**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 4](#_Toc8085563)

[Abstract 5](#_Toc8085564)

[1. Introduction 6](#_Toc8085565)

[2. Background 8](#_Toc8085566)

[3. About Unreal Engine 4 9](#_Toc8085567)

[3.1 Overview 9](#_Toc8085568)

[3.2 Blueprint Basics 10](#_Toc8085569)

[3.3 Animation Blend Spaces and State Machines 13](#_Toc8085570)

[4. Game Development 16](#_Toc8085571)

[4.1 Overview 16](#_Toc8085572)

[4.2 Player Character 16](#_Toc8085573)

[4.2.1 Character Model and Skeleton Rig 16](#_Toc8085574)

[4.2.2 Animations, Blend Spaces, and Montages 17](#_Toc8085575)

[4.2.3 Main Character Blueprint Class 19](#_Toc8085576)

[4.2.4 Actor Component Classes and Blueprint 20](#_Toc8085577)

[4.2.5 Player Character Capabilities and Controls 27](#_Toc8085578)

[4.3 Artificial Intelligence Character 28](#_Toc8085579)

[4.3.1 Overview 28](#_Toc8085580)

[4.3.2 AI Character Blueprint Functions and Graphs 29](#_Toc8085581)

[4.3.2 AI Controller Blueprint 30](#_Toc8085582)

[4.3.3 AI Exclusive Blueprints 30](#_Toc8085583)

[4.3.4 AI Decision Tree 31](#_Toc8085584)

[4.3.5 AI Behavior Tree Nodes 34](#_Toc8085585)

[4.3.6 AI Character Capabilities 35](#_Toc8085586)

[4.4 User Interface 36](#_Toc8085587)

[4.5 Level Design 37](#_Toc8085588)

[5. Conclusion and Future Works 38](#_Toc8085589)

[6. References 39](#_Toc8085590)

# 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.

The style of combat in *Prototype* is inspired by From Software’s *Dark Souls* titles. *Figure 1* shows a screenshot of the game.



*Figure 1*. A screenshot of From Software’s *Dark Souls* game.

Many of the combat elements found in *Prototype* are attempts at reproduction of *Dark Souls’* combat mechanics. UI elements are similarly inspired by the design choices made by From Software.

# 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.

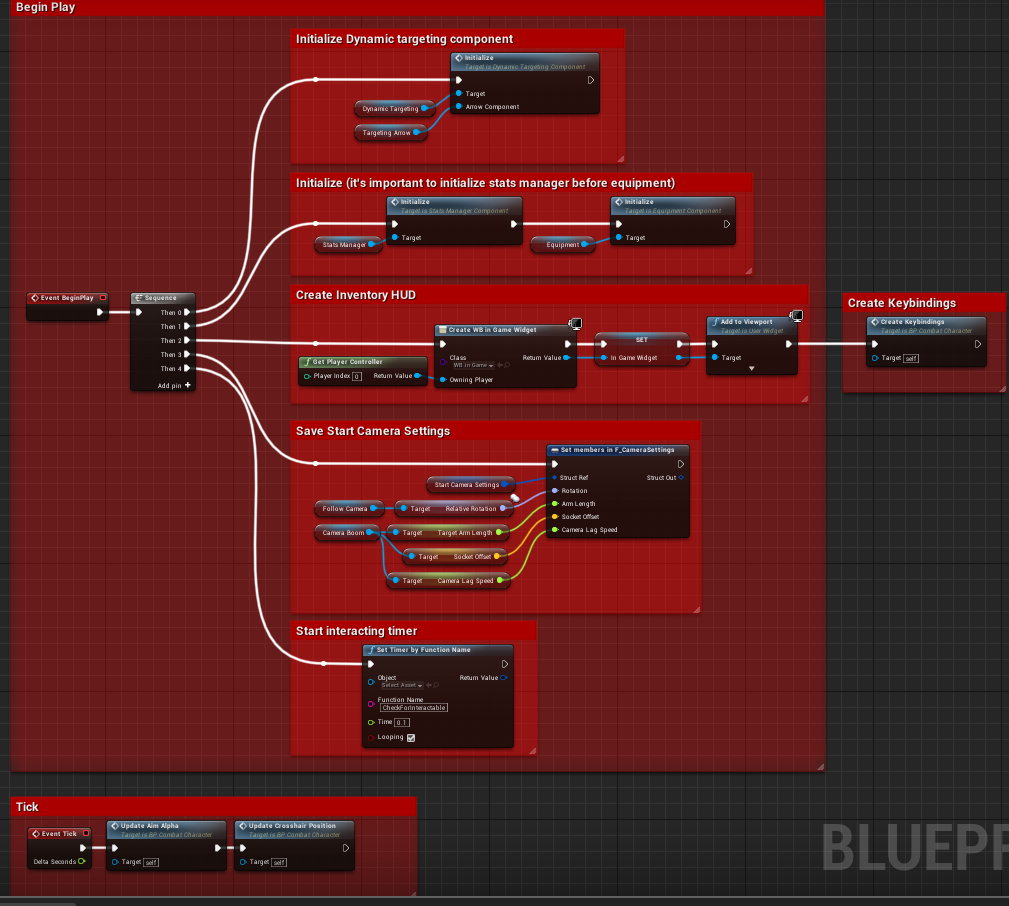
Animations are handled by a state-machine based system, and a specialized system unique to Unreal Engine called animation blend spaces. The state-machine for Unreal Engine 4 works like any standard state-machine, switching between states based of predetermined conditions, which trigger the change in 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.

