**Prototype:**

A Senior Project in Video Game Development

with An Emphasis on Combat AI

By

O’Brian Cox

CS 4395 – Senior Project Report

In partial fulfillment of a Bachelor of Science in Computer Science

Department of Computer Science and Engineering

University of Houston – Downtown

Spring 2019

Faculty Advisor:

Dr. Ling Xu

Course Instructor:

Dr. Ken Oberhoff

Table of Contents

[Special Acknowledgements 3](#_Toc7868251)

[Abstract 4](#_Toc7868252)

[1. Introduction 5](#_Toc7868253)

[2. Background 7](#_Toc7868254)

[3. About Unreal Engine 4 8](#_Toc7868255)

[3.1 Overview 8](#_Toc7868256)

[3.2 Blueprint Basics 10](#_Toc7868257)

[3.3 Animation Blend Spaces and State Machines 14](#_Toc7868258)

[4. Game Development 18](#_Toc7868259)

[4.1 Overview 18](#_Toc7868260)

[4.2 Player Character 18](#_Toc7868261)

[4.2.1 Character Model and Skeleton Rig 18](#_Toc7868262)

[4.3 Artificial Intelligence Character 19](#_Toc7868263)

[4.4 User Interface 19](#_Toc7868264)

[4.5 Level Design 21](#_Toc7868265)

[5. Conclusion and Future Works 22](#_Toc7868266)

[6. References 23](#_Toc7868267)

# Special Acknowledgements

I would like to thank Dr Ling Xu for being my advisor over the course of this semester. This project fell outside the realm of her expertise, but the input she offered was welcome and taken to heart. Several of the UI choices and sounds were pushed by her to make my project more robust than I had originally planned. However, I thoroughly enjoyed the freedom to oversee the core design of the project, and I thank her for her patience when dealing with this complex game engine. I would also like to thank Dr. Oberhoff for a similar degree of patience when dealing with the few absences I had during the semester, and for giving me the opportunity to complete this project properly.

On a more personal note, I would like to thank my wife Eli for putting up with the late nights and long hours working on this project in my never-ending attempt to reach perfection, even if it was never attainable.

Lastly, I want to acknowledge the Udemy community, which proved invaluable during the learning process. The course taught by Ben Tristram provided the groundwork for my knowledge of Unreal Engine 4.

# Abstract

Prototype is a third person sword fighting technical demonstration developed with Unreal Engine 4. The purpose of this project was to delve into the complexities of artificial intelligence using a 3-dimensional game development IDE, but technical limitations prevented an in-depth exploration of AI and genetic algorithms. Once AI development with genetic algorithms had proven to be infeasible with the timeframe given, focus of the project shifted toward a complex combat system with various mechanics on display. The player is put into an arena with three artificial intelligence enemies, placed in specific locations for demonstration of the systems and mechanics implemented. The player can explore the area freely and engage each enemy as they see fit. The AI, while not as complex as initially planned, still provides enough of a challenge to demonstrate each mechanic thoroughly. This combat system implements target-locking, simple AI systems, blocking, parrying, dodging, stamina restrictions, backstabbing, light, heavy and special attacks in addition to the beginning stages of an equipment system. The project implements complex animation systems within Unreal Engine 4, and utilizes skeleton rigging, meshes and animation blend spaces. All of the systems were implemented using Unreal Engine 4’s Blueprint programming language.

# 1. Introduction

This project was developed by a single student beginning in the Spring 2019 semester beginning in mid-January. The original idea behind the project was to explore the capabilities of implementing a genetic algorithm system into the AI of a game. Research conducted prior to the beginning of the project showed that genetic algorithms had not been developed for any major gaming title, and no peer reviewed scientific articles discussed the viability of genetic algorithms within games. Most scientific papers that focused on genetic algorithms and video games did so by developing an algorithm that would produce AI agents that could play existing games as if they were a human player. Unfortunately, this idea would have taken at least another month to develop, and the time limitations of the semester would not allow it.

What took precedence upon this discovery was to emphasize the combat system. When implementing a genetic algorithm was the priority, the combat system was basic and included light/heavy attacks, stamina and blocking. Priority shifted toward improving this system dramatically, and the result is a combat system deep enough to be expanded into a full-fledged game with the addition of a simple story and clearer objectives. The player can approach one of the three AI enemies placed throughout the level and engage them in different ways. One enemy his restricted to a raised platform, and the system for combat disengagement can be tested. Another is facing a wall, where the player can sneak up on the enemy and perform a backstab. The last enemy shows off the AI patrolling features, where they walk a specific path laid out, searching for the player.

