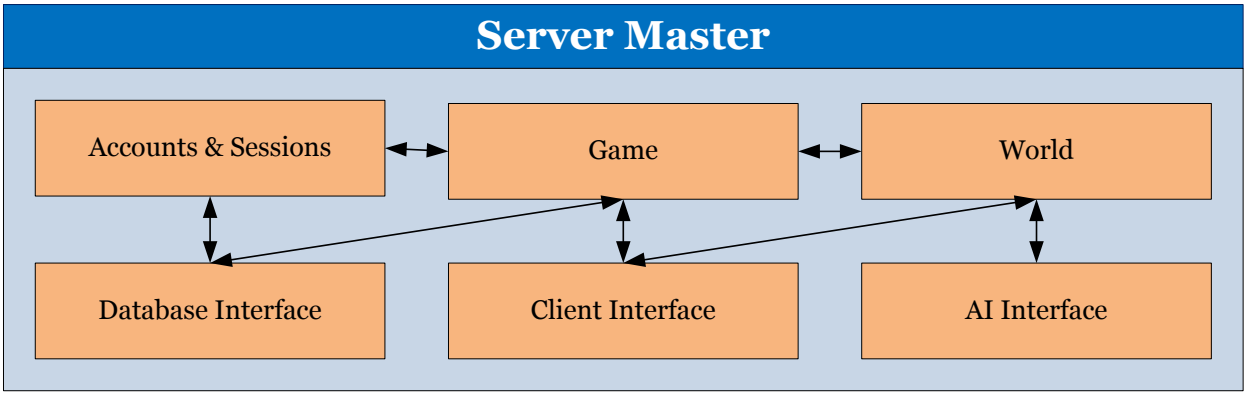


**Figure 40: Master Game Server**

Once a game has started, the third subsystem that manages an abstracted world state takes over. This stores data about player state, boss state, the positioning grid, projectiles, environment etc. without directly running an instance of the game world. As a result of the lack of player vs. player combat in gameplay, it is unlikely that a dispute between two clients would actually be in opposition. Therefore, the game client can be adequately trusted to report non-trivial events such as collisions. This means that trust is issued on a first-come, first-serve basis. Lastly, a fourth subsystem manages the player accounts and sessions to control logins, creations, password changes etc.

### 4.9.1 Interfaces

The master game server needs interfaces to communicate with the game client, the relational database, the artificial intelligence controller and the web server. Data received from the game client includes account creation and login, game management (e.g. create, join, leave, start), chat messages, player actions (e.g. walk, jump, use skill) and character modifications. Data sent to the game client includes account creation and login results, a list of games, game lobby state, chat messages, game state, boss state, player state (including one's own character) and environment state.

The relational database stores information about many of the different subsystems in the master game server. The communication between the master and the database is accomplished via SQL queries and result sets. Persisting data is vital to building a community and user base in an online environment. The database is crucial to the rest of the server modules and subsystems.

The communication to the artificial intelligence controller consists entirely of game state data and behaviors. The master game server assembles all of the information about the game world and then send it to the artificial intelligence controller. This data is used to compute a series of algorithms and then ideally arrive at a behavior. That behavior is sent back to the master game server which attempts to execute the series of decisions and actions by communicating with the game clients. Once the behavior has finished executing, the master sends the results to the artificial intelligence controller again to determine the effectiveness of the behavior and to request a new behavior. The artificial intelligence controller is entirely unaware of individual games that are in progress. It exists as a singular computational instance that takes game state data as input and provides a behavior as output—similar to a function, albeit more complex.

### 4.9.2 Game List

Multiplayer operates by the storage of separate game world instances. After a user has logged in, he or she is first be presented with a list of available games. The user may then choose to join an existing game or create a new one. The game client sends data indicating a challenge to join a game with certain credentials (e.g. name and password) create a game with certain properties (e.g. name, password, public, user limit). Assuming the credentials are correct, the login will pass. Assuming the game name does not already exist, the creation will pass. Passing will cause the game master to create a record in the database either with a new game or with a new user session in a game lobby.

Once in the game lobby, the creator of the game has a few options: start the game, leave the game or boot a player. Once the game has started, no more players may join. At this point, the game list subsystem is not responsible for the actual carrying out of the gameplay. However, it does maintain a record that the game is in progress (to avoid duplicate names) until the game state hits a win or loss condition—at which point, the database record for the game is removed. Regular players only have the option to leave the game. All players may chat with one another

in the game lobby before the game starts. If the creator leaves the game, all of the players are automatically booted and everyone returns to the game list screen.

### 4.9.3 World State

The master game server must keep track of the state of each individual game as its being played. Due to the scope of the project and the expected cooperation between players, only a limited representation of the game state is actually necessary to facilitate gameplay. In fact, many of the technical decisions along with the responsibility for reporting the game state are delegated to the game client. The reason this is possible stems from the fact that no two players is fighting against one another, and would not have a conflict of interest in terms of a decision. This does not of course rule out the possibility of hacking by reporting false collisions, but that sort of protection is not within the scope of this project.

Specifically, the world state subsystem is in charge of all of the player and boss data, including health, energy, skills, position, orientation and perception among other things. It is also in charge of projectiles and the environment. While the system does not determine when objects actually collide, it is in charge of synchronizing all of the game clients. Any slight error or disparity that occurs and goes unchecked will grow over time and eventually result in a gross difference of game state between two clients. Therefore, it must be corrected. A major component of the positioning is a navigation mesh (node graph), placed around the map. It is used to approximate the exact placement of a character in the world into discrete positions. This is very useful for the artificial intelligence controller in terms of path finding and computing patterns with positions.

### 4.9.4 Accounts & Sessions

A user must be logged into an account to play the game and each account has a password associated with it. This requires having first created an account and then passing a logon challenge. Once the logon challenge is passed, a session is created that is linked to the account the user entered. Creating an account creates a new database record. Creating a session also creates a new database record. Only one session may be associated with an account at a time, so part of passing the logon challenge is ensuring that the requested account currently has no sessions.

## 4.10 Database

The database instance is a software package separate from the other server modules. It is a standard implementation of a SQL-based relational database. This might be something like MySQL, sqlite or MSSQL. Primary keys and indexing are used to perform efficient queries. Ideally, the chosen implementation also supports stored procedures. This allows for highly optimized queries as well as an added layer of security against injection. The data is organized into databases that contain tables which hold rows. The data that must be stored includes accounts, records, sessions, games, chat, a log, the artificial intelligence state and some things for the web server (e.g. forums), shown in Figure 41.

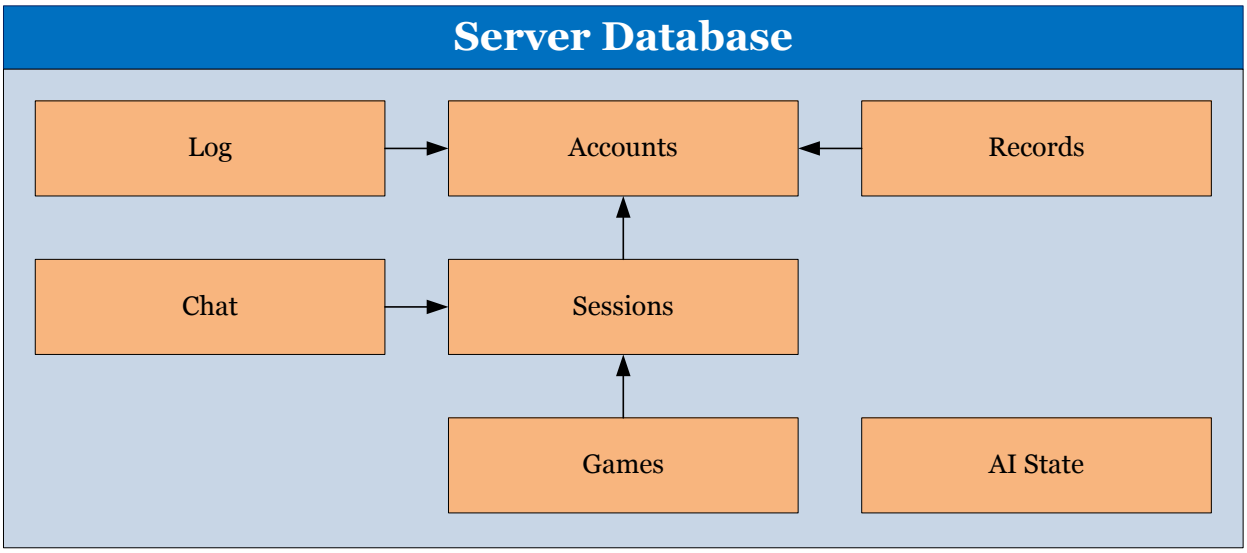


**Figure 41: Relational Database**

The log keeps track of when login attempts have been made and when sessions have occurred. Other things such as chat and game creation are also logged. Accounts store a unique username with an associated password. To protect the user's privacy, passwords are stored in a cryptographic format. All of the character data are stored with the account. Records contain all of the game results for a particular account, so that a player's gaming activity history is preserved over time. Sessions simply contain a list of the active players that are logged into the master game server, and a foreign key to the account with which they are associated. The games table stores a list of games and has a foreign key directly to sessions. Once a player has logged in and has a session, they may start playing games. The artificial intelligence state contains data that is used to compute the different algorithms in the controller.

