

Janessa Aulén

# Animation State Machines in Unreal Engine

Bachelor's Thesis

Degree Programme in Information Technology

April 2020



**Kaakkois-Suomen  
ammattikorkeakoulu**

<b>Tekijä/Tekijät</b>	<b>Tutkintonimike</b>	<b>Aika</b>
Janessa Aulén	Insinööri (AMK)	Huhtikuu 2020
<b>Opinnäytetyön nimi</b>		79 sivua
Animaatiotilakoneet Unreal Engine -pelimoottorilla		
<b>Toimeksiantaja</b>		
Gamelab		
<b>Ohjaaja</b>		
Marko Oras		
<b>Tiivistelmä</b>		
<p>Tämän opinnäytetyön tarkoituksena oli tutkia ja opetella käyttämään Unreal Engine -pelimoottorin animaatiotyökaluja ja työvaiheiden kulkua. Tarkoitus oli myös saada lisättyä aiemmin aloitettuun peliprojektiin pelaajahahmolle ja muille hahmoille perittäviä animaatiotilakoneita.</p> <p>Tämän dokumentin on samalla tarkoitus toimia raporttina siitä, mitä ongelmia ja huomioita tuli vastaan toteutuksen aikana ja mitä pitäisi ottaa huomioon animaatioihin liittyvissä asioissa tulevaisuudessa projektin kehittämisen kannalta.</p> <p>Pohjana käytetty projekti on toteutettu Unreal Engine -pelimoottorilla, joten myös opinnäytetyön aikana tehty työ on tehty samalla pelimoottorilla suoraan samaan projektiin.</p> <p>Opinnäytetyön aikana toteutettiin peliprojektiin animaatiotilakoneet pelaajahahmolle ja yhdelle lisähahmolle. Suunnitelmista poiketen tilakoneet eivät ole suoraan perittävässä muille hahmoille. Koska pelimoottorin animaatioiden työkulusta ja työkaluista ei ollut ennalta suurta kokemusta, toteutuksesta on suuri osa ollut kokeilemista ja kaikki alkuun suunnitellut toiminnot eivät toteutuneet niin kuin oli aluksi tarkoitus.</p> <p>Opinnäytetyön tärkeimmät tavoitteet saavutettiin. Tulevaa kehitystyötä varten myös animaatioon liittyvistä ongelmista ja tarvittavista ennakkotiedoista on projektin jäsenille tehty selkokielistä projektikohtaista dokumentaatiota.</p> <p>Tämän työn lukijan olisi hyvä ymmärtää C-kieleen perustuvaa ohjelmointikieltä. Työssä näkyvä koodi on erittäin helppolukuista, mutta ohjelmoinnin perusasioita ei käydä työssä läpi. Olisi myös hyvä, että lukijalla on hallussa Unreal Enginen tai vastaavan pelimoottorin perusteet.</p>		
<b>Asiasanat</b>		
dokumentaatio, peliprojekti, Unreal Engine, animaatio, animaatiotilakoneet		

<b>Author (authors)</b>	<b>Degree</b>	<b>Time</b>
Janessa Aulén	Bachelor of Engineering	April 2020
<b>Thesis title</b>		79 pages
Animation State Machines in Unreal Engine		
<b>Commissioned by</b>		
Gamelab		
<b>Supervisor</b>		
Marko Oras		
<b>Abstract</b>		
<p>The purpose of this thesis was to study and learn to use the animation tools and workflow in the commercial game engine Unreal Engine. It was also intended to make inheritable animation state machines for a game project's player character and other characters.</p> <p>This thesis also serves as a report of the problems and discoveries regarding the animations during the implementation, which should be considered in the future development of the project.</p> <p>The project used as a basis for the thesis has been developed with Unreal Engine, so the work done for the thesis has also been done with the same game engine and implemented directly into the game project.</p> <p>During the thesis study, animation state machines for the player character and one additional character were implemented. Contrary to the initial plan, the state machines cannot be directly inherited for other characters. Since there was only little prior experience with the animation tools and workflow, many parts of the thesis had been experimental and not all of the features that were initially planned were implemented in the way originally intended.</p> <p>For future development, clear project-specific documentation concerning the problems and possible information needed prior to working in animations was also made and is now available for the project members.</p> <p>The code shown in the work is very easy to read, but the basics of programming are not covered and so the reader should understand a programming language based on the C language. It would also help the reader to be familiar with the basics of Unreal Engine or a similar game engine.</p>		
<b>Keywords</b>		
documentation, game project, Unreal Engine, animation, animation state machines		

## Table of Contents

1	INTRODUCTION .....	8
2	PROJECT .....	8
3	TOOLS .....	10
3.1	Blender .....	10
3.2	Mixamo .....	11
3.3	Unreal Engine 4 .....	12
4	ANIMATION .....	12
4.1	Traditional Animation .....	12
4.2	Computer Animation .....	13
4.2.1	2D Animation .....	14
4.2.2	3D Animation .....	15
4.3	Animation in Games .....	16
5	ANIMATION SYSTEM IN UE4 .....	17
5.1	Terms and concepts .....	17
5.1.1	Animation Tools .....	17
5.1.2	Skeleton Assets .....	18
5.1.3	Animation Sequences .....	18
5.1.4	Animation Notifications (Notifies) .....	18
5.1.5	Animation Blueprints .....	18
5.1.6	Blend Spaces .....	19
5.1.7	Animation Montage .....	19
5.1.8	Skeletal Control .....	19
5.1.9	State Machines .....	19
5.2	Animation Blueprints .....	20
5.3	Reusing Animation Blueprints .....	20
6	IMPLEMENTATION .....	21

6.1	Original plan for implementation .....	21
6.2	Assets.....	21
6.3	Importing assets into UE4.....	22
6.4	Finished animation logic and blueprints.....	22
6.5	Implemented animation states.....	23
6.5.1	Idle state .....	23
6.5.2	Movement state .....	29
6.5.3	Jump state .....	30
6.5.4	Attacking state .....	42
6.5.5	Crouching state.....	44
6.5.6	Aim offset.....	47
6.6	Retargeting animations to another character.....	50
6.7	Other implemented features .....	59
6.7.1	Camera.....	59
6.7.2	Simple AI for NPC.....	63
6.8	Future development.....	70
7	CONCLUSIONS .....	71
	REFERENCES .....	73
	LIST OF FIGURES .....	76
	LIST OF TABLES .....	79

## **Terms and Abbreviations**

### **2D**

Two-dimensional. Something that consists of only two dimensions, width and height or x-axis and y-axis. There is no depth or z-axis.

### **3D**

Three-dimensional. Something that consists of three dimensions. Width, height and depth, so there are x, y and z-axis.

### **AI**

Artificial intelligence. A machine or program that tries to simulate human-like behaviour. In this thesis, a non-playable character has a very simple AI implemented which allows it to move from one place to another.

### **CharacterMovement**

A readymade component in Unreal engine where ready-to-use logic for movement has been implemented, for example, falling, jumping and walking. The settings and functions for controlling the movement are contained within the component.

### **NDA**

Non-disclosure agreement. Is a contract where the employee or visitor agrees to not let anything they learn, hear or see leak outside of the company or their team working on the project.

### **NPC**

Non-playable character. In games, this is a character that the player cannot control. Most often NPCs are side characters that bring the game's world to

life as its inhabitants and they may give the player some info regarding a quest or mission.

### **Player Blueprint**

In this thesis, the **Player Blueprint** refers to the logic script used to drive the player character's logic. Actions such as jump, attacking, moving and input are in the **Player Blueprint**. The **CharacterMovement** component can be found inside of this Blueprint and is used to implement some of the player character's logic.

### **Player Animation Blueprint**

This refers to the script that takes care of the player character's animations and animation logic in the thesis project. For example, when the player moves the character, this script makes the walking or running animation play.

### **UE4**

Unreal Engine 4. It is a free-to-use commercial game engine. It can be used to create the game levels and everything a game consists of, such as programming or scripting the logic, adding 3D models and other assets and then at the end it can export the creation as a playable game for a computer, mobile or a console.

## **1 INTRODUCTION**

The purpose of this thesis was to learn more about animation in video games and to learn how to use a commercial game engine, Unreal Engine 4's basic animation workflow and tools to implement animations for a game project's main character and possibly non-playable characters by re-using the same animations and logic.

For this thesis, it was decided to use premade animations that can be downloaded from an online service by Adobe at [mixamo.com](https://mixamo.com).

The intention for this thesis was to implement working animation logic and reusable animations into an ongoing game project.

Additional secondary objectives for this thesis study was to identify what is needed for the project in terms of animation and animation logic in order to work now and in the future and to experiment with the animation tools and related workflow in Unreal Engine 4.

There is in-depth documentation written about the different animation tools online, and anyone who plans or wishes to work with UE4 and animation in any form should properly familiarise themselves with the material. This thesis will focus more on the tools used, but it is still expected of the reader to have a basic understanding of UE4.