This project explores every aspect of game development within Unreal Engine 4, in addition to Blender for some rigging and animation work. Unreal Engine 4 has specialized systems for:

* User Interfaces
* Animations and rigging
* Function handling and scripting
* Physics
* Collision detection
* Landscaping and geometry
* Level design and player control methods
* Artificial Intelligence

The result of this project is a series of technical demonstrations for these systems, and an in-depth look at how they interact with each other to create feature-rich and highly polished games. While the quality is far from professional grade, it is a clear example of how games with Unreal Engine are developed. The goal of the project was to create a technical demonstration of a complex combat system set up as the basis for a larger, more long-term development project.

# 2. Background

# 3. About Unreal Engine 4

## 3.1 Overview

Unreal Engine is a game development IDE designed by Epic Games between 1995 and 1998, initially revealed to the world with the first-person shooter game *Unreal*. Despite being developed primarily for first-person shooters, many other game genres have had success with the IDE, such as stealth, fighting, Massive Multiplayer Online Roleplaying Games (MMORPGs), and other Roleplaying Games (RPGs). While not truly open-source, Unreal Engine’s source code can be viewed and modified, but falls into the “source-available” category with Epic Games’ licensing policies. That is, unless a developer pays a premium price for the engine’s license, any games developed with Unreal Engine must comply with royalty negotiations should the finished product be sold commercially. Luckily as an academic endeavor, this project is not affected by Epic’s licensing policies and no monetary commitment was needed.

Since 1998, Unreal Engine’s source code has always been written in C++, which affords several advantages regarding portability and custom performance enhancements. Usually these advantages are more useful for larger projects than *Prototype*. In addition, if given enough time for development, the artificial intelligence systems would have been developed in C++, as it is a better environment for genetic algorithms than Unreal Engine’s Blueprint coding environment. Blueprint is a streamlined process for C++ which relies more upon visual programming instead of hard code. It provides developers a quick and responsive system for creating the various functions needed to run a game. Elements of C++ coding standards can be seen in how Blueprint implements methods, classes and other object-oriented programming staples. Artificial intelligence utilizes visual decision trees to control how AI controls and reacts to specified states. This behavior tree flow is integral to how Unreal Engine 4 interacts with AI and is the key reason why AI development in C++ would take considerably more time.

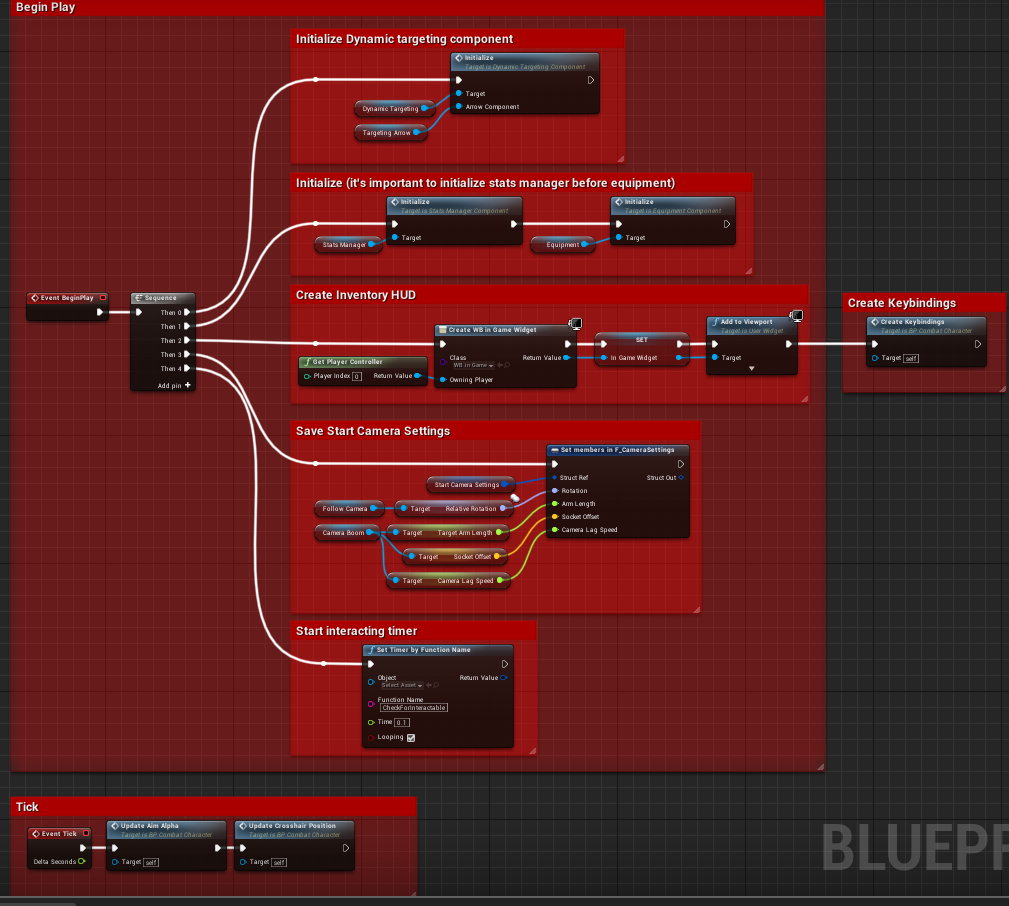
Animations are handled by a state-machine based system, and a specialized system unique to Unreal Engine called animation blend spaces. The state-machine for Unreal Engine 4 works like any standard state-machine, switching between states based of predetermined conditions, which trigger the change in state when met. Blend spaces control the way that the skeleton moves within the game world and are used to specifically smooth the transition between animations. Without blend spaces, a stationary character might animate as if instantly sprinting instead of a more natural transition of increasing velocity until the proper sprinting gait is achieved. These two systems work in tandem to handle most of the work that Unreal Engine does.