## 4.11 Artificial Intelligence

The artificial intelligence controller is largely responsible for choosing the behaviors of non-player characters. It is also responsible for providing implementations of path finding, state machines and other common mechanisms found in games. However, the behavioral system severely overshadows these common mechanisms. It is broken up into a few different subsystems, shown in Figure 42.

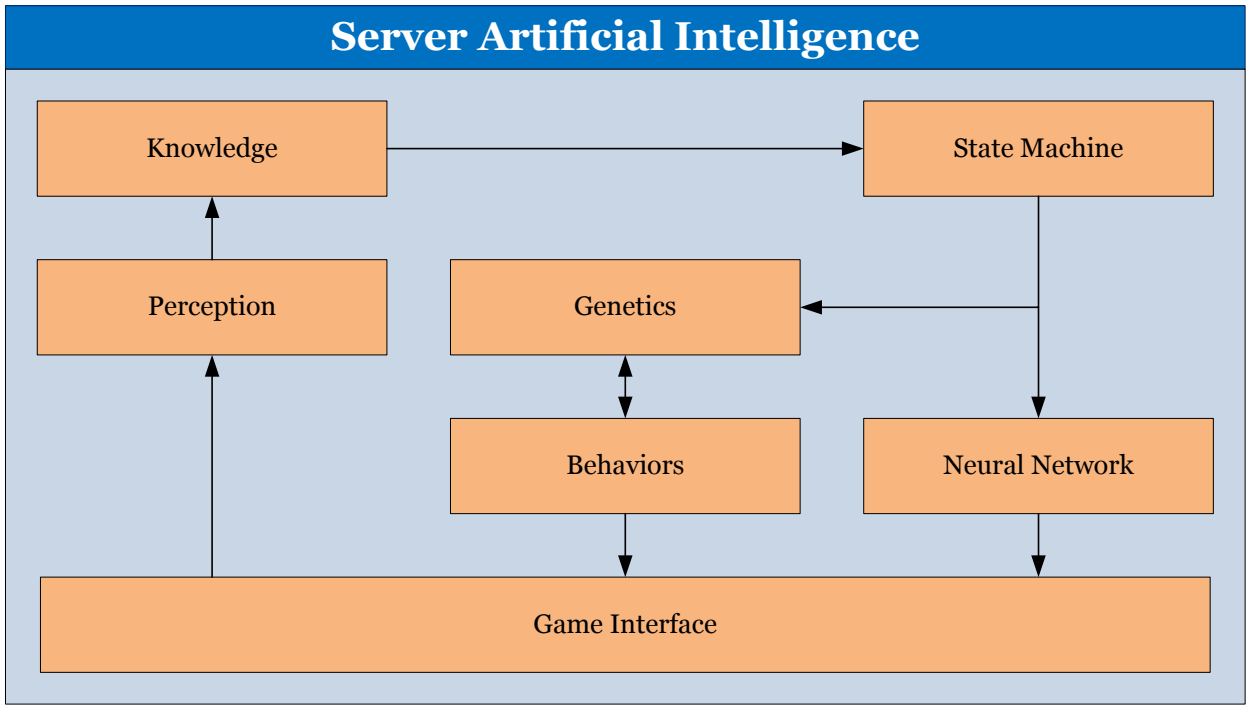


**Figure 42: AI Controller**

The base system is the interpretation of behavior trees. These trees contain logic and actions within their structure and can be used to carry out complex decision-making. These trees can be broken into smaller sections that can be combined and manipulated using the second subsystem which employs the principles behind genetic evolution. Furthermore, the state machine of the boss character is controlled by an artificial neural network. This allows a range of different inputs and game state data to influence the boss' behavior.

### 4.11.1 Finite State Machine

The finite state machine (FSM) controls the boss' actions and exists once for every game instance, just like the boss himself. The states of the FSM include Attack, Fortify, Hide, Idle, Intimidate, Last Stand, Panic, Pursue, ~~Spawn~~ and Wander. Each state must implement three

methods, *Enter()*, *Tick()* and *Exit()*. All three functions take the game world state and the boss' entire system (including the state machine, actually) as parameters. The *Enter()* method is called when the state is first entered and the *Exit()* method is called when the FSM is transition to another state. The *Tick()* method is called continuously in the current state of the FSM.

The FSM allows for it to be told which state it should transition to. The FSM itself does not contain a transition model, rather, more sophisticated mechanisms are employed to handle state transitions. Namely, the artificial neural networks control the state machine. This is explained further on.

The states are relatively self-explanatory. The Attack state is when the boss character is attacking one or more player characters. It runs the majority of the behavior generation through genetic algorithms. The Fortify state allows the boss character to immediately expend a large amount of energy to raise his defense. The Hide state calculates a path to a known hiding spot on the map. The Idle state simply makes the boss character cease movement. The Intimidate state is used to dissuade players from hiding; it allows the boss to cause damage despite not being able to make immediate physical contact with the players. The Last Stand state is when the boss is close to death, and chooses to stop conserving energy in a "guns blazing" effort to stay alive. The Panic state is when the boss has lost most of his perceptual subsystems and decides to no longer rely on them for behavior, but acts in an erratic manner. The Pursue state occurs when the boss has seen a player but is not nearly close enough to attack; he then uses path finding to follow the closest player until he is within range to attack. The Spawn state has been removed, because we chose to not include any sort of "minions" in the game. The Wander state finds a path between the closest node to the boss and a random node in the navigation mesh; this effectively allows him to wander about the map looking for players.

### 4.11.2 Path Finding

The path finding is implemented using Dijkstra's graph search algorithm. Paths are constructed from nodes and edges that have been calculated from the terrain. This generates a navigation mesh, stored in an XML file. The game server loads the entire navigation mesh once when it starts, and uses that for further calculation during gameplay. The pseudocode for the algorithm is straightforward:

```
1  function Dijkstra(Graph, source):
```

```
 2      for each vertex v in Graph:
 3          dist[v] := infinity ;
 4          previous[v] := undefined ;
 5      end for ;
 6      dist[source] := 0 ;
 7      Q := the set of all nodes in Graph ;
 8      while Q is not empty:
 9          u := vertex in Q with smallest dist[] ;
10          if dist[u] = infinity:
11              break ;
12          end if ;
13          remove u from Q ;
14          for each neighbor v of u:
15              alt := dist[u] + dist_between(u, v) ;
16              if alt < dist[v]:
17                  dist[v] := alt ;
18                  previous[v] := u ;
19              end if ;
20          end for ;
21      end while ;
22      return dist[] ;
23  end
```

The time complexity of Dijkstra's algorithm is *O(E + V)* where *E* is the number of edges and *V* is the number of vertices. The pathing system computes the shortest route from one node to another and sends that list to the client. The boss character then follows the route, as the client also contains the navigation mesh.

### 4.11.3 Behavior Trees

Behavior trees are directed acyclic graphs whose nodes are computable in some manner. The typical implementations of behavior trees in games are very similar to hierarchical finite state machines such that each node in the tree is itself, a finite state machine. The motivation behind implementing behavior trees like this is to reorganize transitions between states—and consequently behaviors—into a flexible and reusable data structure. We will look at known implementations and expand where it is prudent to do so.

The AI in Halo 2 utilizes behavior trees. The system essentially consists of behaviors and impulses. Each behavior is a node in the tree that contains a finite state machine that internally decides its state. The state chosen then triggers a change of control to another finite state machine in the tree, so on and so forth. This can be accomplished by prioritized lists, sequential lists, looping through or random choice of behaviors; or by something known as impulse decisions. Impulses are like weights for each child node. All the child nodes of a behavior are allowed to compete at runtime over which is to be executed. The child node with the highest

impulse value is the winner. It is also noteworthy that Halo 2 has a knowledge model that is populated by *perception* and is accessed by the behaviors. All of the Halo 2 behavior trees are static at compile time.

The AI in Spore also utilizes behavior trees. This system was designed with the Halo 2 implementation in mind, but some positive steps have been taken to improve upon it. One of the most important improvements is that the logic for deciding which child node is chosen to be executed was moved into the parent node. Here, the parent is known as a *decider node*. This modification is critical because the behavior is no longer responsible for deciding when or why it should be run; that responsibility is left to other behaviors. Impulse decisions are still available, but are not relied upon. Spore also uses a knowledge model but it is purely transient between executions, unlike those in Halo 2. All of the Spore behavior trees are also static at compile time.

This section outlines technical requirements for behavior trees that are significantly more malleable than traditional implementations in order to create highly dynamic game play. The first modification is lowering nodes to a finer granularity. Rather than being complex behaviors and state machines, they are simple atomic actions; and decisions are implemented as conditions. The second modification is to create decorator nodes. Decorator nodes are allowed to supersede the traversing of its child nodes. A decorator may implement *any* downward traversal scheme.

These changes result in larger and more complex behavior trees, comprised of shallow computational units. Therefore, much more logic is derived from the structure of the tree itself, rather than residing within the nodes. Reasoning for this exposure of logic develops from the third modification, that these behavior trees *will not* be static at compile time. In fact, further mechanisms is used to develop new trees at runtime. Highly malleable logic facilitates truly dynamic behaviors and ideally improve the realism of game play.