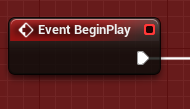
## 3.2 Blueprint Basics

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 2* provides an overview of how Blueprint is used within Unreal Engine 4.



*Figure 2*. A simple overview of a Blueprint graph.

Each square shown in *Figure 2* 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 3* 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.



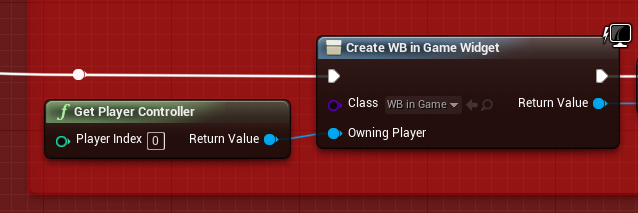
*Figure 3*. An event execution node.

*Figure 4* 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).



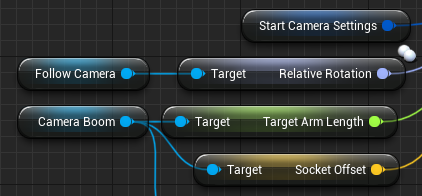
*Figure 4*. A sequence node. Note that more cases can be added as new pins.

Not all pins are execution pins. *Figure 5* 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 5*. 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 6*.



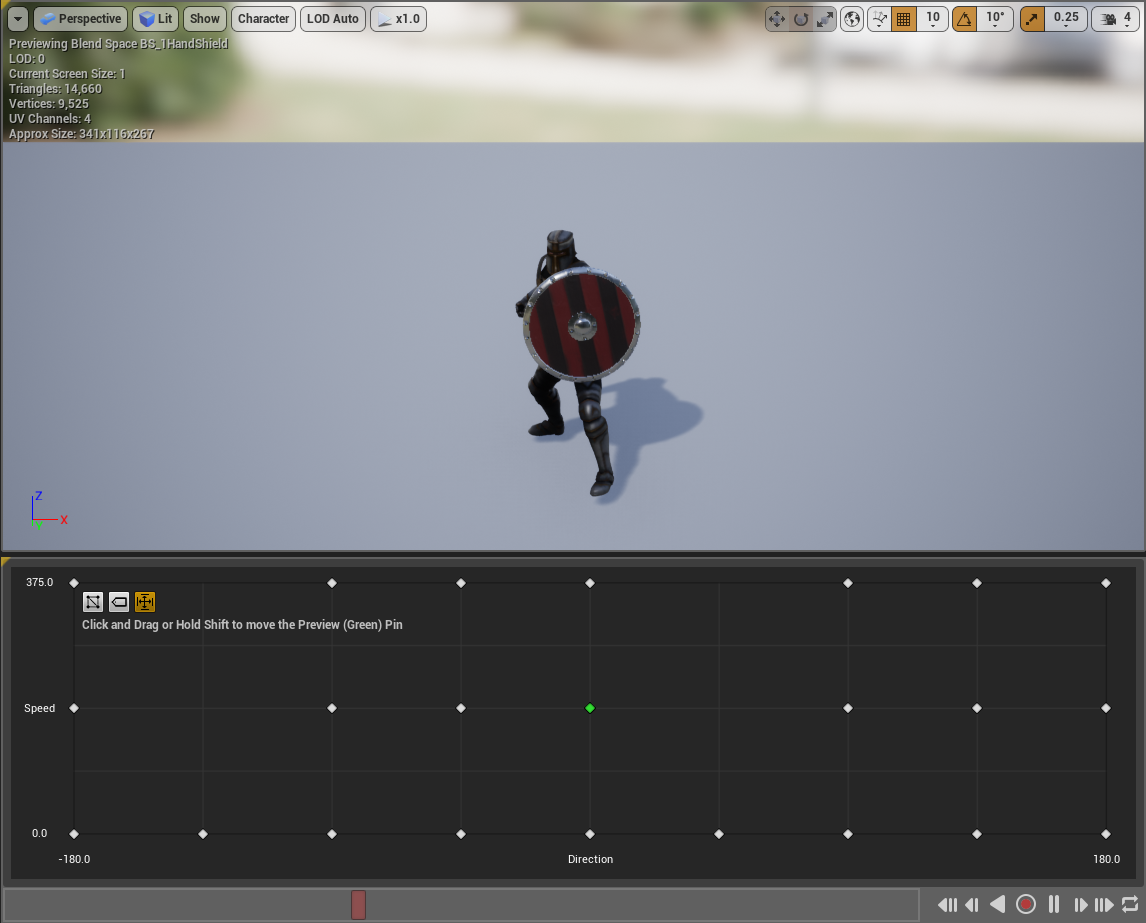
*Figure 6*. A handful of variable nodes.

In *Figure 6*, 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.