The commissioner for this thesis is Gamelab. Gamelab is also a learning environment for students to fully utilise of the industry level skills and knowledge provided by their curriculum and apply them in their game projects as well as any software projects.

## **2 PROJECT**

The game project used as a basis for this thesis is called Chicken Quest, a game in development that is a side-project by a group of three people including the author of this thesis. This project will be finished and released in the future.

Chicken Quest is a 3D platformer and adventure game. The idea for this game came into being at a game jam in 2017. During the game jam, a simple side-view 2.5-dimensional platform styled demo was made with the game engine



Unity and it even made it in to the top three games of the jam. It was a simple demo where a witch-like girl is searching for magical crystals in a cave filled with lava, then if there were time left at the end of the jam add a magic school environment to explore or something, but the magic school did not make it to the demo then. No more story was planned.

Later in the summer 2018, the same idea was taken to another game creation event, the Game Camp, held in Kotka. The game changed there from a 2.5-dimensional to a 3D and the idea was completely revised. More story elements were created with the power of a team of four people, hastily put together at the site. Another demo was made during the Game Camp with Unity as well.

The story elements added included the situation of the main character and some elements of the surrounding world. The main character was to be a boy who is accepted into an out-of-the-way magic school in the countryside, but the school prided its other departments over the magical one. The main character was only accepted owing to his rare magic of transformation, even though he could only turn into a chicken.

The game was to be a parody of magical school fantasies. The game mechanics of turning into a chicken will give the main character the ability to move slightly faster and jump somewhat better and these skills would help him clear the platforming levels and obstacles in the game. Maybe he would learn more “useless” spells later.

The demo contained one level where it was possible to battle with an evil wizard and one where the chicken form was needed to reach a chest with a key. Nothing would happen after the battle or from acquiring the key, since, it was a demo to showcase the mechanics of the game after all.

At the end of the Camp, a question was asked about the game. “Why does he turn into a chicken?” The answer given was “because he is afraid”. Later in the following fall, the project was started a new again, this time with Unreal Engine 4. UE4 was chosen so that the team members still embracing the game idea would be able to work on an interesting project and learn the use of the popular game engine and the demos made with Unity were not really pleasing, it was also easier to start from the beginning as many of the features

made in the demos were just quick fixes on top of quick fixes. The insides of the projects were messy and hard to make heads or tails out of.

In sum, the project is slowly moving forward even at present time. The team working on it do it as a side project, which is the reason it is regularly stalled. At the time of writing this thesis two thirds of the people in the team could not work on it at all. This was part of the reason why the initial plan for the implementation had to be changed.

### **3 TOOLS**

The tools used for the thesis were chosen when the Chicken Quest project was started anew with UE4 in 2018, with the exception of using premade animations to save time for learning to use the tools and workflow for animation as making custom animations would take more time than was possible with the time frame set for this thesis.

Even though the tools were chosen at the very start of the project, they proved to be valid. They are provided with regular updates and changing them mid project would at the very worst cause the project to have to be started from the beginning, which would be the case, for example, if the game engine was changed.

#### **3.1 Blender**

Blender is an open-source 3D toolset. This means that it is a free-to-use computer program that can be used to create 3D models and to rig and animate them. It is also favoured by many because it is free and has massive amounts of tutorials online, so learning to use it is quite easy. It is also quite versatile and can handle modelling, rigging, animating and video editing, to name a few of its features. It is also updated regularly, and it is possible to write custom components for it using Python. This enables the creators to make custom workflows to help them achieve what they desire.

Blender was created to support the entirety of 3D pipeline, the process of working with 3D. This includes modelling, rigging, animation, simulation, rendering, compositing and motion tracking, video editing and 2D animation pipeline. (Blender Foundation no date.)

3D pipeline can be understood to be the different steps of 3D modelling. According to Collins (2018), the typical stages of 3D pipeline are: pre-production, 3D Modelling, UV Mapping, texturing and shaders, rigging, animation, lighting, rendering and compositing.

Blender was used to tweak the premade animations and verify how they look right after being downloaded from Mixamo.

Other 3D programs were not considered as Blender is free-to-use and all the team members were able to use it prior to this project so there was no need to learn new tools or workflows.

### **3.2 Mixamo**

Mixamo.com is a website by Adobe where it is possible to upload premade 3D models of humanoid characters to add animations for them. It also offers readymade 3D characters that can be downloaded and used if there are no custom models available.

According to Adobe (2020) it is the first online character animation service.

Mixamo was used to acquire a base character mannequin and all the animations used in the thesis.

Alternative choices for making animations could have been to make them using a 3D program, such as Blender. However, making all the needed animations for this thesis would have taken more time than what was planned. It must be mentioned that custom made animations would have looked better and given the characters more of a personal feel.

Mixamo first uploads a 3D model to their site, then their online tools make a rig for it and it can be given any of the animations on the site and, finally, the animations can be downloaded for a game engine or other 3D program with that same model. Mixamo also offers their own models, which allows the user to choose a model from them, add animations to it and use it if self-made models are not available.

### **3.3 Unreal Engine 4**

Unreal Engine is a complete package of development tools available for working with real-time technology. They can be used for design visualizations and cinematic experiences to high-quality games across PC, console, mobile, VR, and AR. The user can use UE4 for everything they need from starting and shipping to growing and stand out from the crowd. (Unreal Engine, 2020e.) This means UE4 is basically a program used to make games and interactive content.

UE4 was chosen for the game project as it is one of the largest commercial game engines and has a great number of tutorials and support online to learn from. Games made with it include Borderlands 3, Final Fantasy VII: Remake, Fortnite and Octopath Traveler, which are just few. All the after mentioned games are very popular and different from each other so it can be said that UE4 highly suitable for different genres and types of games.

## **4 ANIMATION**

### **4.1 Traditional Animation**

Animation itself is not the main topic of this thesis, but it still needs to be introduced briefly to better understand animation states. This introduction will focus mainly on computer animation, but to understand the difference between it and traditional animation will be useful for understanding why computer animation works the way it does.

Traditional animation is often explained by referring to old Disney animations where everything was drawn by hand on paper. According to Sito (2013, 218-219) there were multiple stages to making animation and multiple teams working on each small step along the way. These stages included storyboarding, sounds, layouts, backgrounds and the drawing characters and their movement. Many inspections were also made to ensure everything was perfect. For every stage a final review was always made by an animator director. Sito (2013, 220) also indicates that classic animation was good in its time, but that it had become too expensive and labour-intensive to be profitable.

The production of animated films once required large teams of artists, in large buildings which collectively comprised the large animation studio. Nowadays, all that was once done by such a massive team can effectively be produced by a single user on one desktop computer—in theory, at least. (White 2006, 184.)

## 4.2 Computer Animation

Unlike in traditional animation, in computer animation computers are used not only in the management steps, but also for making the graphics and images used in the animation. Also, different animation programs are used to work on other parts of the whole process, for example, drawing, editing and adding special effects. This was not possible in the age of traditional animation, before computers were easily available for individuals and small groups.

In 1977, engineer Dave Wolf attended a screening of Disney's *Sleeping Beauty* at the USC cinema school. "When the old Disney guys asked for questions, I asked if they could see a role for computers. Immediately, I got a lot of ugly looks from everyone." (Sito 2013, 222.)

When computers were brought into animation studios, traditional animators were skeptical about using computers in the production. There were many animators who disliked the thought of computer made animation, but even then, there were those who understood the possibilities a computer could bring into the industry. Sito (2013, 222-223) also says that the demand for quality animation grew in 1980s, and animation studios increasingly started to work CG into their production process. *Beauty and the Beast* was a huge hit, which made many skeptics in Hollywood begin to look at CG seriously (Sito 2013, 231). Also, as is known today, computers did make their breakthrough and are used everywhere and with the computers, animation changed.

White (2006, 184) states that there are differences in the production of different types of animation of which the principal two are 2D and 3D. Pointing them out helps understand the general processes that go into all aspects of making animation.

### 4.2.1 2D Animation

The description of 2D animation in the following four chapters is based on a blog entry by a professional video content creation service Renderforest (2019). 2D animation is one of the primary types of animation and it is extensively used in media from cartoons to advertisements and educational material.

In 2D animation, the characters are created in two-dimensional space and they only have width and height, no depth. This is considered to be the traditional animation style.

2D animation consists of three main phases. In the first stage, pre-production, the story and script are developed, and the characters and a story board are created. After which, the colour palette is chosen, and the voice-overs are made. Production is the second stage where the animation is created by gathering all the created materials and putting them together to produce the scenes. In this step, the backgrounds are painted, and individual scenes and character activities are made into a rough animation and then cleaned and polished. In order to have everything put together, the animators create an exposure sheet where all the instructions on how to make the scene are marked. The third step is the post-production. It is the final editing process where the animation is enhanced and then exported to different formats.