#### 4.11.3.1 The Graph

As stated earlier, the behavior tree is a directed acyclic graph beginning with a single root node. Traversal is done recursively, depth-first and in preorder. That is, the root is executed, followed by the traversal of each child node from left to right (traversal consisting of execution and further traversal of its child nodes). Each node in the graph is an action, a condition or a decorator.

Actions always have their child nodes traversed in the traditional way. A condition may have any positive number of child nodes. During traversal, it chooses a single child node to be further traversed. Decorators modify the traversal procedure of their own child nodes to customize the control flow of the behavior tree unbeknownst to any subsequent or prior nodes. The traversal order can be observed in Figure 43.

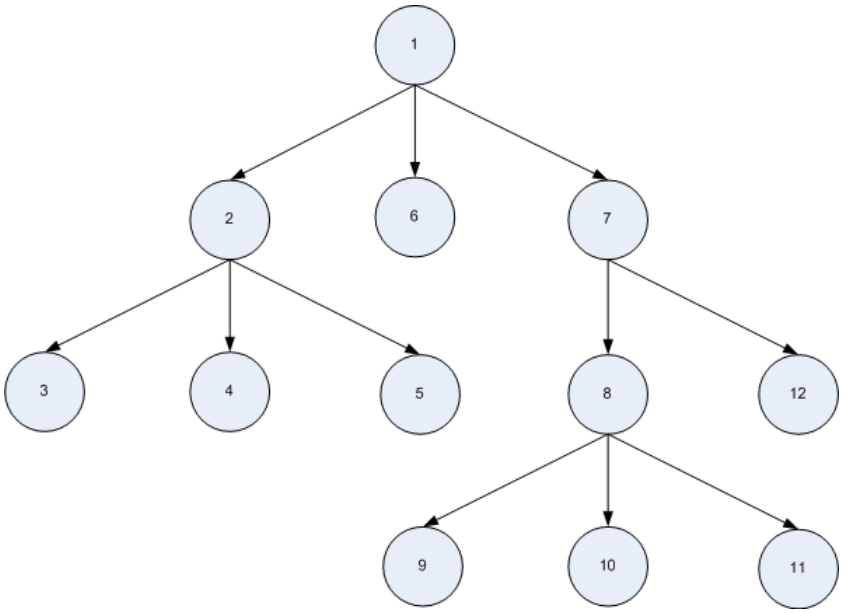


**Figure 43: Behavior Tree Structure & Traversal**

While the tree is comprised of many atomic computations, it can also be thought of as a single procedure when an actor is executing it. A game character will only ever be traversing one behavior tree at a time. Each behavior tree is guaranteed to cease execution eventually. The actor will not be performing any other tasks while executing a behavior tree. It is of course possible to interrupt a task while executing, but should be avoided unless a significant state change warrants it (e.g. the actor dies). It is implicit that the graphs do not contain cycles and do not create circular logic. However, it is important to note that this responsibility lies with the mechanism that generates behavior trees.

There are three stages to the execution of a node. First is a constructor, where the node is allowed to initialize class members and gather game state data. The constructor is followed by the running of the node. During the run stage, the node carries out the guts of its "payload". For an action, this might be an animation or if it is a condition, some logical evaluation. When the node is finished running it returns a result. Results contain information indicating how the

traversal of the behavior tree should continue. Finally the destructor is called after the run stage is completed. It should be noted that these constructors and destructors are not called during object instantiation, rather when the node has been reached for execution during traversal.

### 4.11.3.2 Knowledge Model

The knowledge model is nothing more than a data structure that contains a variable amount of key/value pairs. A reference to this data structure is passed to each node as the behavior tree is being traversed. It is important to clarify what populates the knowledge structure, what occurs when the data in the knowledge structure is altered and the permanence of knowledge structures. Sharing of data between nodes during the execution of behavior trees is essential.

| | |
|---|---|
| **Property_A** | "Rodion Romanovich Raskolnikov" |
| **Property_B** | "Feodor Dostoevsky" |
| **Location_X** | 124.77 |
| **Location_Y** | 98.64 |
| **Location_Z** | 43.43 |



**Figure 44: Sample Knowledge Model**

Immediately before a behavior tree is to be executed, a new knowledge structure is created and filled with data about the current game state, shown in Figure 44. It will not contain any data that was altered during the prior execution of any other behavior trees. Consider it a fresh copy. As the knowledge structure is passed between each node the data may be changed or added to. Subsequent nodes that are executed will see the altered and additional data in the knowledge structure. In no way will any change of the knowledge structure directly affect the game state. When the behavior tree has finished executing, the knowledge structure is destroyed. Behavior trees are not responsible for learning; more advanced mechanisms is applied to that purpose.

### 4.11.3.3 Actions

Action nodes are used to carry out actions. In the context of a game, this might be looking at a point, swinging an arm or firing a gun. However, an action's functionality can be much less complex (e.g. manipulating a number in the knowledge structure). This is important because the granularity of the actions in the behavior tree can be lowered to fine computational units. In this way, actions are not only behaviors; they may be complex, but are likewise capable of only

influencing subsequent nodes in subtle ways. That influence is crucial to introducing truly dynamic behavior in the game play. This iscome clear when further mechanisms are explained. Action nodes are traversed by first executing the action node, then traversing each of its child nodes from left to right; this barring the action having chosen to abort execution.

#### 4.11.3.4 Conditions

Condition nodes compute a function and choose one child node to be further traversed. They are very similar to the aforementioned decider nodes. Unlike Spore's decider nodes however, condition nodes do not allow delegation of choice to the child nodes (i.e. no impulse decisions). Due to the fact that our nodes are not state machines, this is an unnecessary and potentially hazardous complexity.

The outputs of a condition node are its children. Boolean conditions have exactly two outputs. The left output at position zero corresponds to *FALSE* and the right output at position one corresponds to *TRUE*. As far as the imagination dares stretch, Boolean algebra can be incorporated throughout the structure of these behavior trees.

Conditions may also be nested. For instance, a condition for the logical *AND* may take two or more other Boolean conditions as parameters. When the *AND* condition is executed, it internally executes each of the Boolean conditions that were passed as parameters. Assuming every condition returns a result of *TRUE*, the *AND* condition also returns a result of *TRUE*—or otherwise *FALSE*. Rather than having say, 5 levels of conditions in a parent/child chain, each condition is executed in a single node. This allows the reduction of tree complexity where possible, note Figure 45.

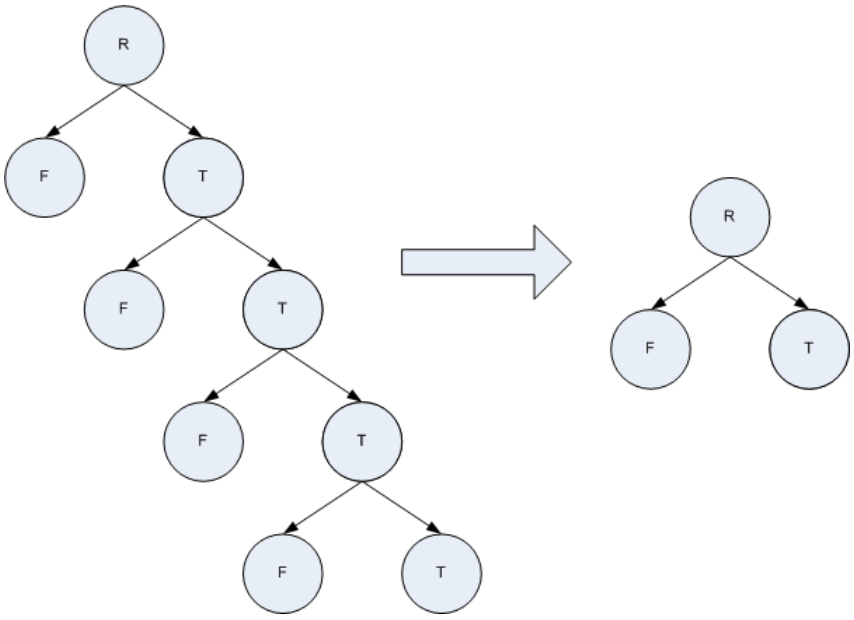


**Figure 45: Logical Reduction and Encapsulation**

### 4.11.3.5 Decorators

Decorator nodes liberally implement the decorator design pattern. Traditionally, decorators are used to add new behaviors to an object at runtime. The decorator nodes perform the same function within the behavior tree, but not in the object-oriented programming sense. To modify how its child nodes are to be executed, a decorator node is allowed to override the traversal algorithm. This gives rise to a surprising flexibility in the structure of behavior tree logic.

Decorators always returns a result indicating success when executed. As previously stated, once the execution is complete the decorator node takes control of how its children are to be traversed. This has any number of possible implications. A potential type of condition could be created using a decorator that traverses some—but not all—of its child nodes. Two practical applications are to employ looping through children nodes and to execute child nodes in parallel. The looping can be a fixed number of times or even based on a nested condition.