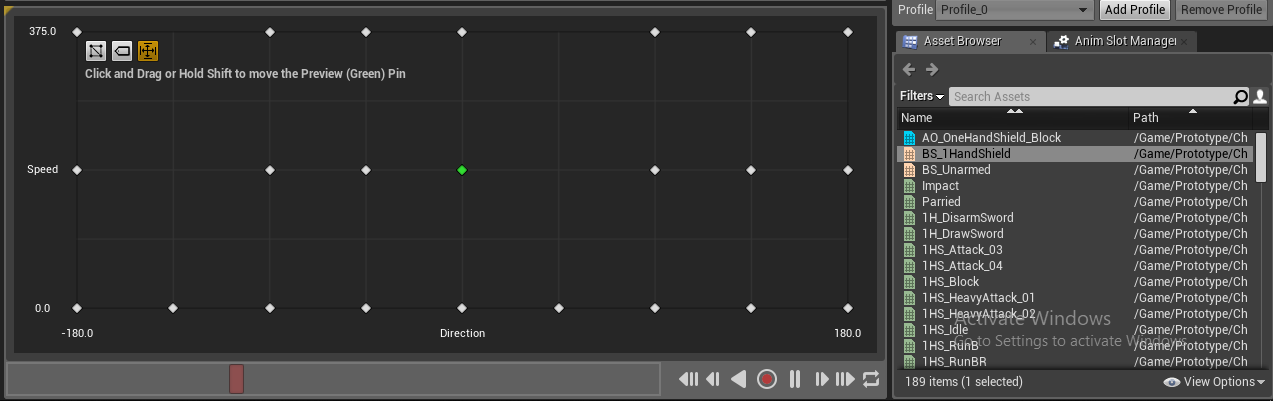
## 3.3 Animation Blend Spaces and State Machines

*Figure 7* shows the two main windows used during the development of blend spaces.



*Figure 7*. An overview of the blend space environment.

The upper half of *Figure 7* 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 8* gives a closer look.

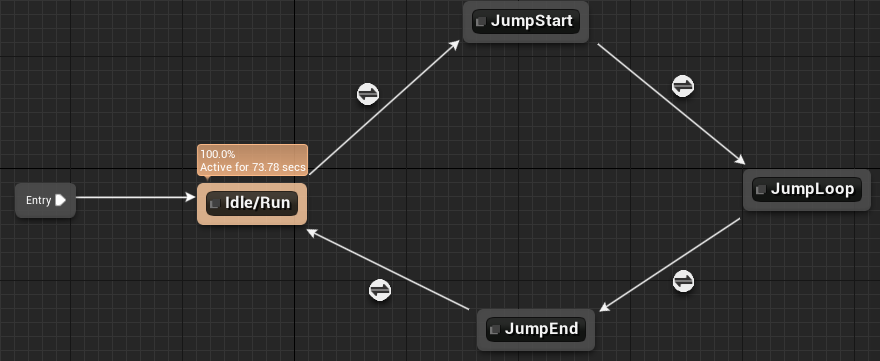


*Figure 8*. One of the two blend spaces within this project.

To the right, there is a list of animations available to the character’s skeleton rig. For the blend space in *Figure 8*, 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.

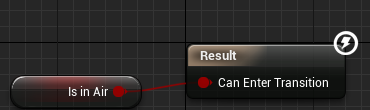
Unreal Engine processes each of the white dots as a position on the x and y plane and associates an animation with each dot. The engine then ensures that interpolated values are blended smoothly between each specified animation. The only way that this process works is by ensuring that the character’s 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 9* shows a basic jumping state machine, where the base state is an idle or running state.



*Figure 9*. 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 10* shows the conditions that need to be met in order to trigger a state change from Idle/Run to JumpStart.



*Figure 10*. 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

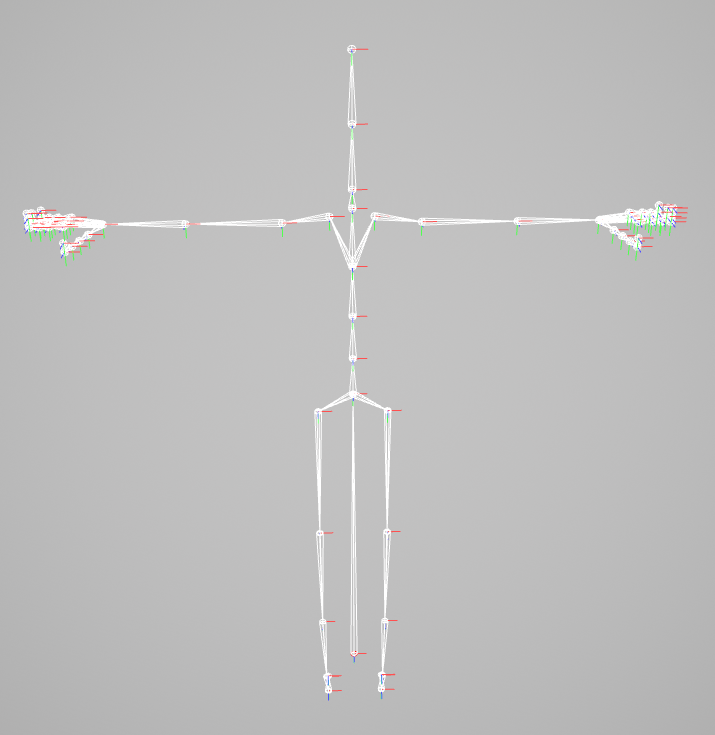
### 4.2.1 Character Model and Skeleton Rig

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 11* shows the player character’s model, downloaded from Mixamo.

*Figure 11*. Front and back views of the character model.

*Figure 12* shows the skeleton rig, which handles all the animation movement within the character mesh.