Another aspect to development with Unreal Engine is the built-in physics system. Most models – called meshes in Unreal Engine – have a toggle-able checkbox that makes the engine simulate physics, with a few options to add weight to the object. However, as research into the development process of Unreal Engine progressed, it became clearer that the simulated physics component of the engine was not always required and would sometimes cause more problems than it would solve. This physics system also handles collision and suffers the same setbacks. To keep both the physics and collision in this game accurate and bug-free, most resources suggest directly implementing methods of handling hit detection.

There is a powerful particle effects system within Unreal Engine, in addition to a sound event system capable of not only emission point sound cues but also layered audio filtering. Both systems are explored on a surface level for this project, though are featured with enough prominence to mention.

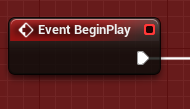
## 3.2 Blueprint Basics

The basis behind Blueprint programming in Unreal Engine is to provide a visual interface for developers to quickly create new assets from existing groundwork developed in the initial stages of production. An enemy type can be created in Blueprint and any instance of this enemy type can be generated based on the blueprint created. Additionally, new enemy types can be created by using existing blueprints, streamlining the process for much larger projects. Blueprint is not exclusive to enemies and is used for nearly everything that one might find in an entirely C++ based program. *Figure 1* provides an overview of how Blueprint is used within Unreal Engine 4.



*Figure 1*. A simple overview of a Blueprint graph.

Each square shown in *Figure 1* is called a **node** in Unreal Engine 4’s Blueprints. Each node represents a different kind of action that the compiler performs when a blueprint’s function is called. *Figure 2* shows a node colored red, with a white arrow that represents the compiler’s execution route – referred to as an execution **pin** in Unreal Engine 4’s Blueprints. This node is called an event execution node.



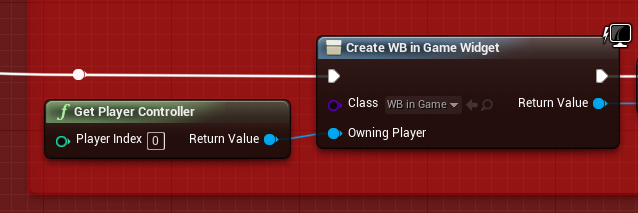
*Figure 2*. An event execution node.

*Figure 3* shows a white colored node that can represent several different basic C++ control flow statements. Pictured is a sequence node, which is functionally equivalent to a switch statement. Though most usage for the sequence node has been without breaks, and used more for organizing what can quickly become messy node layouts. Other white nodes include for loops, while loops, and if statements (called branches)



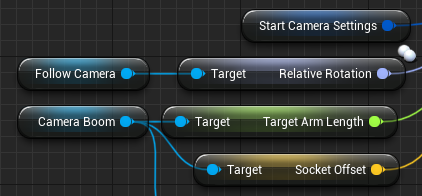
*Figure 3*. A sequence node. Note that more cases can be added as new pins.