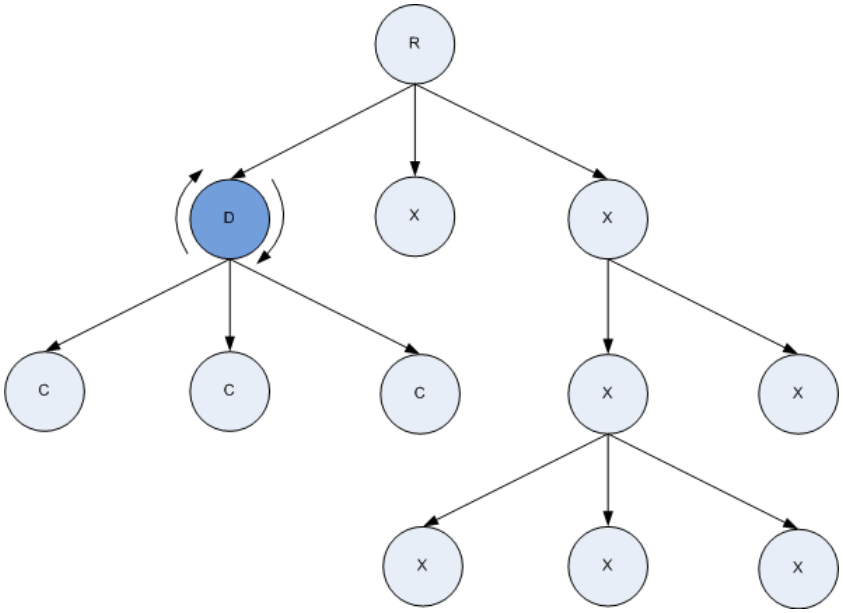


**Figure 46: Decorator Traversal Example**

The behavior tree in Figure 46 will start by executing the R (root) node. It will then reach the D (decorator) node. This node can choose how it will execute any of the C nodes; it cannot influence the R or the X nodes directly. Of course, all of the nodes share the same knowledge structure and indirect manipulation is possible.

Using the knowledge structure and decorators, a list could be iterated through while traversing each child node per item (like a foreach loop). The emergence of this behavior exists purely in the structure of the behavior tree; no code would have to be handwritten to iterate through a list and perform computations on it.

### 4.11.4 Genetic Algorithms

Genetic algorithms are simulations that apply the fundamental processes of natural evolution to a search for solutions within a given problem space. The only requirements of the problem are that the solutions can be encoded and that it is possible to determine the fitness of a solution (i.e. how effective the solution is at actually solving the problem). Starting with an initial "population" of random solutions, the fitness of each is evaluated by the simulation. The solutions with the highest fitness scores are selected for reproduction. Since the solutions are encoded—and DNA is of course encoded, the genetic operators of crossover and mutation are applicable. The chosen solutions are then combined using these operators to form new solutions. When the next generation of these solutions has been spawned, each will have its fitness

calculated and the selection process will continue. Over many generations, effective solutions is found and combined to ideally create more effective solutions. The evolutionary approach to problem solving has an astonishing range of applications including bioinformatics, phylogenetics, computational science, engineering, economics, chemistry, manufacturing, mathematics, physics and of course, game play.

The Creatures series is the one commercial game whose AI most stands out in this area. It serves to simulate a handful of fundamental biological processes. Each creature (e.g. norn, ettin, grendel) has an internal representation of a brain, a body, organs, perception, hormones, a metabolism and genetic material. The game play then relies on emergent behaviors that arise from these biological mechanisms. When creatures procreate—within their species of course—their genetic material is crossed over and mutated. This genetic material dictates the structure of the brain, the physical appearance of the creature, the types of neuroemitters and neuroreceptors, the durability of organs and responses to stimuli etc. Since just about every aspect of a creature's behavior is encoded in the genes, they end up passing behavioral traits to their offspring. Given the environmental dangers of cold, disease, piranhas and grendels combined with genetics and reproduction, the creatures are prone to evolution. The result of this evolution is dynamic behavior passed between generations as encoded genetic material.

It appears that the few other remaining incorporations of genetic algorithms in commercial games are related to the organization of artificial neural networks; unfortunately, there is little to no information available revealing these mechanisms in any major titles. Regardless, in this sort of evolutionary strategy, all of the behaviors and states are defined at compile time and the AI learns when to use them during runtime. Neural concepts is explored further later on, but the process deserves some explanation. Essentially, a neural net has an input layer (i.e. the game state data) and an output layer (e.g. action or state). Between these two are any positive number of hidden layers that are comprised of neurons and synapses (i.e. the connections between neurons). The weighting of synapses—and sometimes the actual composition of the neural network—are the encoding of solutions. This means that the neural network is computed and arrives at an output. This output is compared to the desired output, and the weighting of the edges is then modified to reflect the desired output. Over time, the actual output and the desired output converge to form a trained and functional artificial neural network.

This document outlines technical requirements for genetic algorithms that construct the Milk implementation of behavior trees as the encoding for representing "solutions". More often than not, the solution encoding for genetic algorithms is just a string of bits or digits. Each digit may represent something as simple as turning left or turning right when solving a maze; or it could be something as complex as a piece of an instruction to be executed by an interpreter. However, any other data structure that can serve as the encoding for solutions can also be used as the medium for these algorithms. Naturally, the graph structure of the behavior trees lends itself to structural changes at specified points. However, an "ideal solution" does not exist in terms of the behavior trees. Rather, the goal is to simply generate *new* and *better* behaviors than before.

The solutions is constructed by representing pre-defined portions of a behavior tree as a chromosome and then bonding multiple chromosomes together to form a genome. Given any genome, the underlying behavior tree can be extracted and—of course—executed. Each genome can be thought of as a single "organism" capable of producing its own behavior tree. Two organisms can procreate—using crossover, that result in a new organism with a new genome that produces a different behavior tree. Each organism is analyzed, executed and scored. Then, any number of different methods can be employed to select which organisms is used for reproduction and create the next generation.

#### 4.11.4.1 The Graph

The data structures used to represent the genetic material of these solutions are genes, chromosomes and genomes. These structures are analogous to the behavior tree graphs. The reason for the similarity is that underneath the genetic graph is a behavior graph. There is a one-to-one ratio between behavior nodes (e.g. actions, conditions, decorators) and genes; that is, each gene contains one node. A chromosome is just a standard directed acyclic graph in which each node is a gene. Chromosomes are bonded together by their root and leaf genes to form genomes. A genome is considered a complete behavior. A population is used to store each generation of organisms (i.e. genomes) along with some data about their fitness. When the genetic graph is traversed, it is done so to extract and assemble the underlying behavior tree into a single executable form.

Since each gene encapsulates one behavior node, the actual graph is connected between genes and not behavior nodes. It is not until the behavior nodes are extracted that they are connected.

However, the connections between the genes are preserved as connections between the behavior nodes when the final tree is constructed. Observe the graph layering in Figure 47.

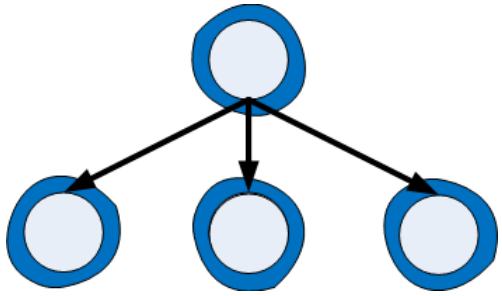

**Figure 47: Behavior Nodes Encapsulated by Genes**

Each behavior node (light gray circle) exists inside of a gene (amorphous dark blue circle). The connections between genes are pre-programmed into chromosomes. The extra level of abstraction with individual genes is not necessary in the formation of chromosomes or genomes; however, it is designed this way in order to eventually allow mutation or random generation of chromosomal gene formations. Currently, that is out of the scope of this project.

A chromosome is a tree of genes that represents an irreducible genetic unit. That means any genetic operators (crossover and mutation) do not manipulate anything below the chromosomal level of abstraction. For the purposes of this project, chromosomes is simple partial behaviors that are defined at compile time. Chromosomes can take on many different structural forms. For instance, a chromosome might be used to walk forward, swing a weapon diagonally or target the nearest enemy. An example structure of a chromosome might look just like the genes example or, the chromosome might be a more complex series of actions and decisions like in Figure 48 and Figure 49.

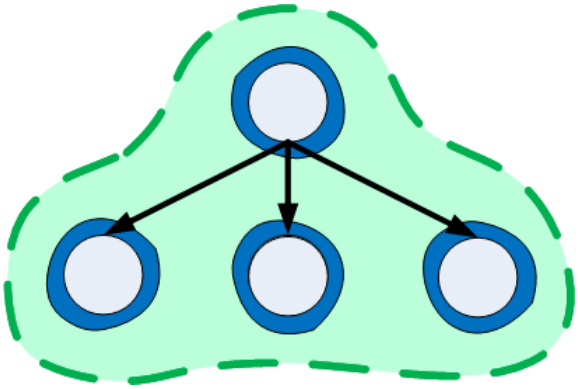

**Figure 48: Example Chromosome A**

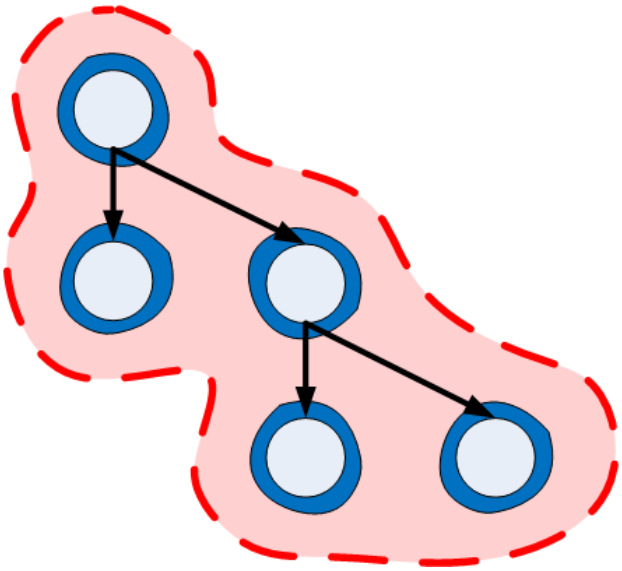

**Figure 49: Example Chromosome B**

These chromosomes can be bonded together to form a genome. The bonds happen at specified "output" genes that are leaf nodes in the parent chromosomal tree. These are bonded to the root (top most) node in the tree of another chromosome. Two types of bonding can occur. A standard bond simply makes a root gene of one chromosome the sole child of an output gene from the other chromosome. A fusion bond however, can take two forms. One gene actually *consumes*—or replaces—the other gene. The default is for the root gene to disappear and transfer its children to the output gene. The inverse is also possible. A genome might look like Figure 50.