In order to make the best possible 2D animation, the 12 principles of animation are usually followed by professional animators. These principles may be best described as tips or tricks by the Disney animators Ollie Johnston and Frank Thomas.

The 12 principles of animation are:

1. Squash & stretch
2. Staging
3. Anticipation
4. Straight ahead & pose to pose
5. Follow-through & overlapping action
6. Slow in & slow out
7. Arcs

8. Secondary action

9. Appeal

10. Timing

11. Exaggeration

12. Solid drawing

As stated by Cooper (2019, 27) although these fundamentals are old and come from the pre-computer graphics days of hand-drawn animation, with slight reinterpretation they translated perfectly for the later evolution of 3D animation.

#### **4.2.2 3D Animation**

As can be understood from the internet article about 3D animation by Chang (2020), 3D animation is the process of creating moving pictures in a digital three-dimensional environment. 3D models are carefully manipulated within the 3D program then the picture sequences can be exported, which will then create the illusion of motion.

According to Chang (2020), 2D graphics are represented on a two-dimensional platform while 3D graphics on a three-dimensional platform. In 2D, the images are flat with only one perspective, and the colouring and shading are relatively simple, for example, shadows are hard and not soft as they appear in the real world. In contrast, 3D images have depth and multiple perspectives. The shadows are also more subtle and soft. Chang (2020) also states that 2D looks unrealistic and cartoonish, but even if 3D can look cartoonish, at the same time it also looks realistic.

In an attempt explain more in depth how, for example, a character could be animated in 3D, an article about 3D animation by Bloop Animation (2019) states that in 3D animation, for example, when animating a 3D character, the character model is rigged. It means that the model has virtual bones programmed into it. This enables the animators to animate or move the character. The act of grabbing a control bone for the character and moving it does not on its own animate the model. For the animation to be registered or to happen, the movement has to be added into a timeline. Selecting the wanted point in time in the timeline and adding a keyframe with the part of the

character moved to its correct position is what tells the program that the part will animate at this time, and the movement is saved. However, with only one keyed frame nothing really happens. In order to complete the animation, the animator then goes forward in the timeline adding all key poses of the full movement, for example, walking, and the program then automatically interpolates the steps between of the keyed frames, and the full movement is generated. Another major factor to take into consideration is the frame rate. In film, there are usually 24 frames per second, which for 2D animation means 24 drawings per one second. However, when there are no significant movements in 2D it is enough to have one drawing for every two frames or, with very still movements, one drawing for every four or five frames. With 3D, the character would seem dead if it is kept completely still for even a second. When the character is not particularly doing anything, it must have an animation, such as an imitation of breathing, when being idle or still in order for it to maintain the illusion of being alive.

### **4.3 Animation in Games**

Based on an article by Pluralsight (2014), animation is different for games and other media such as movies. In a game, the animation needs to look good from every angle, and at times the animated movements need to be fast but realistic. The reason for this is the fact that games are meant to be interactive and many of the controls that drive the animation lie with the player. For example, in shooting games some animations such as the reloading animation needs to be fast enough so the player can go quickly back into action but at the same time slow enough to be realistic and add a challenge to enhance the gameplay experience.

As said also by Cooper (2019, 44), games are made of many short animations playing in sequence rather than long flowing animations. As such, they are often stopping, starting and overlapping.

Also, the article by Pluralsight states that game animation is not about creating top-notch performances such as are seen in Frozen, the Disney movie, but ensuring the animation will work well for the player.



The after mentioned points are a few to keep in mind about animations in games and help to understand the possibilities and difficulties that may arise when creating games.

## 5 ANIMATION SYSTEM IN UE4

The Animation System in UE4 is consists of several different tools and editors and they mix skeletal-based deformation of meshes with morph-based vertex deformation to allow for elaborate animation. The system is used to make the basic player movement more realistic and create customized special moves. For example, it can be used for scaling ledges or walls, applying damage effects, controlling facial expressions, directly controlling bone transformations or telling the character through logic which animation it should be given depending on the situation. (Unreal Engine 2020c.)

### 5.1 Terms and concepts

UE4's documentation lists the terms and concepts of the Animation System on their documentation as listed below. All the following explanations about the terms are based on UE4's documentation.

#### 5.1.1 Animation Tools

Several different Animation Tools are used to create animated characters. Each of the Tools focus on different steps of animation.

For example, the **Skeleton Editor** is where everything begins and, it is used to manage the bone (or joint hierarchy) that controls the **Skeletal Mesh** and its animation. The **Skeletal Mesh Editor** is used to modify the **Skeletal Mesh**, the outward appearance of a character, and it is linked to a **Skeleton**. In the **Animation Editor** animation assets can be created and modified and is where the tuning and tweaking of animation assets are done. The **Animation Blueprint Editor** is used to create the logic that controls what animation a character uses and when, it is also what controls how animations are blended together. Last is the **Physics Asset Editor**, which can be used to create and modify the **physics bodies** that are used for the collision of **Skeletal Meshes**. (Unreal Engine 2020c.)

### 5.1.2 Skeleton Assets

A **Skeleton** is a hierarchy of bone locations and rotations used to deform a **Skeletal Mesh**. In UE4, **Skeletons** are separated from **Skeletal Meshes** in their own asset. The animations are applied to the **Skeleton**, and not the **Skeletal Mesh**. Thus, when multiple **Skeletal Meshes** use the same **Skeleton** they can share animations. (Unreal Engine 2020c.)

### 5.1.3 Animation Sequences

An **Animation Sequence** is a single animation asset that can be played on a **Skeletal Mesh**. They contain keyframes that indicate the position, rotation, and scale of a bone at different points in time. The bones of a **Skeletal Mesh** are smoothly animated by playing the keyframes back in a sequence, with automatic blending between the frames. (Unreal Engine 2020c.)

### 5.1.4 Animation Notifications (Notifies)

**Animation Notifications**, **AnimNotifies** or **Notifies**, can be used to setup events to occur at specific points during an **Animation Sequence**. **Notifies** are mostly used to add effects during the animation. The effects can be sounds for footsteps or particle effects for rising dust from the ground. It is also possible to extend the system with custom notifications to accommodate the needs of any type of game. (Unreal Engine 2020c.)

### 5.1.5 Animation Blueprints

An **Animation Blueprint** is a specially designed **Blueprint** that controls the animation of a **Skeletal Mesh**. **Graphs** are edited inside of the **Animation Blueprint Editor** to directly control the bones of a **Skeleton**, perform animation blending and setup the logic that will define the final animation pose for a **Skeletal Mesh** to use per frame. (Unreal Engine 2020c.)

### 5.1.6 Blend Spaces

**Blend Spaces** are assets which can be sampled in **Anim Graphs**. **Blend Spaces** are used to blend multiple animations together based on multiple values, which are currently limited to two. To blend together only two animations the standard **Blend** nodes can be used and are available in **Animation Blueprints**. (Unreal Engine 2020c.)

### 5.1.7 Animation Montage

**Animation Montages** or **Montages** enable a large variety of animation effects, mainly used to expose animation controls within **Blueprint Visual Scripting** or through code. Animation effects, along with intelligent loops of animation and logic-based animation switching can also be created by using **Montages**. (Unreal Engine 2020c.)

### 5.1.8 Skeletal Control

**Skeletal Controls** or **SkelControls** can be used to directly control bones within a **Skeleton** asset. Within **Animation Blueprints** it is possible to use them to control an individual bone or create IK chains. This direct control of the hidden **Skeleton** enables the creation of procedural, dynamically driven animation. The **transform** of a bone can be used to drive another or can be used to coordinate the feet of a character to the ground while playing a walk animation, for example, on an uneven ground. It is also possible to modify, tweak or override the bone **transforms** applied by **Animation Sequences**. (Unreal Engine 2020c.)

### 5.1.9 State Machines

**State Machines** provide a visual way to break the animation of a **Skeletal Mesh** into a series of states. These states are controlled by **Transition Rules** that handle how to blend from one state to another. **State Machines** are used to simplify the design steps of **Skeletal Mesh** animation, a visual graph can be created easily, and used to control how characters alternate between different

animations without the need for a complicated **Blueprint** network. (Unreal Engine 2020c.)

## 5.2 Animation Blueprints

As **Animation Blueprints** are some of the most essential tools used in the implementation part of the thesis so they will be given more coverage than the other tools.

**Animation Blueprints** are visual scripts used for the creation and control of complicated animation behaviours (Unreal Engine 2020a). According to Unreal Engine (2020a), there are two main components in **Animation Blueprints**: The **Event Graph** and the **Anim Graph**. **Event Graph** updates the values used in **Anim Graph** which is used to drive the animation states, blends and other nodes using said values. **Anim Graph** is essentially where the different animation states are shown and where the new states will be added when creating new animation behaviour. To change the animation, transition rules are made between the wanted states for when the animations should go from one state to the next. For example, on the player character, this is where the movement animations and other animations states can be found and edited. A character could have a walking and running state, the transition rule for the change could be if the character's speed rose over a certain value to blend to the running animation. As for **Event Graph**, it is the visual representation of the logic scripts that drive the animations. It is where the values used to change between animation states are obtained and they can then be accessed in the **Anim Graph** inside the animation state transitions. Here, for example, the speed could be fetched from the **Player Blueprint** and made into a local variable for the **Animation Blueprint** to use in the transition rules between the different animation states.