Not all pins are execution pins. *Figure 4* shows two different types of function calls. The green node is a function that can be called without an execution pin, and a blue function node requires an execution pin to be called.



*Figure 4*. Two different kinds of function calls within Blueprint.

Starting from the execution pin, the blue node calls a function named *Create WB in Game Widget*, a built-in function for Unreal Engine’s UI system. The first action of this node is to set the value of the class to “WB in Game”. The next action of this node is to set the value of “Owning Player”, which is when the next function is called. *Get Player Controller* is another built-in function which returns a value of the player’s controlled character. Finally, *Create WB in Game Widget* finishes its execution and returns the value of the constructed widget. This structure of execution and variable handling is an extremely basic layout of how the entire Blueprint system works in Unreal Engine 4. Variables are handled in a similar way, as shown in *Figure 5*.



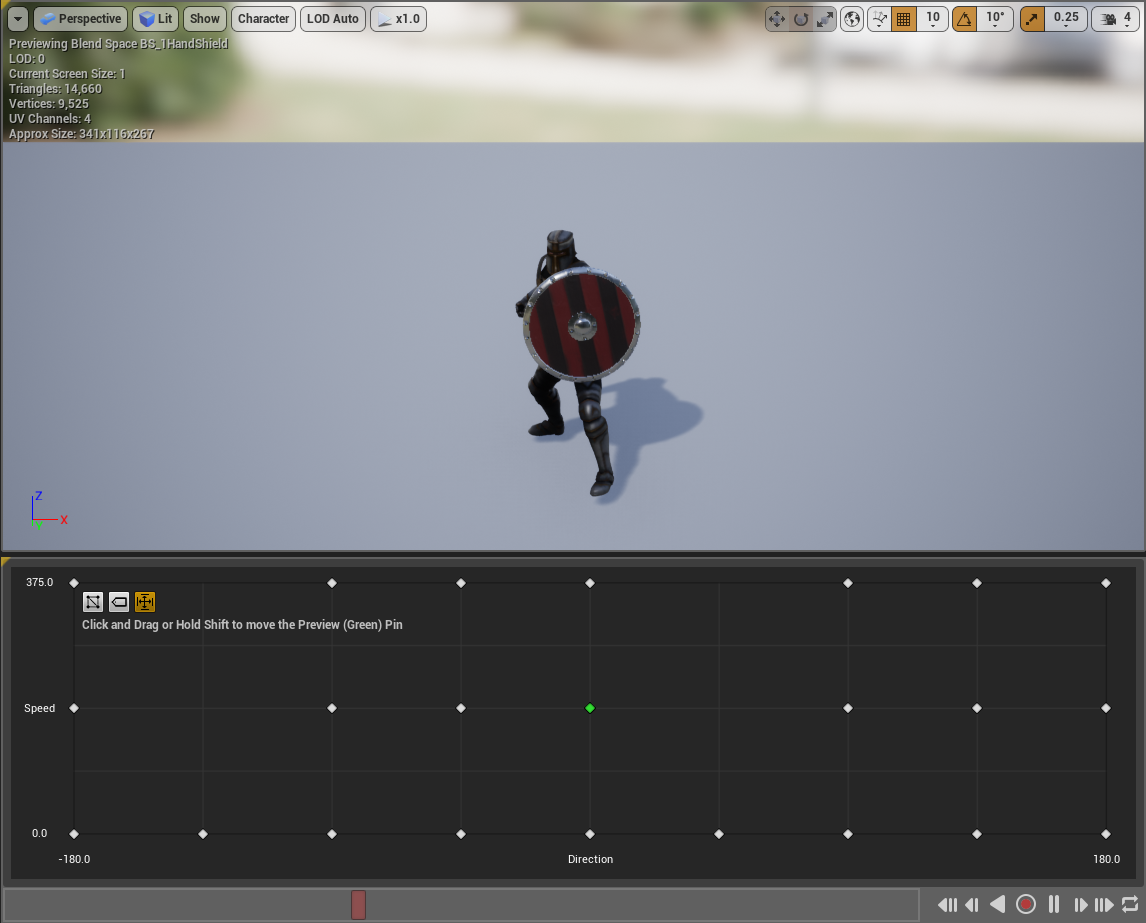
*Figure 5*. A handful of variable nodes.