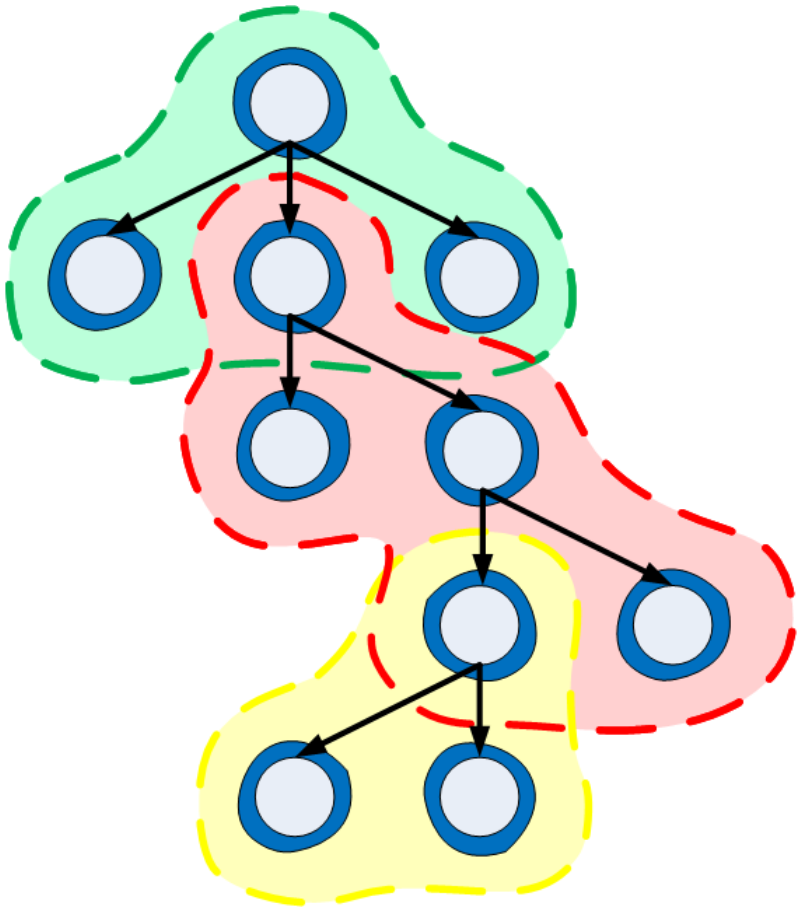

**Figure 50: Example Genome**

It is clear there are three different chromosomes that have been bonded together to form a genome. Here, each bond is a fusion bond but could very well be standard bonds or some combination of the two. This genome is an example of an organism—what might be called a potential "solution" during the computation of a genetic algorithm. Given a fully assembled genome, the underlying behavior tree can be extracted for computation and further analysis. For this example, the behavior tree would look like Figure 51.

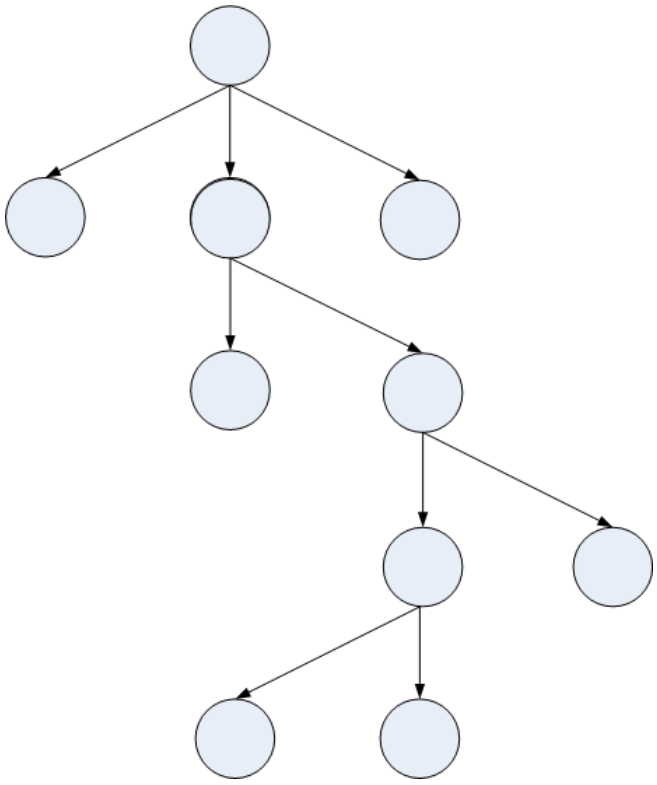

**Figure 51: Behavior Tree Extracted from Genome**

By continual iteration of analysis, selection and reproduction, the algorithm produces new generations of these organisms. The transmutation of this behavioral and logical genetic material results in the emergence of competitive and heavily dynamic game play. The results of these evolutionary strategies are artificially intelligent behaviors that were not engineered by humans. These systems understand their repertoire of available behavioral primitives and combines them in unique and unexpected ways, very similar to the evolution of organic systems on our planet.

### 4.11.4.2 Simulation

To actually run these evolutionary computations, it takes more than just genes, chromosomes and genomes. Evolution of course requires more than just a species to exist in a given instant. Time must pass, the environment changes, organisms die and some things is successful where others are not. To simulate the remaining elements of the evolutionary process, there are a handful of subsystems that take care of reproduction, fitness evaluation and the interface to the game world. Specifically, these subsystems include a breeder, a gene pool, a fitness calculator, environmental stress (natural selection), a knowledge layer (perception), and populations of genomes. Finally,

there is a "world" that orchestrates all these subsystems to operate in concert. A game that implements this library primarily interacts with the world system.

The breeder is in charge of applying the different available types of crossover and mutation to one or two organisms at a time. This subsystem is initialized when the youngest generation has been fully analyzed and a new population of organisms is required. The gene pool contains all available chromosomes and is accessible by mutations which use it to manipulate a genome. A fitness calculator contains the mechanisms for applying static and dynamic analyses to score an organism. For dynamic analyses, the knowledge model is required to assemble data about the game state via perception. The calculator is also in charge of normalizing these scores so that any organism from any generation can be compared to any other organism. Once the youngest generation has been scored, an environmental stress is applied to choose which the parents of the next generation. This cycle continues until there is human intervention to update the system, reset the simulation or permanently store behaviors.

#### 4.11.4.3 Procreation

Organisms can reproduce in two fundamental ways: asexually and sexually. These two methods of procreation serve evolution by the genetic operators of mutation and crossover, respectively. Sexual reproduction is also subject to mutation. There is a variety of different mutation and crossover techniques. Once again inspired by real processes that occur in nature during reproduction, mutations can take the form of addition, deletion and duplication. Likewise, crossover can take the form of insertion and translocation. The processes are illustrated in Figure 52 for biological chromosomes.

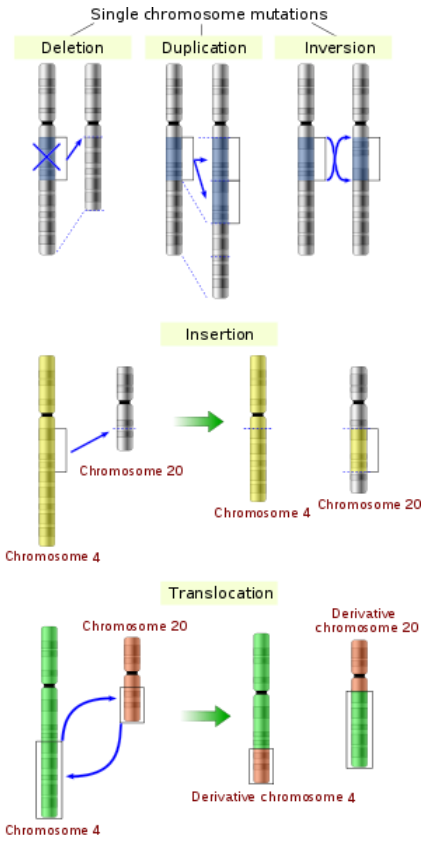


**Figure 52: Real-World Genetic Mutations**

The inversion mutation simply is not practical given the top-down nature of the genetic material. Since there is also a pre-defined gene pool of chromosomes, other types of mutations can be implemented (e.g. addition).