## 5.3 Reusing Animation Blueprints

When working with multiple characters it is possible to share the animation logic between the characters if they use the same skeleton by using **Sub Animation Instances**. With the **Sub Anim Instance** node, a reference to a separate **Animation Blueprint** can be created and used for accessing as well

as including all of its logic within another **Animation Blueprint** (Unreal Engine 2020a). Another example of using old animation logic with new similar characters is the use of **Child Animation Blueprints**. If multiple characters are similar but they should have different animations from each other, it may be possible to use a **Child Animation Blueprint** to override the animation assets, which should be replaced. This way there is no need to create individual **Animation Blueprints** for every character. The **Child Animation Blueprint** will inherit everything from the parent, and it enables the overriding and specification of single animations via the **Asset Override Editor**. (Unreal Engine 2020a.) As with the **Sub Anim Instance**, the characters must have the same skeleton to be able to behave properly.

## 6 IMPLEMENTATION

As the implementation part is finished, it has become apparent that there are some aspects that could be improved, but to within the framework of this thesis it was deemed necessary to halt the development work and work on the documentation as well.

### 6.1 Original plan for implementation

Before starting the implementation, the initial plan for what was needed and what was to be done included many of the features that were eventually finished by the end.

The initial plan was to use self-made custom models that would be used in the game, make one parent Animation Blueprint and have the rest inherit from it to minimise the workload. The **Animation Blueprint** was intended to have states for being idle, moving, jumping and maybe attacking. The aim was to achieve this by using **Child Animation Blueprints** which are a component in UE4.

### 6.2 Assets

In order to be able to do any animation logic or testing, there was a need for 3D models and animations for them. Differing from the original idea, it was not possible to acquire custom 3D models for the implementation part of the study

as there were no resources to make them by the time they were needed. There was one self-made 3D model from an earlier project that was used with testing the animation logic but as it used an unconventional custom rig, it was later replaced. Instead, models from Mixamo were used. For the base or parent Animation Blueprint, a simple mannequin model was chosen from Mixamo. For another model, a cartoon styled girl character was chosen testing the inheritance. After that the needed animations were given to the mannequin and downloaded. The 3D models and animation files were both .fbx files as those are what UE4 understands (2020f, UE4). Fbx files are available from Mixamo without the need to convert them via another program.

### 6.3 Importing assets into UE4

When importing assets into UE4, the assets files can be dragged and dropped from the folder to the **Content Browser**. There is also the option to select import from the Content Browser header. More comprehensive guides for best export settings for .fbx files from Blender to UE4 and import settings for UE4 can be found online.

In this thesis, the export settings are not explored as no custom models were used. However, it should be mentioned that an announcement has been made by UE4 (2020d) that they are building an add-on for Blender that takes care of the exporting of .fbx files from Blender to UE4 themselves, but it is not available yet.

As for the importing settings used in the implementation part for this thesis, they were left at default. Only when trying to use the NPC model with the **baseCharacter\_skeleton**, was the skeleton to use chosen to be the **baseCharacter\_skeleton** instead of UE4's default skeleton.

### 6.4 Finished animation logic and blueprints

This chapter will focus on what was implemented for the project. The implemented features do not include every aspect that was initially largely because proper documentation and tutorials for UE4 regarding the topics could not be found. This might be because UE4 is such a significant engine and is widely used by AAA-studios so the best techniques and workflows for

certain more advanced topics stay inside the companies and the people working on them might be restricted by NDAs and cannot publish tutorials or guides on everything that can be done. In the case of this thesis study, there were many parts that seemed hard or difficult to read, but no better alternatives for making the parts could be found. Many of the planned features would have also needed custom made 3D rigged models, but it was impossible to acquire them for the implementation part of this thesis due to time restrictions.

## **6.5 Implemented animation states**

What was initially planned for the project to was make a base character with animation states for being idle, moving, jumping and attacking. Additional elements were implemented, because some of the other planned features had to be discarded. These being, the use of **Child Animation Blueprints**. It was initially thought the animation logic could be inherited to other characters from Mixamo as they should be sharing the same skeleton, but at the time of importing the new characters to the project it became apparent that UE4 broke the new character models as it could not attach the previously added skeleton properly. Many different settings were tested following UE4's own documentation, but none of these worked with the importing of new Mixamo characters. Therefore, so as to not waste more time trying to make the temporary characters to work, the use of **Child Animation Blueprints** was abandoned in favour of polishing the main character's animation logic and retargeting the animations for the NPC instead.

### **6.5.1 Idle state**

The idle state means that when the character is not moving, it has a default standing and breathing animation to bring it to life. A completely still 3D model would seem dead.

It was desired for the main character to have a base idle where it would be standing and breathing. If the player took no action for a longer time, the character should make random extra movements, such as stretching or looking around. At the start of the idle state a timer would be started for

counting the time the player character stays still. These extra idle states would activate if there was no input from the player for a long time then they would play once then go back to base idle and the timer for doing an extra idle would then reset. The action would be stopped if the player started moving. The actual transition rule for going from the base idle to the extra idles is to check for two booleans, one for the player moving and another for the making the extra animations. The latter boolean is set to true after the idle timer is more than a set value, for example, ten seconds. The extra idles were implemented by using a **Conduit State**. It is an animation state node from which one of many choices connected to it can be taken by giving the states connected to it their own rules as to when they can be played. Figure 1 below illustrates the representation of the idle system.



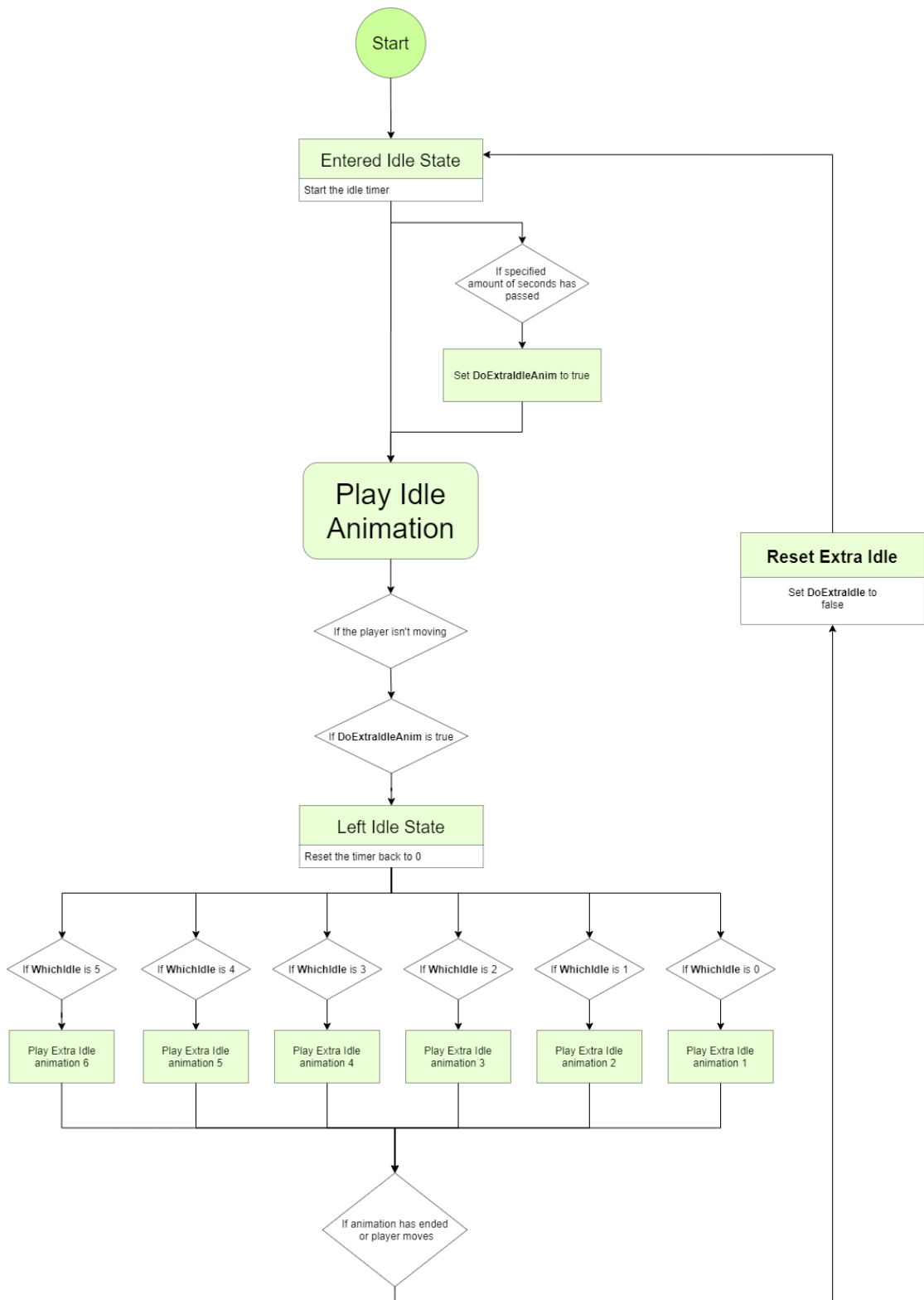


Figure 1. Idle system flowchart.