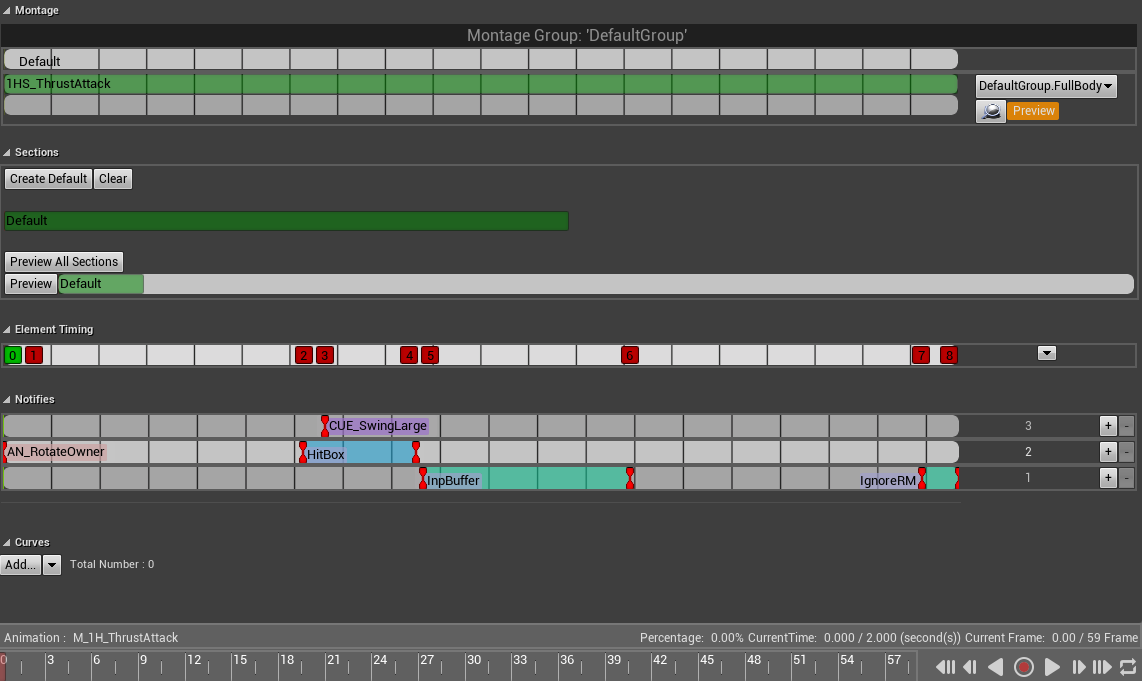
*Figure 12*. The skeleton rig for the character model in *Figure 10*.

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

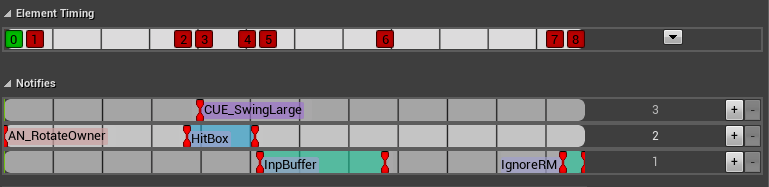
Every animation within *Prototype* was pulled from Mixamo to be used with the character model and skeleton rig as defined in the previous section. Most movement animations, such as running and walking, are placed into blend spaces. State Machines handle the transition between blend spaces, as discussed 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 13* shows the animation montage for the player’s thrust attack ability “M\_1H\_ThrustAttack”.



*Figure 13*. The animation montage development screen.

The most useful section, the notifies, is zoomed in for *Figure 14*.



*Figure 14*. 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.

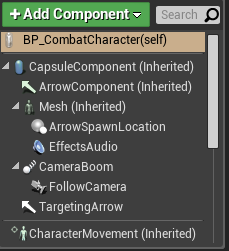
### 4.2.3 Main Character Blueprint Class

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 15* shows the list of components attached to the player’s character blueprint.



*Figure 15*. 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 11*, 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.

### 4.2.4 Actor Component Classes and Blueprint

In total, there are 15 actor components that can be added or removed from any character blueprint created for this project. For the player character, 14 were used. *Table 1* shows the list of the actor components used, the member functions associated with each component, and their purpose summarized.

*Table 1*. Actor Component Class table.

|  |  |  |
| --- | --- | --- |
| Component  Name (Type) | Member Functions | Purpose |
| StateManager  (State Manager) |  | Manages the Character’s state, switching between idle and predefined activities. Functionally similar to a state machine and is used for animations and combat. |
| InputBuffer  (Input Buffer) |  | Provides the Character a buffer between input actions and the various timers in place. This ensures that a Character that wants to block immediately after attacking, before the montage finishes, does not get locked into the wrong state. |

*Table 1* *cont.*

|  |  |  |
| --- | --- | --- |
| Component  Name (Type) | Member Functions | Purpose |
| MeleeCollisionHandler  (Collision Handler) |  | Controls the collision settings for the character’s equipped weapons. Turns on collision during specific points in attack movements and prevents multiple collisions per swing. |
| ExtendedHealth  (Extended Stat) |  | This component takes a stat associated with a character and defines additional functionality based on the stat’s needs. For this component, the extended stat is Health, and thus the component controls whether the Character’s health can regenerate or not, how much health the Character has left, etc. |
| ExtendedStamina  (Extended Stat) | *See (ExtendedHealth)* | Contains all the same functions as ExtendedHealth, but for the Character’s stamina stat. |
| MontagesManager  (Montage Manager) |  | Handles the Character’s montages. Plays animations based on input actions and ensures a Character model does not get stuck executing an animation. |

*Table 1* *cont.*

|  |  |  |
| --- | --- | --- |
| Component  Name (Type) | Member Functions | Purpose |
| DynamicTargeting  (Dynamic Targeting) |  | This component allows the Character to lock onto enemies inside the level. It handles the angle that the camera faces, and searches for the target to be locked when target lock is toggled. |
| Effects  (Effects) |  | This component applies effects to specified actions that the player takes, in addition to various stun effect. If the player is hit and stunned, this component handles the application. Backstabs are also implemented through this component. |