In the addition mutation a new chromosome is randomly selected from the gene pool and randomly bonded to one of the available output genes. In deletion, one bond is removed. This has a cascading effect, such that any bonds or chromosomes that are children of the deleted bond must also be removed. This prevents vestigial pieces of genetic material that lies dormant as part of the genome without ever being connected to the main tree. Duplication is similar to addition, except that the new chromosome comes from the genome being mutated, not the gene pool.

The two types of crossover are insertion and translocation. Insertion is when a chromosome is taken from one parent and bonded to an output node of another parent. The resulting genome is the child organism and is part of the next generation assuming it passes some standard challenges (e.g. no cycles, below maximum tree height threshold). Translocation is when a bond from the

first parent and another bond from the second parent swap the root gene part of the bond. Randomly, one of the two subsequent genomes is chosen as the child and the other is discarded.

Both mutation and crossover are extendable, so games that implement this library may have their own custom genetic operators. Additionally, the game designers may choose to disable any of the default genetic operators explained earlier.

#### 4.11.4.4 Fitness

There are two types of analyses when determining fitness scores: static and dynamic. Static analysis occurs immediately after a new generation of organisms has been spawned. This means that the intrinsic structure of the genomes is checked. This can be used to reject genomes with cycles, genomes that have a graph height exceeding some arbitrary threshold or perhaps to discourage certain chromosomes being bonded together (e.g. target closest enemy) because the sequencing is inefficient. The static scores are preserved apart from the dynamic scores.

Dynamic analysis occurs at the whim of the game itself. Only when the AI system is ready to execute a brand new behavior tree will the evolutionary algorithm perform dynamic analysis and continue its computations. The knowledge model is essential to carrying out dynamic analysis. The knowledge "layer" computes all of the game state data into variables that can be relied upon by each dynamic analysis. This is first done before the behavior tree is executed. The behavior tree is then executed. The knowledge layer again computes the game state data after execution has been completed. Each dynamic analysis compares the state before the execution to the state after the execution and determines its score. The score of each dynamic analysis is summed and that composite score is assigned to that organism.

The default implementation of the fitness calculator normalizes the fitness scores of a generation. For the static scores, this is done immediately after they are all calculated. For the dynamic scores, normalization is also done after all the scores have been calculated. That of course requires every behavior tree in the population to have been extracted, analyzed and scored. The normalization procedure is to first calculate the sum of all scores, called $S$. Then the normalized score $N_i$ for some individual i is its dynamic score $D_i$ divided by $S$. That is, the normalization formula is $N_i = D_i / S$ Consequently, the sum of all the normalized scores for a population should be exactly 1.

### 4.11.4.5 Selection

The operant force that drives the process of natural selection is environmental stress. In nature, this is typically predators, disease or harsh climate. In terms of a genetic algorithm, the fitness of each organism must be computed and then one of a few techniques can be used to determine which organisms is selected for reproduction. The default methods of natural selection are fitness proportionate, stochastic universal sampling, tournament style and truncation. This subsystem is initialized with the breeder because the breeder uses the natural selection to produce new generations.

Fitness proportionate selection is when the probability of an individual being selected for reproduction is directly proportional to the fitness of that organism with respect to the fitness of all the other organisms in that population. In Figure 53 is the probability equation $p_i$ for an individual $i$ and an illustration of the population from which individuals are being selected:

$$p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}$$

**Equation 1: Probability of Selection**

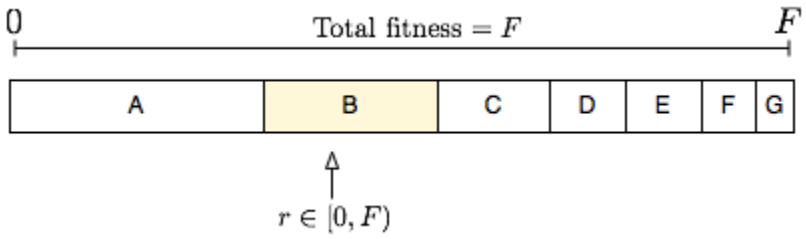


**Figure 53: Fitness Proportionate Selection**

Stochastic universal sampling is an adaptation of fitness proportionate selection except that it has one random value used as an evenly spaced interval. Rather than generating a random number per iteration, the selection method passes over the population's probability spectrum choosing each individual it lands on. It looks likeFigure 54, where $F/N$ is the interval and each diamond along the selection line is a chosen individual.

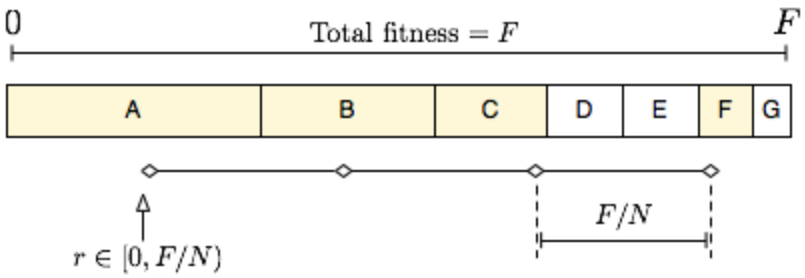


**Figure 54: Stochastic Universal Sampling**

Tournament style selection is a method where $2^n$ individuals are chosen at random and then compared to one another in rounds. The winner of each round goes on to the next round. A final individual is the champion and is used for crossover and mutation when producing the next generation. This can be illustrated using a traditional tournament bracket; the first round is on the outer-most edges and the champion finally arrives in the middle, as in Figure 55.

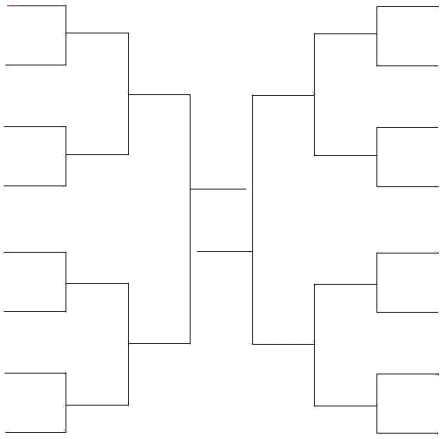

**Figure 55: Tournament Selection**

The last method of "artificial natural selection" to be implemented is truncation selection and is actually quite simple. The stress mechanism has a cut-off point, typically a percentage. Any individual whose fitness score does not meet or surpass the truncation threshold is—not surprisingly—truncated. All of the remaining individuals is rotated through in descending order of fitness scores for selection during the breeding process.

### 4.11.5 Artificial Neural Networks

The implementation of the artificial neural networks is actually quite simple. The graph consists of a Brain, which contains Clusters—or layers—of Neurons. The Clusters connect Neurons to

one another through a Synapse—similar to an edge in a graph. Each Synapse is accompanied by a weight, used later in the backpropagation algorithm. There is an input Cluster, a variable number of hidden Clusters and finally an output Cluster. In our specific implementation, there is only a single hidden cluster that contains ten neurons. The Neurons propagate floating point numbers to indicate each input and output values for each neuron. Each Neuron sums the input synapses, adds its own internal weight, feeds that into a sigmoid function and *that* value is what is forward propagated through the neural network into each of that Neuron's output synapses. The sigmoid function's graph looks like Figure 56.

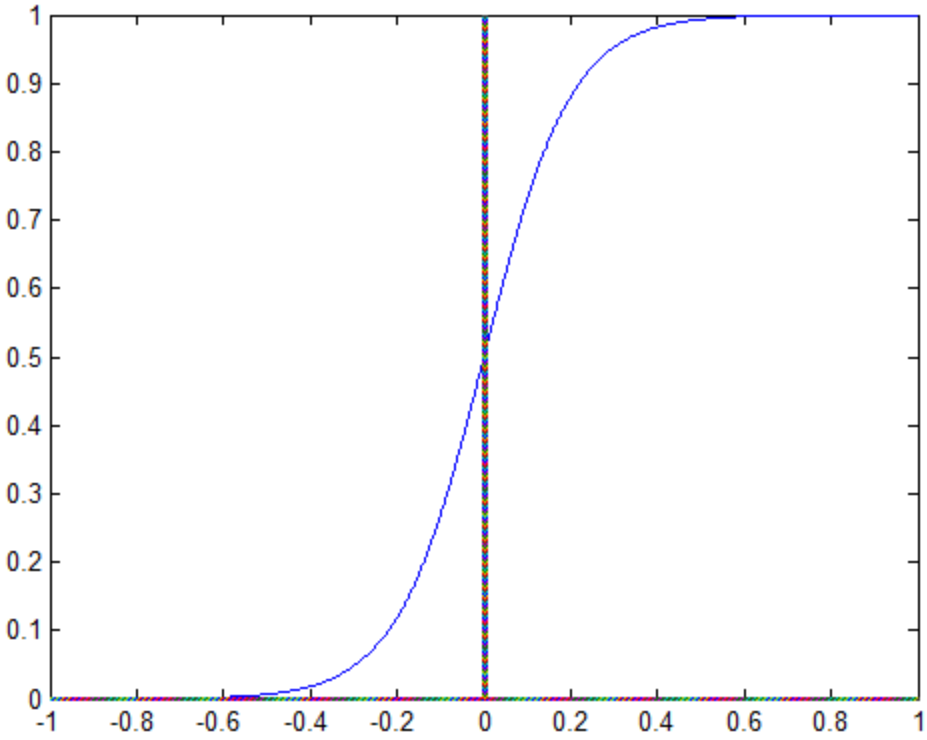


**Figure 56: Sigmoid Function**

The artificial neural network used in game has been pre-trained by a large number of expected input and output cases. The justification for using the neural network is that the relationship for mapping the inputs to the outputs is not very clear. Rather, we do know the inputs and expected outputs, but again, not the best way to map one to the other. A standard feed-forward artificial neural network trained with backpropagation is the ideal solution. The psuedocode for backpropagation:

```
Initialize the weights in the network (often randomly)
Do
       For each example e in the training set
            O = neural-net-output(network, e) ; forward pass
            T = teacher output for e
            Calculate error (T - O) at the output units
            Compute delta_wh for all weights from hidden layer to output layer
            Compute delta_wi for all weights from input layer to hidden layer
            Update the weights in the network
Until all examples classified correctly or stopping criterion satisfied
Return the network
```

The graph of error vs. training iterations looks like Figure 57.