In *Figure 5*, five different types of variables are shown. Light blue nodes are objects within the project. These can be anything from meshes to cameras. The dark blue node is a struct, while the light purple node (“Relative Rotation”) is a rotator variable. The green node is a float, and the yellow node is a vector. Other possible variables are Booleans, integers, bytes, and strings to name just a few. Additionally, any variable can be specified as an array or a set. Each variable as a pin associated with its call.

Even this in-depth look at how Blueprints work is only just barely scratching the surface and learning this method of development took a significant amount of time. Blueprint is a powerful tool for Unreal Engine developers, and it is crucial to learn for any developer looking to work within the IDE.

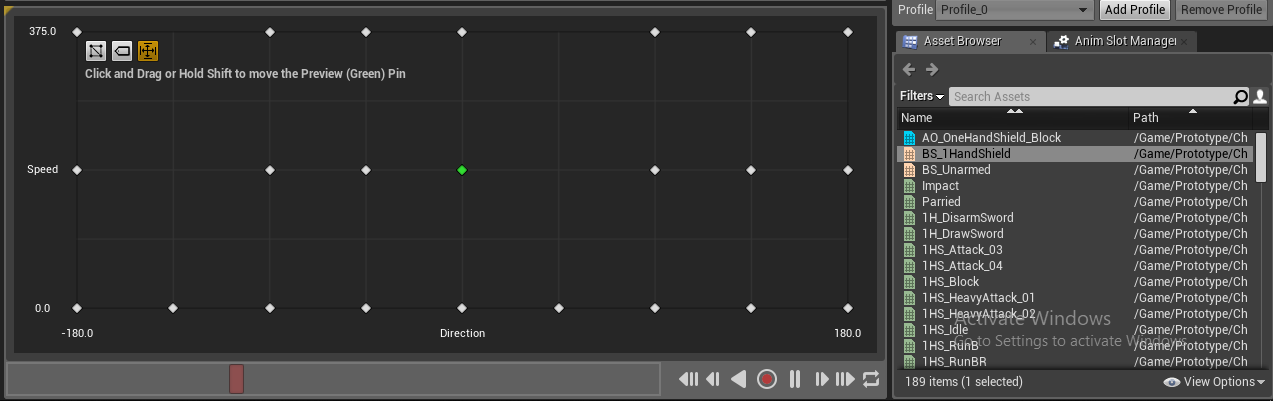
## 3.3 Animation Blend Spaces and State Machines

*Figure 6* shows the two main windows used during the development of blend spaces.



*Figure 6*. An overview of the blend space environment.

The upper half of *Figure 6* is the viewport, where the various animations for the character model can be previewed while working. The lower half is the meat of the blend space. Each of the white dots represents a different animation. *Figure 7* gives a closer look.

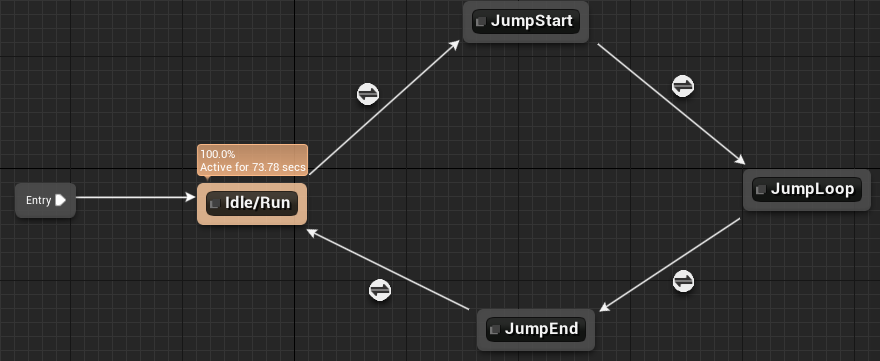


*Figure 7*. One of the two blend spaces within this project.