Figure 2 illustrates the idle system in the game project. The white arrows between the states illustrate the transition rules which can be opened by double clicking on them. The **IdleForLong** is a state conduit, distinguishable from a normal state by the symbol or three lines on the left side of the box.

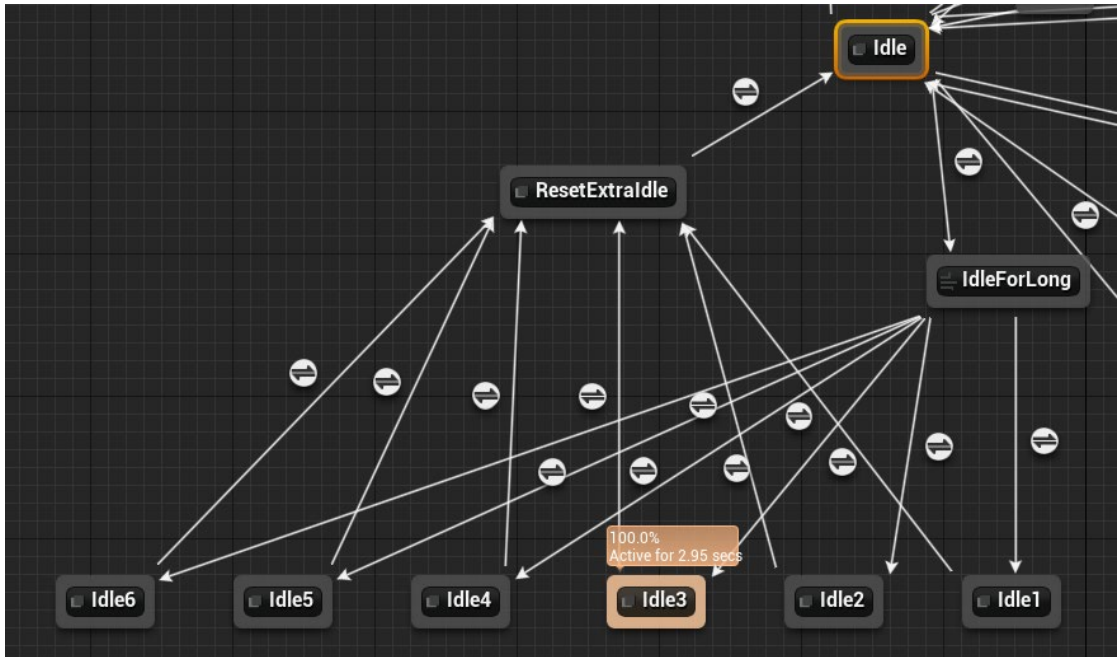


Figure 2. Idle system animation states in the project.

In Table 1, the Transition rules between the idle system states can be seen as they are in the project. However, only one of the extra idles is shown as they are all similar, the WhichIdle variable is checked against different hardcoded values from 0 to 5.

Table 1. Transition rules for idle system.

	<p>From Idle to <b>IdleForLong</b>.</p>
	<p>From <b>IdleForLong</b> to the first extra idle, <b>Idle1</b>.</p>

	<p>From <b>Idle1</b> to <b>ResetExtralidle</b>.</p>
	<p>From <b>ResetExtralidle</b> to the base idle, <b>Idle</b>.</p>

The extra idle played is chosen at random. This means it might be the same one played previously, or it might be a different one. During testing, the same animation was not played consecutively too many times for it to be noticeable for the player.

The idle animations are randomised by choosing a random integer from range of 0 to the number of extra idle animations. The maximum number is hard coded and set to 5 because no way was found to receive the number of states leaving from a state conduit by code. The idle states play depending on which number the integer variable **WhichIdle** is at that moment. It is randomly set every time the character goes back to base idle state. In the details panel for base idle, there is a menu for **Animation State Events**. There, a custom event **Entered State Event** is set to **EnteredIdleBC** to enable calling it whenever the state is entered. This makes it into an **AnimNotifiy**. It can then be called in the **Graph Editor** to add logic. In Figure 3 and Figure 4, the use of the custom event in the project is shown.

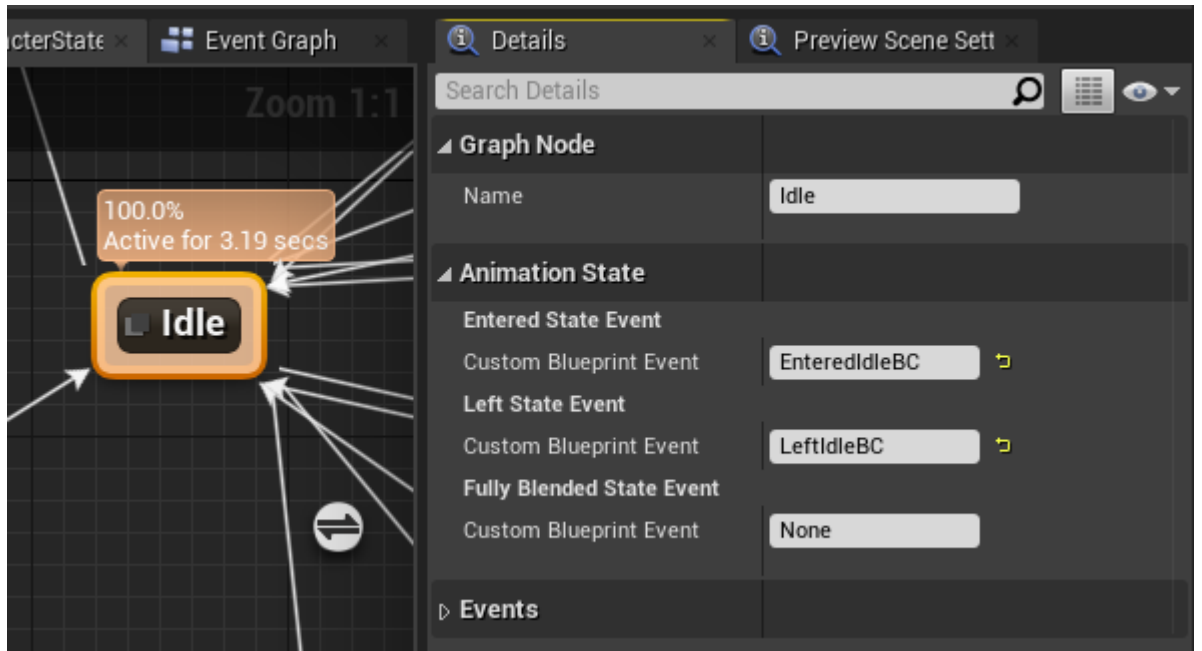


Figure 3. Custom events from Idle animation state set to enable using them.

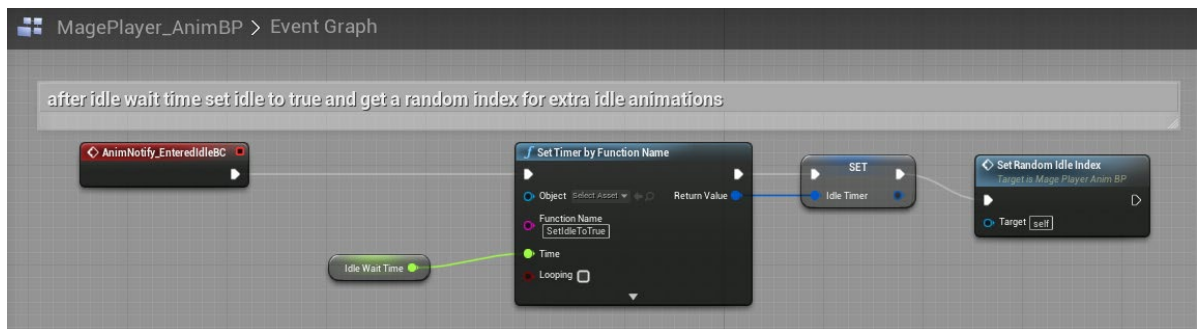


Figure 4. Calling EnteredIdleBC as an AnimNotify to add logic from the Event Graph.