*Table 1 cont.*

|  |  |  |
| --- | --- | --- |
| Component  Name (Type) | Member Functions | Purpose |
| MovementSpeed  (Movement Speed) |  | Controls movement speed at a more customized level than the default functions available to Character Movement. |
| StatsManager  (Stats Manager) |  | Manages the stats of the Character. Stats include damage, modifiers such as backstab damage and number of times the Character can be hit before being stunned, to name a few. |

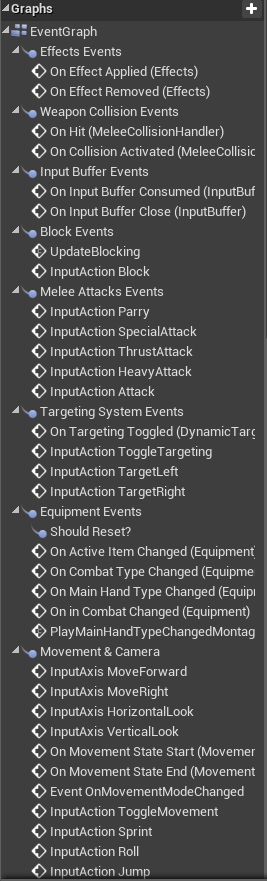
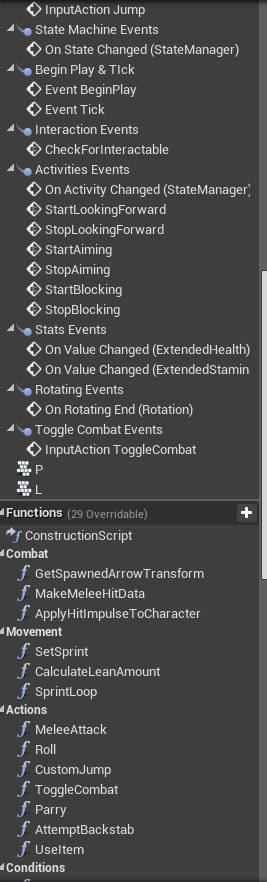
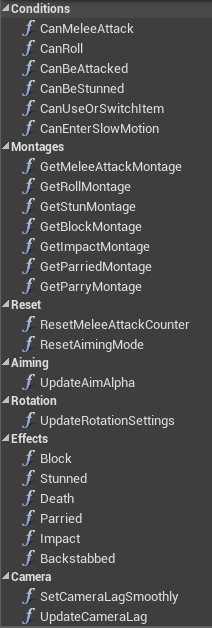
*Table 1 cont.*

|  |  |  |
| --- | --- | --- |
| Component  Name (Type) | Member Functions | Purpose |
| Dissolve  (Dissolve) |  | This component handles the dissolve effect that plays when a Character dies. It applies a visual effect, and then systematically destroys the Actor until it no longer exists within the game world. |
| Inventory  (Inventory) |  | This component manages the Character’s inventory. The foundation for a larger system is in place, but the component only handles the Shield and Sword items in *Prototype*. |
| Rotation  (Rotating) |  | Controls character rotation at a more customized level than the default functions available to Character Movement. |

*Table 1 cont.*

|  |  |  |
| --- | --- | --- |
| Component  Name (Type) | Member Functions | Purpose |
| Equipment  (Equipment) |  | This component handles the equipment that is attached to the Character. Like the Inventory component, the foundation for a larger system is in place. This component would allow a developer to add or remove equip-able items from the Character as development needs evolved. For *Prototype* this Equipment component handles the Sword and Shield items that are attached to all the Characters in *Prototype.* |

Every component found in *Table 1* can be attached or removed from any Character – player controlled or not – depending on the needs of that Character for the game. *Figure 16* shows the blueprint event graphs and functions that are exclusive to the Player Character’s blueprint.

*Figure 16*. The functions and events of the Player’s Character blueprint.

### 4.2.5 Player Character Capabilities and Controls

At the start of the level, the player is spawned into the world with their sword and shield either equipped or unequipped. This is due to a bug depending on if the game is being played from the Unreal Engine Editor, or from a standalone executable. In either case, the player can toggle this by pressing R. If the player is not currently holding the sword and shield, they can press the left mouse button to draw their weapons instead of pressing R. The controls for the character are as follows:

* Move Forward – W, Up Keys
* Move Left – A, Left Keys (While not locked-on)
* Move Right – D, Right Keys (While not locked-on)
* Move Toward Camera – S, Down Keys (While not locked-on)
* Light Attack – Left Mouse Button
* Heavy Attack – Left Mouse Button + Shift Key
* Thrust Attack – Left Mouse Button + Ctrl Key
* Special Attack – Left Mouse Button + Alt Key
* Block – Right Mouse Button
* Parry – Right Mouse Button + Alt
* Toggle Equip Sword & Shield - R Key
* Toggle Target Lock-On – Tab Key
* Switch Between Targets – Q, E Keys (While locked-on)
* Sprint – Shift (While moving)
* Roll – Spacebar
* Jump – Spacebar + Shift Key
* Restart Level – L key
* Quit – Esc Key

Additionally, the camera is controlled with the position of the mouse, where the camera turns with the x-axis of mouse movement and angles up or down with the y-axis of mouse movement.