To the right, there is a list of animations available to the character’s skeleton rig. For the blend space in *Figure 7*, the x-axis of the grid represents which direction the character is moving relative to which direction it is facing. A value of 0 for the x-axis represents forward movement, while -+ 180 is backward movement. Positive 90 is right movement, while negative 90 is left. The y-axis represents the speed at which the character is moving. Unreal uses a standard unit measurement where 1 unit in the Unreal Engine (uu) represents 1 cm in real life. For movement speed, a value of 0 uu translates to 0 cm/s, while the character’s maximum speed in this case is 375 uu, or 375 cm/s. Every blend space must have the axis set manually by the developer and is not handled automatically by the engine.

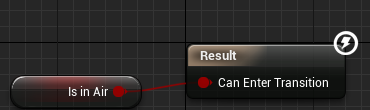
Unreal Engine processes each of the white dots as a position on the x and y plane and associates an animation with each dot. The engine then ensures that interpolated values are blended smoothly between each specified animation. The only way that this process works is by ensuring that the character’s movement is not instantly set. For example, to move forward, the W key must be pressed. However, a keyboard does not posses pressure sensitivity and functions more like an on/off switch. So, instead of setting the character’s movement speed from 0 to 375 when W is pressed, the character’s speed must be gradually increased while W remains pressed. This is vital to a blend space’s functionality.

In addition to blend spaces, Unreal Engine utilizes animation state machines to shift between different animation spaces. *Figure 8* shows a basic jumping state machine, where the base state is an idle or running state.



*Figure 8*. Movement state machine with jump states attached.

This state machine will execute the Idle/Run state until specific conditions are met, in which it will transfer to the JumpStart state until different conditions are met. In this case, it is simply a timer that lasts until the initial jump animation finishes, which initiates a falling loop until the character lands on the ground, shifting all the way back to the Idle/Run state. *Figure 9* shows the conditions that need to be met in order to trigger a state change from Idle/Run to JumpStart.



*Figure 9*. The condition that needs to be met before the state can be changed.

“Is In Air” is a bool which must be true in order to allow the state machine to transition. The engine is constantly checking to see if this condition ever changes. Each state is associated with a different animation. Some states are simple animations, others are blend spaces.

# 4. Game Development

## 4.1 Overview

Development for *Prototype* took place over the course of 3 months. At least a month of development time was spent on tutorials and research in preparation for creating the final project. Most of the tutorial projects did not make it into the final project.

[UPDATE]In total, there are 131 animations within the project, 6 animation blend spaces, 72 blueprints, 20 UI widgets, 4 data tables, 18 enumerations, 12 structs, 1 AI behavior tree, 23 meshes, 15 materials, textures, 22 sound effects, and 1 visual effect.

## 4.2 Player Character

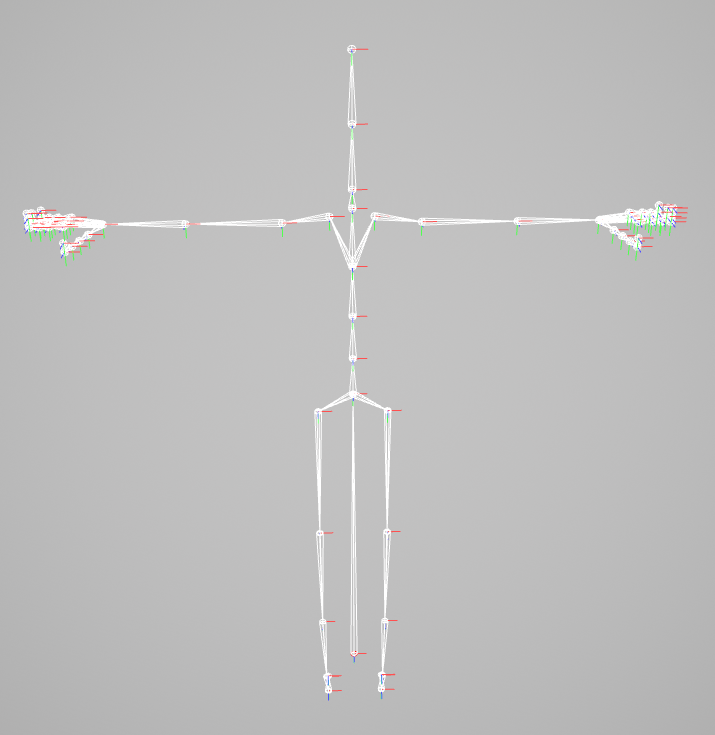
### 4.2.1 Character Model and Skeleton Rig

Since creating a 3D character and skeleton rig from scratch takes a significant amount of time and could take up an entire project on its own, a pre-existing character model was downloaded from Adobe’s Mixamo website, which hosts dozens of models that are free for non-commercial use. Each model has a Mixamo-specific skeleton rig which is compatible with any and all animations available on the website. The main setback with using Mixamo rigs and animations is the lack of root motion. Root motion allows the animation to play while the character is stationary within the game world. For example, when running without root motion, the player’s camera would remain stationary while the model would move forward. When the animation finished, the character model would teleport back to its starting position. *Figure 10* shows the player character’s model, downloaded from Mixamo.