The idle needs a boolean called **DoExtrIdleAnim** to be true to obtain permission to transition into the extra idle states. This boolean also needs to be set to false after having played the extra idle. After that, the timer is also reset back to 0. This is so that the extra idle animation is only played once, and the loop can then start again. In order to enable this, a new state was added after the extra idle states. This state is called **ResetExtrIdle** state and is used to call a custom event to enable the logic for resetting the idle system timer. The state itself plays the same animation as the base idle, but only for a minuscule amount of time and should not be able to be seen playing. At first, there was a custom timer for setting the boolean to false. It was called after leaving the base idle state, but this created logical errors, and set the boolean to false at the wrong times either stopping the extra idles completely or playing

them continuously with no regard to the idle system timer. In order to fix this behaviour, the **ResetExtraIdle** state was added and all the extra idles go to this state after having played. The **AnimNotify** for the **ResetExtraIdle** can be seen in Figure 5.

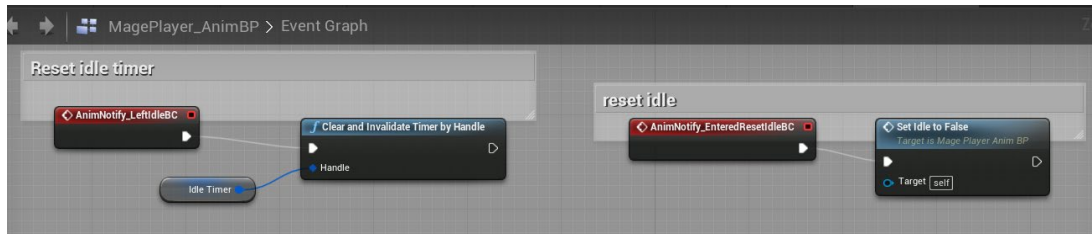


Figure 5. Reset idle AnimNotify for resetting the boolean for playing extra idles animations.

## 6.5.2 Movement state

The movement state for the character is a **Blend Space**. **Blend Spaces** are readymade components in UE4. By giving it the speed and direction of the character, it automatically blends between the given walk and run animations. Figure 6 illustrates the movement Blend Space, and Figure 7 shows how the **Movement Blend Space** is given its needed values in the **Anim Graph** of the **Player Animation Blueprint**.

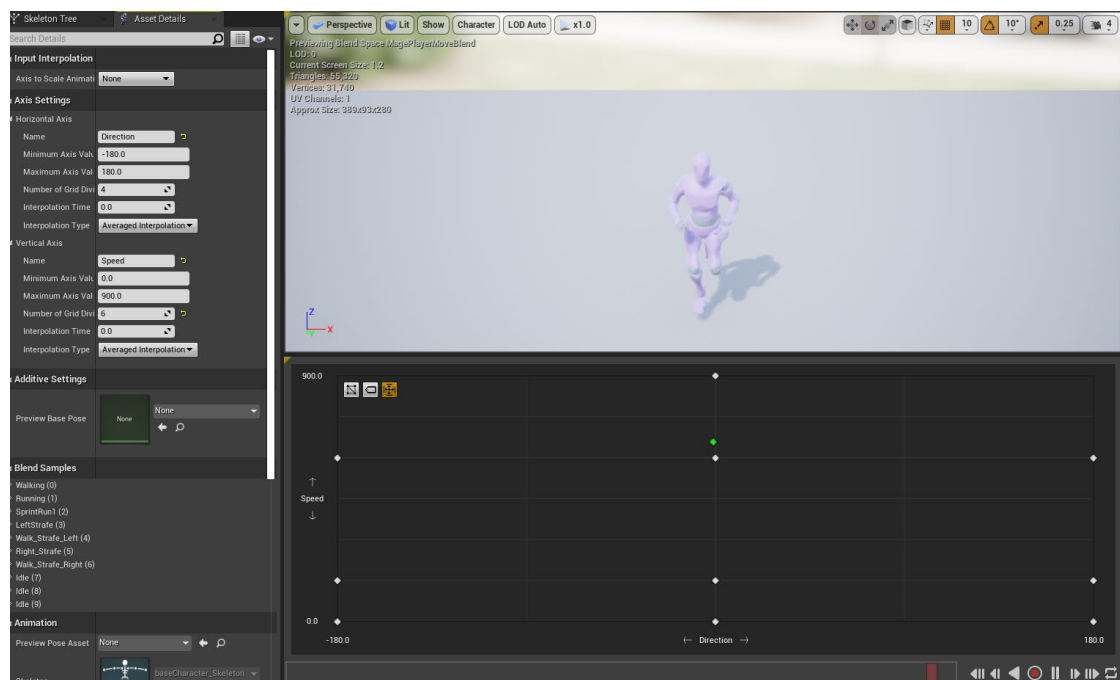


Figure 6. Movement Blend Space.

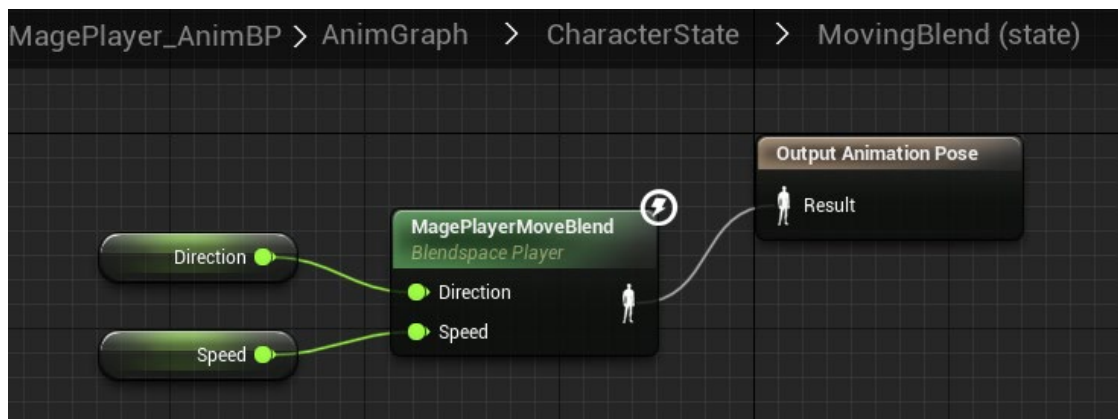


Figure 7. Movement Blend state in the Anim Graph.

The character transitions to the movement state if it is not jumping or in the air and its movement speed is more than zero. From the movement it goes back to idle state if the movement speed is zero and it is not jumping or in the air. Figure 8 below for clarifies this.

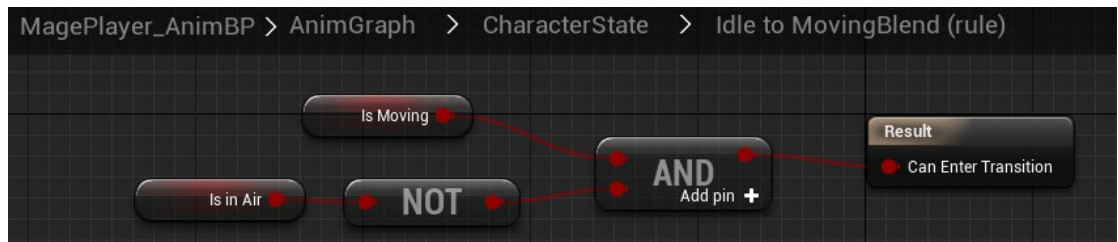


Figure 8. Transition between idle and movement state.

### 6.5.3 Jump state

As Chicken Quest is a platforming game jump is perhaps one of the most important features. As stated by Cooper (2019, 42), jumping in platforming is most fun when the character is controlled in an analogue manner and that the jump is as much of a challenge in programming, design and animation. And that has proven to be true. The jump has caused even more trouble than trying to inherit the **Animation Blueprints**.

The jump is controlled by input from the player. When the character jumps, it goes up in the air a somewhat slower than when it comes down. The character can also be controlled while in the air. These features are

preferences of the project team, but it also seems to be how many platformers implement the jump. If the jump comes down with the same speed as it goes up, it will feel like the character is floating. Also, the ability to control the character while it is in the air is a preference, many other platformers also give the player the freedom to control the character while it is in the air. It is not necessary to imitate real life in games and many prefer to give as much control to the player as possible. A game that does not respond to inputs from the player will soon start to seem broken and inferior. Figure 9 shows the script for the jumping logic.