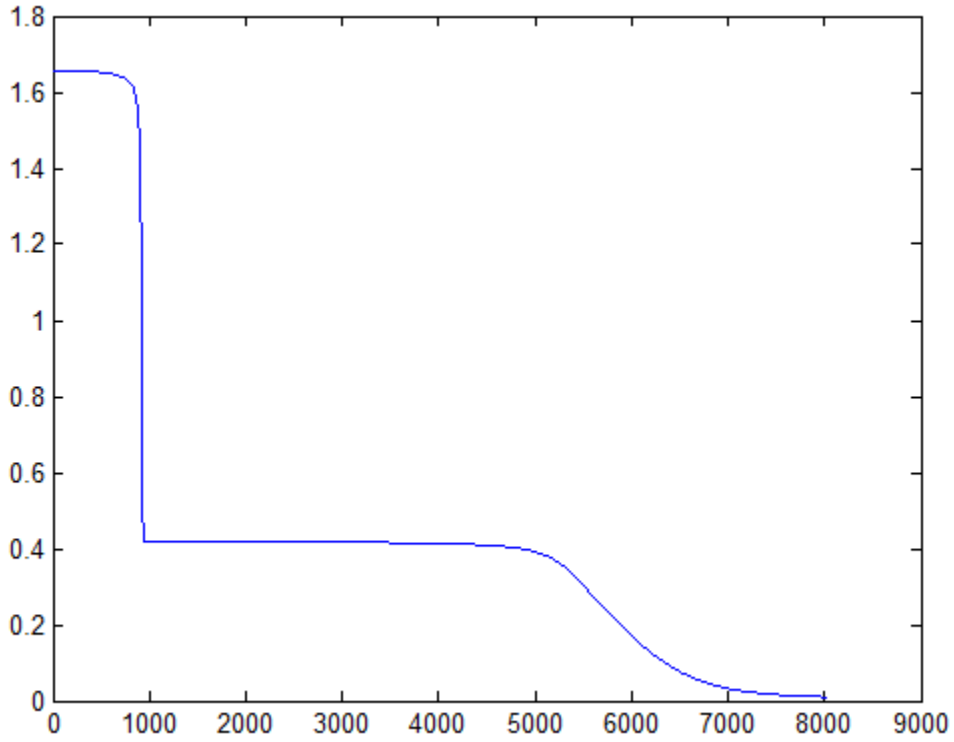


**Figure 57: Graph of error vs. training iterations**

This specific artificial neural network was trained appropriately with a < 0.01 error rate in just over 8000 training iterations. This level of accuracy is more than acceptable for the application domain.

The inputs to the artificial neural network are generated from the boss' internal representation of the game state. These are also known as Emotions. The list of Emotions includes Aggression, Anxiety, Cowardice, Desperation, Fatigue, FreeWill, Frustration, Pain, Rage and Terror. Aggression increases when the boss can perceive players. Anxiety increases as the boss moves

further from hiding spots. Cowardice increases as the number of players (and the stronger they are) within range of the boss increases. Desperation increases as the boss' subsystems become disabled. Fatigue increases as the boss' energy decreases. FreeWill is entirely random, but is bounded to the range *[0.4..0.6]*. Frustration increases as the time the boss has perceived players without being in range to attacking increases. Pain increases as the boss' health decreases. Rage increases when the boss is within range to attack a player. Lastly, Terror increases as the boss is immediately about to be attacked (especially by a projectile).

All of these inputs are fed into the input Cluster of the artificial neural network. The output cluster contains a single node, whose value is within the range *[0..1]*. This allows for a spectrum to control the state machine based on the output of this node, as shown in Figure 58.

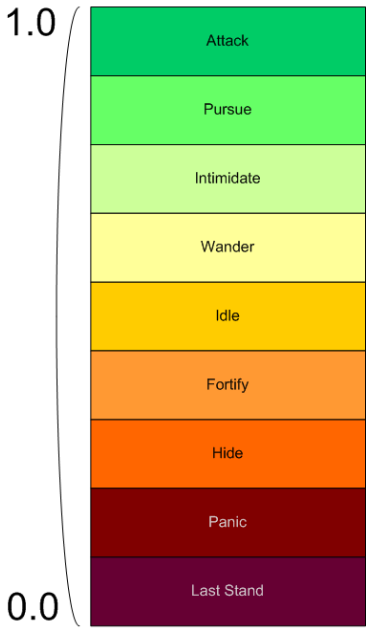


**Figure 58: Spectrum of state machine**

As mentioned earlier, the finite state machine is controlled by the artificial neural network. Based on the output spectrum, when the network is actually computed, the finite state machine either transitions to a new state, or continues in its current state. As there are nine states rather than a clean ten, fractions istter illustrate the decision process between the neural network and the finite state machine. The Last Stand state's range occurs between 0 and 1/9. However, the *training examples* for Last Stand have the expected value of 1/18, halfway between 0 and 1/9.

This gives a leeway of plus or minus 1/18 error when the neural network is actually used. Since the final error rate is less than 0.01, it is well within the bounds of a usable system.

Additionally, the spectrum has been designed in such a way that the most *offensive* states are close to 1, the most *neutral* states are close to 0.5, and the most *defensive* states are close to 0. If for some reason a calculation was outside of the 1/18 error lenience, it would only leak over to a state that is similar in nature to the desired state. If the boss is trying to Wander, but instead ends up in Idle, there is no real harm done. If the boss is trying to Pursue a player, but instead Attacks, the behavior might seem erratic, but it is not absurd, and only adds to the realism and depth of the boss.

### 4.12 Web Server

There are two separate web servers for this project. As the Photon server is running on a Windows virtual machine, it was prudent to serve the automatic updater via HTTP from this server as well. The update server is running IIS 7.0 and simply allows the downloading of files. A directory at http://obsidian.codeartlife.com/update contains the full listing of game files. The updater references the timestamp of each file in the web directory against the local directory, and downloads any new files.

The second web server is a LAMP stack that contains the forums and a MySQL rdbms. The forum software is the SimpleMachines package. The Photon server connects to the MySQL server so that it can integrate the Obsidian accounts to the SimpleMachines accounts. Additionally, some new tables have been created in the MySQL instance to store persistent player account information including skills and win/loss records.

# 5 Testing

This section covers the play tests run to find any issues still in the game. Included are the documents used in the play-tests, the testing process, and the results from these tests.

During the play-test, volunteers were asked to play the game and then fill out two questionnaires related to general game issues and combat respectively.

## 5.1 Testers

The IMGD lab in Fuller 222 was reserved for two play-testing sessions. E-mails were sent out to all IMGD majors and they were asked to join in play testing our game. The first play-testing session included fifteen volunteers who played and filled out the forms. From these documents we derived a great deal of fixes for the game. While the second play-test was unsuccessful due to server related issues, those issues did allow us to solve some server issues that were plaguing the game.

## 5.2 Results

Despite the test being done with an early alpha version of the game, we derived many useful changes to the game. The results were broken down into animations, camera work, controls, environment, and combat/gameplay.

For animations, testers noticed that many of the animations had clipping issues with the terrain. For instance, when the enemy is killed, if he is near a ledge he simply falls through it. Further, the spear animation seemed to have many issues. Testers who quickly clicked with the spear had their characters spear throwing animation refuse to play. Further, during an idle animation, the spear would poke through the terrain.

The camera was generally well-liked by the users. They thought it was smooth and commented that the ability to zoom in and out was very useful. One tester expressed a desire to have the camera be defaulted to completely zoomed out.

Testers also commented on our control scheme. The most often mentioned desire was for our roll ability to roll the character in the direction they are moving, as opposed to which way the camera is facing. This stemmed from a desire to be able to roll away from the boss without moving the camera to another direction. Further, one tester wrote that if the player is moving backwards, the roll shouldn't be available.

As for issues with the environment, testers quickly found many holes in the invisible walls set up to prevent players from leaving the game area. Further, testers were able to get their characters stuck in certain areas across the map. We marked these spots down, and the areas were fixed in further iterations of the game.

The spear was the most popular weapon by far, as many testers said it was satisfying to throw the spear and have it stick into the boss. Many of them did ask for a reticle for aiming the spear, as it was difficult to judge exactly where your spear was going to land. In addition, several asked for a power meter for the spear, so players could decide exactly how hard they would throw the spear.

For player abilities, testers were interested in having a stun, a side roll, and a double jump. Several testers also mentioned in their questionnaires that player abilities, such as the jump and roll, should use stamina so as to prevent players from constantly rolling out of the enemy's range during combat.