A light attack will do the least amount of damage to an enemy, and will leave the player open to being stunned, but is the fastest attack available. A heavy attack is much slower but deals more damage and does not allow the player to be interrupted. The thrust attack is the slowest attack but has vastly increased range compared to other attacks. The player is still vulnerable to stuns while thrust attacking. The special attack takes the most stamina to perform and deals the most damage but is only slightly faster than the thrust and cannot be interrupted.

The player can block any attack made by the enemy AI at the expense of stamina. If the player is blocking with low stamina and the AI brings the player to 0 stamina, the player is briefly stunned. Additionally, the player can parry enemy attacks. That is, if the parry move is timed correctly, the enemy AI’s attack will be interrupted, and they will be left open for counter-attack. If the parry is not timed correctly, the player will take full damage from the enemy attack.

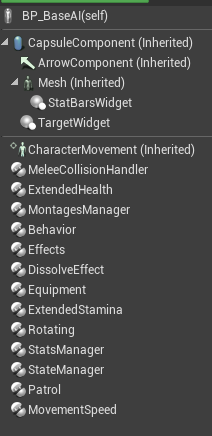
The player can lock-on to an enemy and switch between targets if they have engaged more than one AI. However, lock-on is not required to fight, and there are certain advantages to fighting without it.

## 4.3 Artificial Intelligence Character

### 4.3.1 Overview

The AI Character Blueprint is similar to the Player’s Character Blueprint. One of the key differences is that the AI Blueprint starts with a base version. This version has all the necessary components and base functions attached. However, it does not have any meshes attached. Any AI that is placed into the world must be based off a blueprint created as the base version as a template. This allows the developer to create new AI characters that have the same basic functionalities but can be altered at-will without having to worry about corrupting the original version.

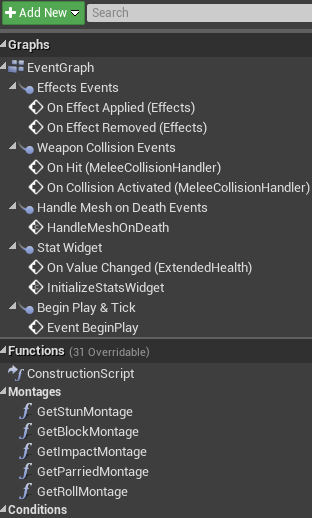
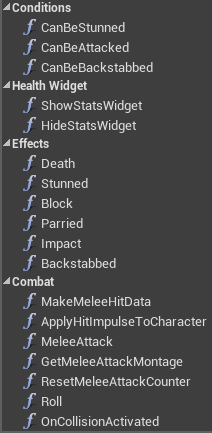
This method of creating AI is useful for making different enemy types without having to rebuild the base blueprint repeatedly. It also allows for assigning the same enemy type different values. For example, a “boss” AI can be created from the base blueprint and have all its stats increased dramatically with simple adjustments by the developer. *Figure 17* shows the components attached to the base AI Character Blueprint.



*Figure 17*. Components of the base AI Character Blueprint.

### 4.3.2 AI Character Blueprint Functions and Graphs

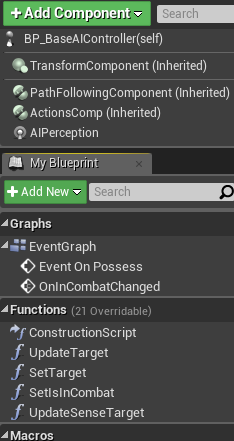
As with the Player Character, the AI Character has a set of functions that exist within the base AI blueprint. *Figure 18* shows a list of these functions and graphs.

*Figure 18*. AI Character event graphs and functions.

### 4.3.2 AI Controller Blueprint

The AI Character is controlled by an AI Controller, just as the Player Character is controlled by a Player Controller. The AI Controller has its own set of components and functions attached, as show in *Figure 18*.



*Figure 18*. The components and functions of the AI Controller.

Just as with other blueprints, certain components are inherited by the parent class. The name of the AI Controller for *Prototype* is “BP\_BaseAIController” and inherits from the “AIController” class of blueprint. An AI perception component must be added to the controller to allow the AI to perceive the game world in many ways, including sight and hearing.