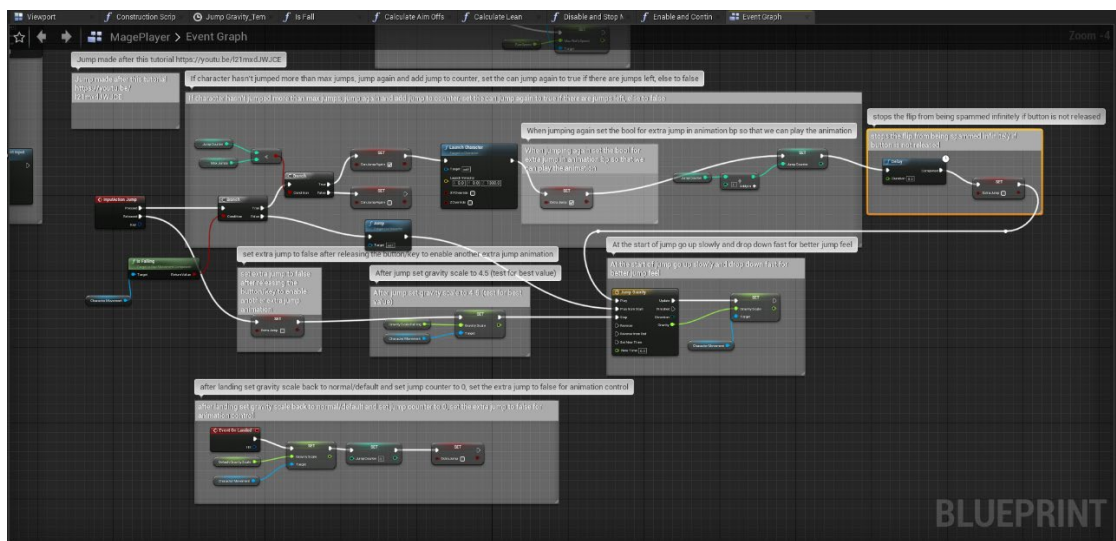


Figure 9. Jump script in the project.

As for the jump animation system itself, there were two jump systems implemented, one for jumping from stand-still and one for jumping from movement. After testing, what seemed to work the best for the game was to have states where the character is going up in the air, falling and then landing. There is also an option of making an extra jump. As such, for the both jump systems there are four stages. There is a limited number of extra jumps that can be performed if the character is in the air. The available extra jumps are reset when the character touches the ground again. Being in the air is also called the jump loop, and it is usually an animation where the character is posed as if it is falling but some parts of the model, for example, arms or clothing, are swaying slowly to simulate the wind.



The jump systems are controlled with five boolean values, three for the basic jumping and two for the extra jump. The boolean **IsInAir** is a variable received from the **CharacterMovement** component attached to the player. **IsInAir** is true when the character is not touching the ground. In addition, there are booleans **FallingDown** and **JumpingUp** for whether the character is falling down or going up. If the character is in the air and its Z-axis velocity is more than 0, the character is going up, and **JumpingUp** is set to true. Otherwise, it is false and the **FallingDown** is set to true. For the extra jump, the two booleans are **DidExtraJump** for the player jumping again while already in the air and **CanJumpAgain** for confirming if the character can be allowed to do the extra jump. The values for these booleans are received from the **Player Blueprint**. Figure 10 illustrates the jump system.

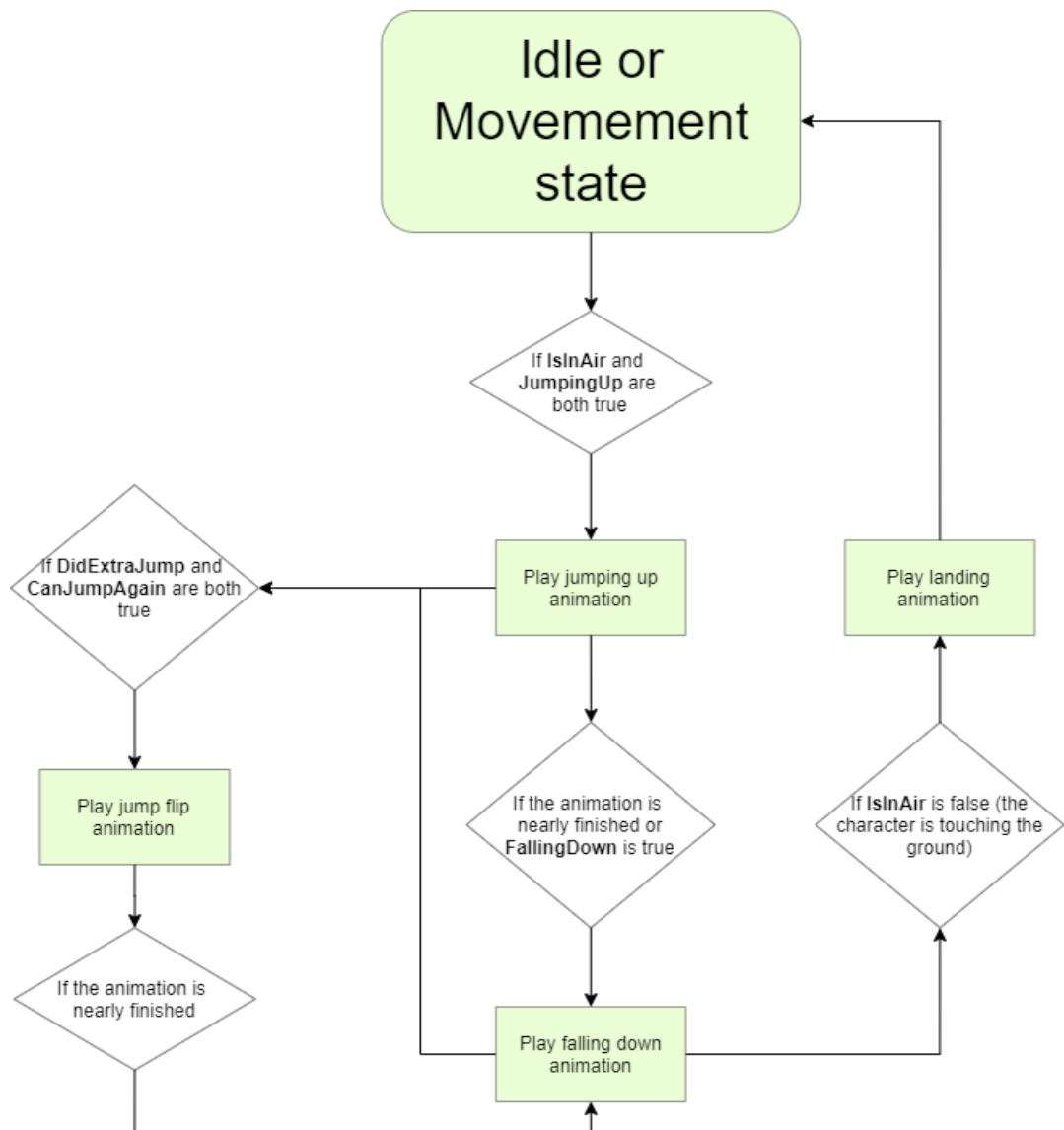


Figure 10. Jump system in the project.



The single most difficult part of the jump in the project was the extra jump or the jump flip, as the animation sequence for it is a flip in the air. The idea behind it is simple. Using an integer to count the jumps, **jumpCounter**, it is tested if the counter is equal or more than the maximum possible jumps that can be made. If this is the case, the player must not be allowed to make any more jumps until the character has hit the ground and the counter has been reset.


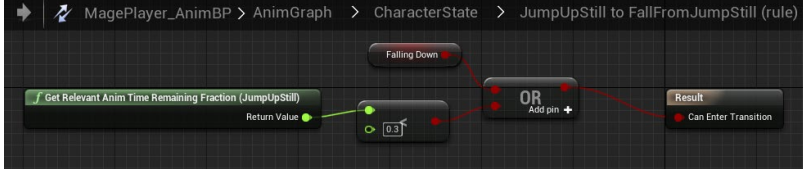

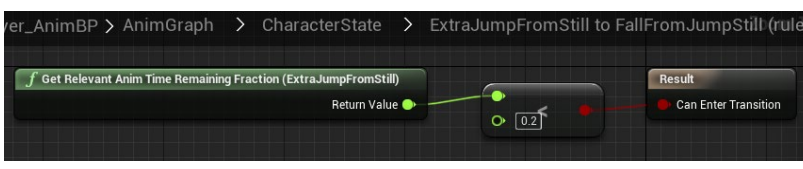

Instead of a jump start animation where the character squats down and springs off, it was decided to use an animation where the character launches directly into the air because if there is a squat, the animation for the squat is played when the character is already in the air. Because the game is supposed to be a 3D platformer, it would not make any sense to delay the actual jump until the animations to finish playing. Also, speeding up a squat animation to keep the responsiveness would look unrealistic. Because of this, it was decided to leave out the squatting part.

Many problems arose while working with the jumping states, mostly due to logic failure. These problems were fixed by adding transitions between the states or adding checks for seeing if something undesired was happening which might hinder the proper behaviour.