While many of the suggestions were not fully implemented, the play testing sessions were very useful in providing a critique, and by extension, perspective of our work. The team agreed with many of the testers, and only due to time constraints were their suggestions not included in the final release of the game.

# 6 Conclusion

The *Obsidian* project team was successful in accomplishing many of its goals set in the design and implementation process, and the result is a fully functioning game.

Our design phase successfully set out a vision for our game, giving a plan of action for the art, technical, and design aspects of the development process. During this time, the artists designed and drew concepts of important art assets while the coders put together a thorough and clear plan for the server architecture and adaptive enemy artificial intelligence.

With the design finished, the team began creation of all assets, technical and artistic. During this time the environment, enemy, and players were modeled, textured, and brought into the game. The server was set up and the enemy's artificial intelligence was developed. At the end of this phase, a freeze was placed on art asset creation so the team could focus on gameplay and play testing.

The implementation of gameplay elements and the play testing of those elements was the team's final task. A skill tree, multiple weapons, and player classes were implemented to move *Obsidian* past being a simple hack and slash game. Testers provided us with valuable knowledge on ways to improve diverse aspects relating to gameplay, art, and technical bugs. Perhaps more important, the testers enjoyed their experience.

*Obsidian* is now a finished product that can be accessed through our website and played by anyone. While there are many aspects of our original design that the team desired to implement, the game stands on its own as a modern piece of game development.

# 7 References

The following were used in the design and implementation of Obsidian.

Terrain Textures derived from http://www.hourences.com/textures/.

Tutorials, scripts, and shaders from the Unity Wiki. URL: http://www.unifycommunity.com/wiki/index.php?title=Main_Page.

Unity Forums. http://forum.unity3d.com/

ZBrush forums and tutorials. http://www.zbrushcentral.com/

Foliage Textures derived from http://www.cgtextures.com/

"Spore Behavior Tree Documents" http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs

"The Artificial Intelligence of Halo 2" http://electronics.howstuffworks.com/halo2-ai.htm

References for art design:

*God of War Trilogy:* http://www.godofwar.com/

*Darksiders:* http://community.darksidersvideogame.com/age-gate/enter

*Shadow of the Colossus:* http://us.playstation.com/ps2/games/shadow_of_the_colossus/ogs/

# Appendices

## Appendix A – Assets

*Table A1 – Texture List*

| *Textures* | | |
|---|---|---|
| *Name* | *Description* | *Source* |
| *Boss* | *Diffuse, Normal, Specular* | *Digitally Painted* |
| *Player* | *Diffuse, Normal* | *Digitally Painted* |
| *Throne* | *Diffuse, Normal, Specular* | *Digitally Painted* |
| *Bridge* | *Diffuse, Normal* | *Painted, CGTextures,CrazyBump* |
| *Column* | *Diffuse, Normal* | *Painted, CGTextures, CrazyBump* |
| *Terrain – Cliff* | *Diffuse, Normal* | *Hourences* |
| *Terrain – Grass* | *Diffuse, Normal* | *CGTextures* |
| *Terrain - Rock* | *Diffuse, Normal* | *Hourences* |
| *Terrain - Dirt* | *Diffuse, Normal* | *Hourences* |
| *Grass 1* | *Diffuse* | *CGTextures* |
| *Grass 2* | *Diffuse* | *CGTextures* |
| *Grass 3* | *Diffuse* | *CGTextures* |
| *Banana Plant* | *Diffuse* | *CGTextures* |
| *Bush 1* | *Diffuse* | *CGTextures* |
| *Grass Blades* | *Diffuse* | *CGTextures* |
| *Palm* | *Diffuse* | *CGTextures* |
| *Tree 1* | *Diffuse* | *CGTextures, Unity3D* |
| *Tree 2* | *Diffuse* | *CGTextures, Unity3D* |
| *Torch* | *Diffuse, Normal, Specular* | *Digitally Painted* |
| *Throne Base* | *Diffuse, Normal* | *Digitally Painted* |
| *Mushroom1* | *Diffuse, Normal, Luminosity* | *Digitally Painted* |
| *Mushroom2* | *Diffuse, Normal, Luminosity* | *Digitally Painted* |
| *Lava 1* | *Diffuse* | *Photoshop* |
| *Lava 2* | *Diffuse* | *Photoshop* |
| *Stairs* | *Diffuse, Normal* | *Digitally Painted, CrazyBump* |
| *Sword* | *Diffuse, Normal, Luminosity* | *Digitally Painted, CrazyBump* |
| *Spear* | *Diffuse, Normal, Luminosity* | *Digitally Painted, CrazyBump* |
| *Dagger* | *Diffuse, Normal, Luminosity* | *Digitally Painted, CrazyBump* |

*Table A2 – Model List*

| *Models* | | |
|---|---|---|
| *Name* | *Description* | *Source* |
| *Boss* | *Huge beast determined to kill the player* | *Maya, ZBrush* |
| *Player* | *Tribal member fighting boss with sword, spear, and dagger* | *Maya, ZBrush* |
| *Throne* | *Obsidian throne on a base of* | *Maya, ZBrush* |

| | | |
|---|---|---|
| | *skulls for boss* | |
| *Bridge* | *Bridge for crossing lava* | *Maya* |
| *Column* | *Columns placed near bridges* | *Maya* |
| *Rock1* | *Generic Rock* | *Maya, ZBrush* |
| *Rock2* | *Generic Rock* | *Maya, ZBrush* |
| *Rock Spire* | *Generic Rock* | *Maya, ZBrush* |
| *Stairs* | *Stone slab stairs leading out of lava* | *Maya* |
| *Banana Plant* | *Banana plant* | *Maya* |
| *Bush 1* | *Red and purple big leaf* | *Maya* |
| *Grass Blades* | *Ten large grass blades* | *Maya* |
| *Palm* | *Tall tropical palm tree* | *Maya* |
| *Tree 1* | | *Tree Creator* |
| *Tree 2* | | *Tree Creator* |
| *Torch* | *Torch* | *Maya, ZBrush* |
| *Throne Base* | *Rock base that throne sits on* | *Maya, ZBrush* |
| *Mushroom* | *Mushroom* | *Maya, ZBrush* |
| *Main Lava* | *Mesh for Lava* | *Maya, 3DSMax* |
| *Lava Waterfall* | | *Maya, 3DSMax* |
| *Sword* | *Large, angular sword used by player* | *Maya* |
| *Spear* | *Long throwing spear* | *Maya* |
| *Dagger* | *Short, jagged dagger* | *Maya* |

*Table A3 – Sounds List*

| *Sounds* | | |
|---|---|---|
| *Name* | *File name* | *Source* |
| *Jungle Ambience* | *Ancient Jungle* | *AudioJungle* |
| *Lava Ambience* | *Low Rumble Various* | *AudioJungle* |
| *Ocean Ambience* | *Sea Waves Crashing* | *AudioJungle* |
| *Monster Roar* | *Monster Roar 1* | *AudioJungle* |
| *Monster Growl* | *Monster Growling* | *AudioJungle* |
| *Male Grunt 1* | *grunts_male_2* | *AudioJungle* |
| *Male Grunt 2* | *grunts_male_5* | *AudioJungle* |
| *Male Grunt 3* | *grunts_male_6* | *AudioJungle* |
| *Male Grunt 4* | *grunts_male_6* | *AudioJungle* |
| *Male Grunt 5* | *grunts_male_11* | *AudioJungle* |
| *Male Grunt 6* | *grunts_male_12* | *AudioJungle* |
| *Male Grunt 7* | *grunts_male_15* | *AudioJungle* |
| *Male Grunt 8* | *grunts_male_16* | *AudioJungle* |
| *Male Grunt 9* | *grunts_male_20* | *AudioJungle* |
| *Male Grunt 10* | *grunts_male_21* | *AudioJungle* |

## Appendix B – Testing documents

Before you play, we would like to ask you to pay attention to any of the following and answer them either during or after your time playing. Feel free to put down as much information as you like.

1. Have you experienced any unexpected behavior or “bugs” during your time playing Obsidian? Please write down any issues you experienced.

2. Are the controls intuitive? If not, what changes would you make to them?

3. Do you like how the camera feels?

4. Did the world seem believable? In your opinion, what effect if any did the environment have on the gameplay?

5. Did you notice any issues with animations for the boss? For the character? Which ones had issues, and what issues did you perceive?

6. Any other comments you have not related to these questions?

Thank you for helping our team play-test Obsidian, we here at Milk Studio greatly appreciate your time and help!

The purpose of this play-test is to focus on the combat of our game Obsidian. To that end, we'd like to ask you a few questions about your experience playing our game. Feel free to put down as much information as you like.

1. When you attack the boss, do you feel that you are actually causing him harm? If not, what are you looking for when your character is striking the boss?

2. Does the boss appear to be intelligently fighting you? If not, what actions are the boss making that seem to be out of the ordinary?

3. Do you feel that combat is balanced between the players and the boss?

4. Was the environment conducive to fighting the boss? If not, in what ways was it not conducive, and what do you think should be changed to fix that issue?

5. Finally, are there any abilities you wish your character had?

6. Any other comments you have not related to these questions?

Thank you for helping our team play-test Obsidian, we here at Milk Studio greatly appreciate your time and help!