### 4.3.3 AI Exclusive Blueprints

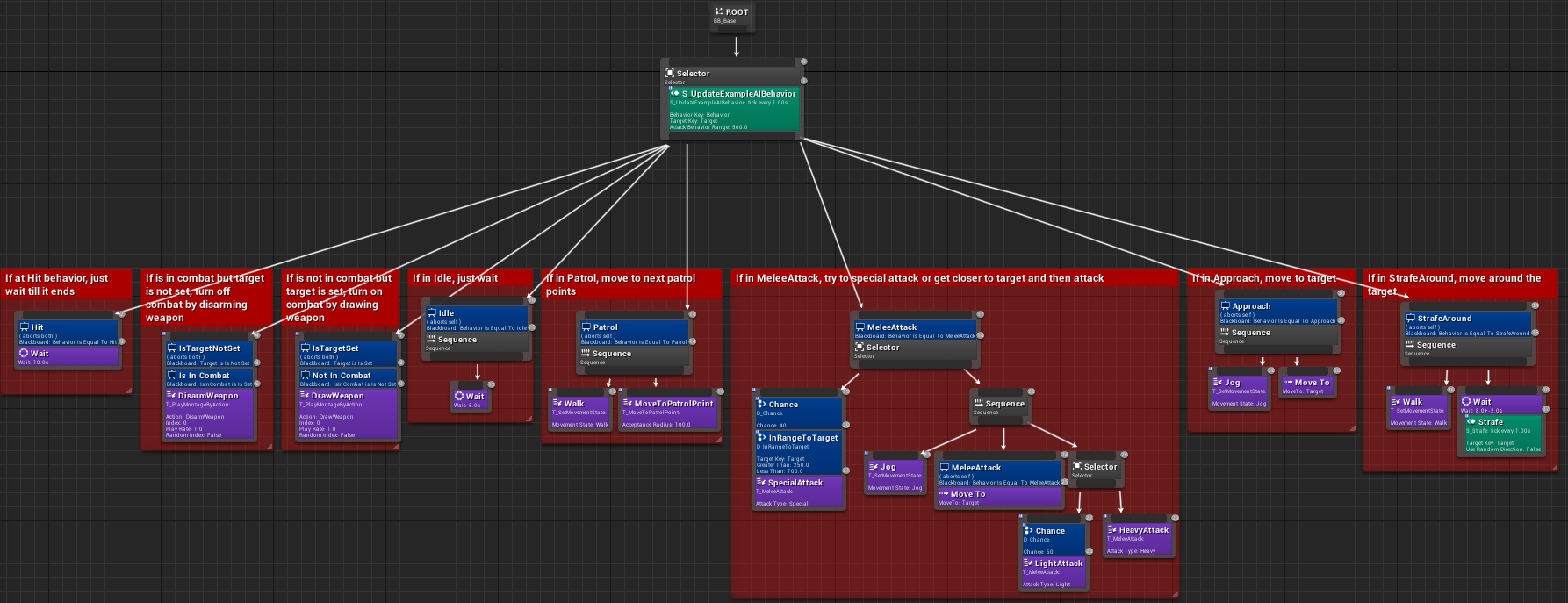
Notice that most of the components found on the AI Character in *Figure 17* are also present on the Player Character. Additionally, the AI Character Blueprint does not have all the same components as the Player Character. Components present on the Player but not the AI are the InputBuffer, DynamicTargeting and Inventory components. There are two components present on the AI Blueprint that are not attached to the player, another Actor Component table with these two components can be found in *Table 2*.

*Table 2*. Components found on the AI Character base Blueprint.

|  |  |  |
| --- | --- | --- |
| Component  Name (Type) | Member Functions | Purpose |
| Behavior  (Behavior) |  | This component controls which Actors the AI considers to be enemies. The AI behaves in an aggressive manner to any designated Character type, including other AI. |
| Patrol  (Patrol) |  | The purpose of this component is to control the patrol path for the AI Character. It finds the associated patrol, then communicates with the AI Controller to tell the AI where to move. |

### 4.3.4 AI Decision Tree

AI behavior is dictated by its associated behavior tree, which is initiated when the level is loaded. AI behavior consists of two parts, first is the Behavior Tree while the other is the Blackboard. The behavior tree functions similarly to any tree data structure, where nodes are travelled based on leftmost rules. The blackboard communicates with the behavior tree and accepts values from blueprints associated with the behavior tree. *Figure 19* shows an overview of the behavior tree used for the AI enemies.



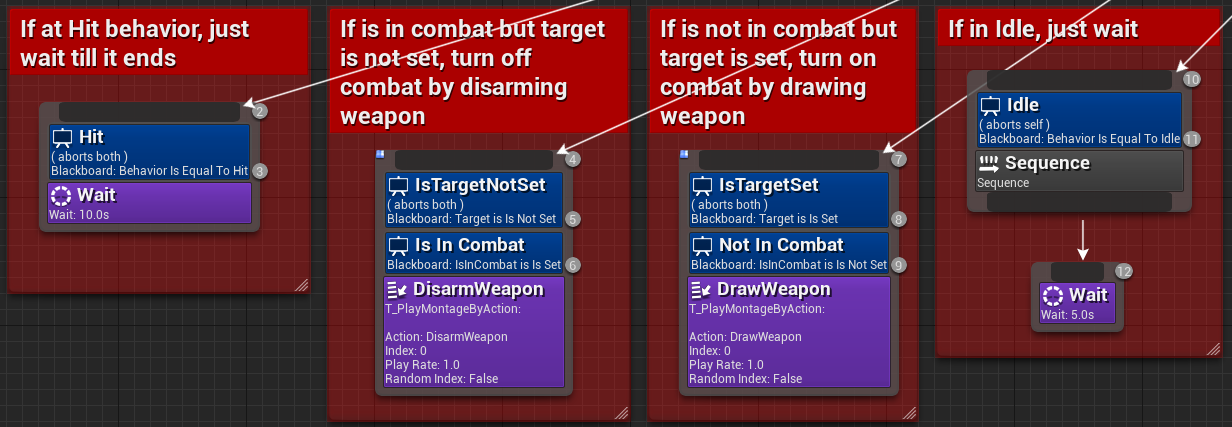
*Figure 19*. The AI Behavior Tree overview.

Starting at the uppermost nodes, the structure of the tree can be defined, as shown in *Figure 20*.



*Figure 20*. Zoom of the Root node.

The Root node is simply the start of the tree and is not actually included in any runtime decisions. Because of this, the first node coming from the root node is labeled as node 0, as shown in *Figure 20*. Right below the Selector node, a custom function is attached, and runs immediately after 0, making it the 1 node. These hierarchy positions are assigned automatically based on each node’s physical location on the graph, which is updated immediately upon any changes made to positions. AI Behavior Trees in Unreal Engine 4 travel down the leftmost node, executing any functions. However, since this tree starts with a Selector node, it behaves differently. A Selector node chooses any of its children based on specific conditions. Each node is assigned its own set of conditions, which are evaluated during gameplay. *Figure 21* shows the first 4 children nodes of the Selector node.