*Figure 10*. Front and back views of the character model.

*Figure 11* shows the skeleton rig, which handles all the animation movement within the character mesh.



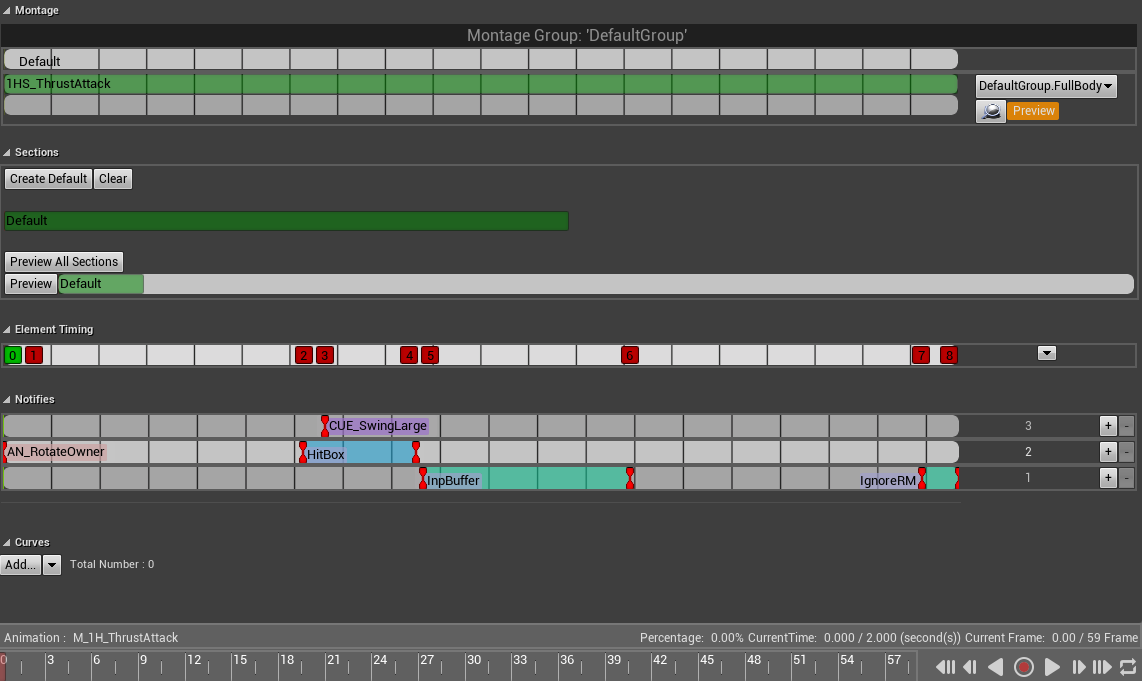
*Figure 11*. The skeleton rig for the character model in *Figure 10*.

The long bone between the leg bones starting at the hip represents root motion in the skeleton rig. Each of the red markers indicate the start-point for each bone.

### 4.2.2 Animations, Blend Spaces, and Montages

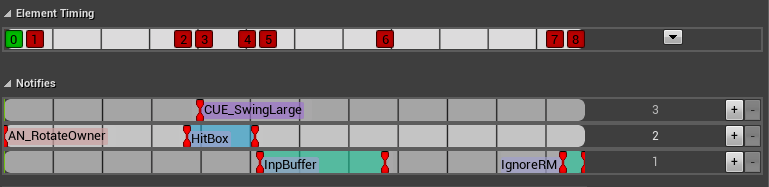
Every animation within *Prototype* was pulled from Mixamo to be used with the character model and skeleton rig as defined in the previous section. Most movement animations, such as running and walking, are placed into blend spaces. State Machines handle the transition between blend spaces, as discussed in section 3.3.