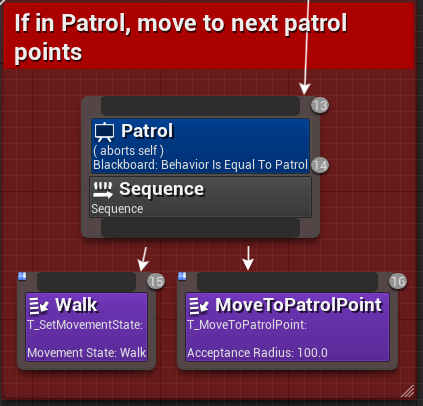


*Figure 21*. A closeup of the first 4 child nodes of the root.

The first child node is 2, with its associated Wait function as 3. The Selector reaches this node, and if the Hit condition in node 2 is not set, it aborts all the children of node 2. The Selector evaluates every child node until it reaches the first node that succeeds. As an example, an AI that is not in combat and does not have a patrol will follow this sequence of events each time the game evaluates the tree:

1. Travels to node 2, evaluates Hit as fail, aborts node 3
2. Travels to node 4, evaluates IsTargetNotSet as success
   1. Travels to node 5, evaluates IsInCombat as fail, aborts node 6
3. Travels to node 7, evaluates IsTargetSet as fail, aborts nodes 8 and 9
4. Travels to node 10, evaluates Idle as success, executes 11 (a Sequence node)
   1. Travels to node 12 and executes Wait, then restarts at node 0

The Sequence node in the next section of the tree better exemplifies the behavior of that node, as shown in *Figure 22*.



*Figure 22*. The patrol behavior Sequence node.

Here, the Sequence node travels down each of its child nodes in order before returning to its own parent node. That is, when the Selector at node 0 reaches node 13, and evaluates Patrol as a success, the sequence executes a movement function along the designated patrol path for that AI Character.

There are four kinds of functions shown in the behavior tree. The first is the Sequence and Selector nodes, which have already been described. The second type of function is the Task, which is shown in purple. Tasks work just like any other function, executing code when the node is reached. Only one Task can be assigned to any one node. The third type is called a Decorator and is the blue sections on the nodes. Decorators control the flow of the tree and can interrupt or lock the traversal of the AI Controller. Decorators return either success or failure depending on given conditions. Decorators associate their condition checks with Blackboard Key values as defined by the developer. The fourth kind of function is called a Service and are the green sections of the nodes. They function similarly to any other blueprint and interact with the Behavior Tree through Blackboard Key values.

### 4.3.5 AI Behavior Tree Nodes

As mentioned in section 4.3.4, the Behavior Tree uses Decorators, Tasks and Services to communicate with the AI Controller and give the AI Character commands while the game is running. *Table 3* includes every Behavior Tree node in use by the Behavior Tree of the AI.

*Table 3*. The blueprints of the Behavior Tree nodes.

|  |  |  |
| --- | --- | --- |
| Node  Name (Type) | Member Functions | Purpose |
| D\_Chance  (Decorator) |  | Returns a random integer between 1 – 100, used for melee attack probabilities. |
| D\_InRangeToTarget  (Decorator) |  | Checks whether the AI’s target is in range to engage in combat. |
| T\_MeleeAttack  (Task) |  | Tells the AI Character to perform the Melee Attack function (seen in *Figure* |

*Table 3 cont.*

|  |  |  |
| --- | --- | --- |
| Node  Name (Type) | Member Functions | Purpose |
| T\_MoveToPatrolPoint  (Task) |  | Tells the AI Character to locate the next patrol point in the patrol path and move to it. |
| T\_PlayMontageByAction  (Task) |  | Plays an AI animation based on the action taken in the Behavior Tree. |
| T\_SetMovementState  (Task) |  | Toggles the AI’s movement state between Idle, Walk, Jog and Sprint. |
| S\_Strafe  (Service) |  | Makes the AI Character strafe its target depending on the target’s location relative to its own. |
| S\_UpdateBehaviorAI  (Service) |  | When called by the tree, this Service sets the behavior key of the Blackboard to the desired value, depending on the AI Character’s current state (Idle, Patrol, MeleeAttack, |

### 4.3.6 AI Character Capabilities

Every AI Character has the same abilities as each other, as time restricted creating varying enemy types. Every AI has a sword and a shield, just like the player, and uses many of the same animations.

When the AI can “see” the player, the character will move into range and engage the player in combat. The AI will either circle the player in a defensive stance or approach to attack, depending on what the behavior tree tells the AI. The AI is capable of the following:

* Attacks – Light, Heavy, Special
* Blocking
* “Seeing” the player
* Equipping and unequipping sword and shield
* Patrolling a designated path
* Death

When the AI is brought to 0 health, they are sent into ragdoll mode and after a few seconds, the Dissolve Component removes them from the game world.

## 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 17* shows the positioning of the player’s health and stamina bars.



*Figure 17*. 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 18* shows these two attributes in the display window.



*Figure 18*. 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

The design of the level for *Prototype* was built to facilitate combat testing. Structures and enemy placement are used to test each feature individually. *Figure* shows an overview of the level with the player’s Character model placed for scale. The white line is one of the AI Character’s patrol paths.



*Figure* . Overview of the level.

# 5. Conclusion and Future Works

# 6. References