Montages are a subset of animations that include one or more base animations played sequentially. The main usage of montages is to utilize Unreal Engine’s Animation Notify system. Notifies send messages to specified blueprints that activate certain functions. *Figure 12* shows the animation montage for the player’s thrust attack ability “M\_1H\_ThrustAttack”.



*Figure 12*. The animation montage development screen.

The most useful section, the notifies, is zoomed in for *Figure 13*.



*Figure 13*. The animation notifies section of the montage development screen.

Each numbered square on the Element Timing strip represents a message being sent to blueprints and is defined by a separate blueprint that handles the message. Each of the numbers corresponds to a notify in the notifies section below. For this montage, 0 is the start of the animation. 1 is called immediately, “AN\_RotateOwner” allows the animation to rotate along with the player’s directional input. 2 calls the start of “HitBox”, which activates collision for the weapon that is equipped to the character. 3 calls the start of “CUE\_SwingLarge”, which is a sound effect for the attack. 4 calls the end of “HitBox”, which disables collision for the weapon. 5 calls the start of “InpBuffer”, which signals to the blueprint to open the input buffer. The input buffer will be discussed further in the next section. 6 calls the end of “InpBuffer”, which closes the input buffer. 7 and 8 call the start and end of “IgnoreRM” respectively, which first deactivates and then activates root motion, which helps smooth the transition between montages and blend spaces.

Each notify in the “M\_1H\_ThrustAttack” montage can be called in any other montage, which provides the groundwork for developing the animation system of the project alongside blend spaces.

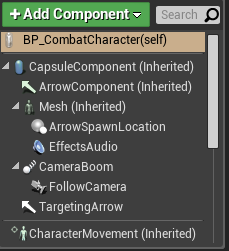
### 4.2.2 Main Character Blueprint Class

Unreal Engine Blueprints have several different hierarchies built-in. The hierarchy for a character blueprint is as follows: Actor > Pawn > Character. That is, all characters are pawns, and all pawns are actors, but not all actors are pawns, nor are all pawns characters.

Unreal Engine defines an Actor as an object that can be placed or spawned in the world. A Pawn is an Actor that can be ‘possessed’ and receive input from a controller. A Character is a type of Pawn that includes the ability to walk around. In Unreal Engine Blueprints, a controller is an actor responsible for controlling a pawn. A controller can be either a Player Controller or an AI Controller. Naturally, for the Player Character blueprint, it is a Character that is controlled by a Player Controller.

The key for development of this character blueprint is a series of Actor Components. An Actor Component is a class that defines reusable behavior that can be added to different types of Actors. This is important to allow components to be reused between both the player character’s blueprint and the AI character’s blueprint. It also allows the distinction of adding specific components to the player while excluding them from the AI, reducing system resource needs.

A Character consists of several different components. *Figure 12* shows the list of components attached to the player’s character blueprint.



*Figure 12*. A list of specific types of components attached to a Character.

Components labelled as (Inherited) are components that are assigned to the Character parent class. Any new blueprints based off the Character class will have these components attached to the blueprint automatically. CapsuleComponent handles basic collision and is what keeps the character from falling through the level or walking through walls. Mesh is the character model shown in *Figure 10*, while ArrowComponent is an editor-only visible representation of which direction the Capsule is facing, allowing the proper alignment of the character’s mesh. CharacterMovement handles movement logic for the associated Character and supports various movement modes including walking, falling, swimming, flying, and any custom modes added later.

which allows the player to enter multiple attack or movement commands while this timeframe of the montage plays without losing their desired actions

## 4.3 Artificial Intelligence Character

## 4.4 User Interface

The User Interface was originally utilizing an *Elder Scrolls: Skyrim* approach to health and stamina displays. However, as development progressed and the combat system shifted more toward a *Dark Souls* style of combat, it was prudent to adopt a similar style for the UI. *Figure 10* shows the positioning of the player’s health and stamina bars.



*Figure 10*. The player’s UI displaying health and stamina in the top left corner.

A standard color denomination of red for health and green for stamina was used. In addition to player health information, the UI also displays enemy health bars when targeted and a white dot centered on the target to indicate that the player is locked-on. *Figure 11* shows these two attributes in the display window.



*Figure 11*. Health bar of the enemy and the lock-on reticule.

As seen in *Dark Souls*, enemy health bars are displayed above their heads and follow them as they maneuver around the player.

## 4.5 Level Design

# 5. Conclusion and Future Works

# 6. References