When starting to test the jumping from being idle and from movement, the animations for jumping from being idle failed to be played. This was because in the transition rule from the idle state to jump. It was testing if the character was moving, if it was not, the transition could be made. This was possible by testing if the character had any speed. Every time the result would be positive, the character had speed, thus it was moving. The reason behind the problem was that when jump was pressed, the character started going up, and as the character is using UE4 readymade components for the character controls, the movement of going up and down is also regarded as speed, vertical speed. As so, when jump was pressed, the character would go into the movement state. This was bypassed by adding a new transition rule from idle to movement. The character could now go from idle to movement if it was moving, but not in the air.

A more visual problem was that if during the jump from being idle the player started giving input for the character to move, the jump animation would play to the end, transition to the landing animation and the character would still be playing the landing animation, when it was in fact already moving. This looked like the character was sliding along the ground. This behaviour was easily fixed by adding a new transition from the jump fall to the movement state. If the player started moving after jumping from being idle, right at the moment the character touched the ground after the jump, it would transition to the movement state and play either walking or running. Table 2 and Table 3 illustrate the transitional rules used with the jump states. The behaviour of the jump systems and the transition rules for them are mostly the same.

Table 2. Transition rules for jumping from idle.

	<p>Transition rule from the idle state to the jumping up state.</p>
	<p>Transition rule from the jumping up state to the falling down state.</p>
	<p>Transition rule from the falling state and the jumping up state to the jump flip state is the same for both.</p>
	<p>Transition rule from the jump flip state to the falling state.</p>
	<p>Transition rule from the falling down state to the landing state.</p>

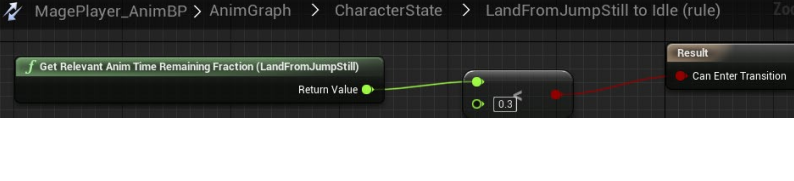
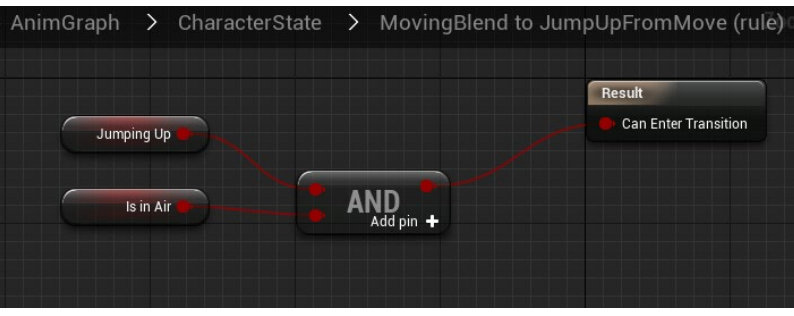
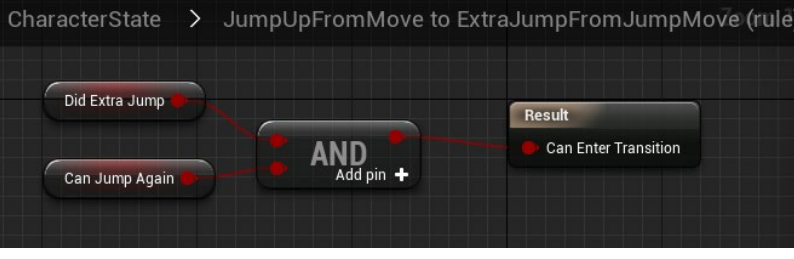
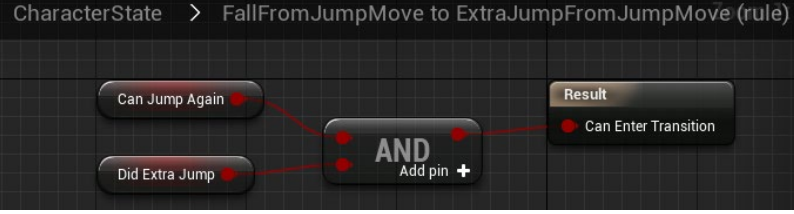
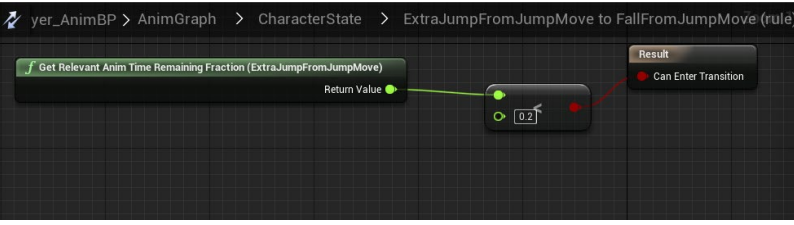
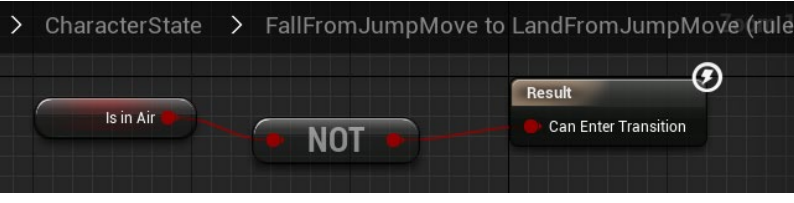
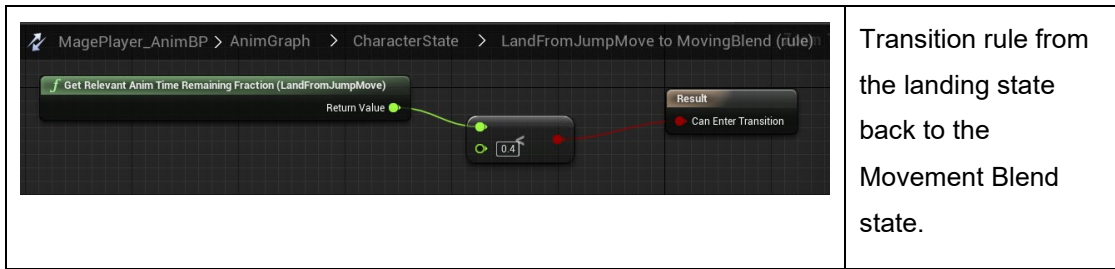
	<p>Transition rule from the landing state back to the idle state.</p>
--	---

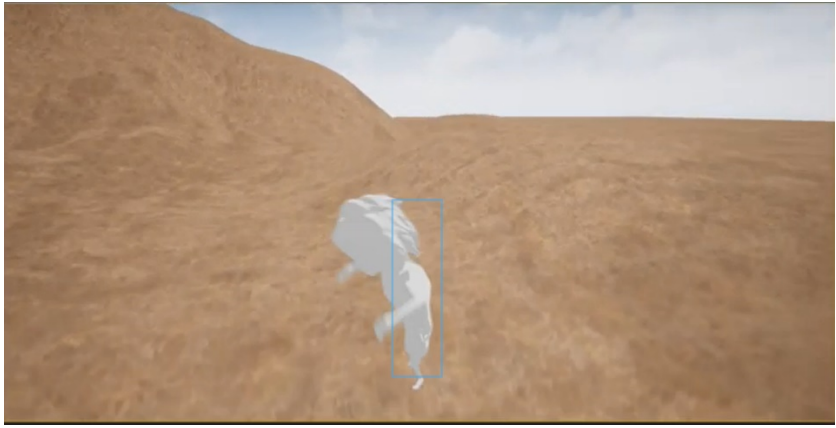
Table 3. Transition rules for jumping from movement.

	<p>Transition rule from the Movement Blend state to the jumping up state.</p>
	<p>Transition rule from the jumping up state to the falling down state.</p>
	<p>Transition rule from the falling state and the jumping up state to the jump flip state is the same for both.</p>
	<p>Transition rule from the jump flip state to the falling state.</p>
	<p>Transition rule from the falling down state to the landing state.</p>



A significant problem with the jumping animations was the jump flip, or the extra jump. The logic itself for it was simple enough and proved functional after testing. The problem was the location of the 3D model during the jump flip animation and it consumed a great amount of time to identify this. In the project, UE4's own component for the **CharacterMovement** was used to save time and effort as many required features were already incorporated in it such as input and crouching, movement and checks for falling. That can be really beneficial unless specific custom logic is required for the game to work.

With the jump flip, the problem was that when the jump flip was performed, during the animation, the character model would always behave weirdly and be positioned either too high or too low. First, the cause was thought to be the animation itself. The animation was downloaded from Mixamo and it was discovered that in the jump flip animation, the animation itself positioned the model high into the air instead of its local origin point. Therefore, the animation had to be modified so that the flip would not move the model but would happen in a fixed place. This was important because the character is moved with code, and when the animation was moving the model as well it was too high, and the actual collision for the character would not move with the animation. Figure 11 provides a visual representation of this. The blue rectangle drawn on top of the images shows where the collision and centre point of the character is in reality during the jump.



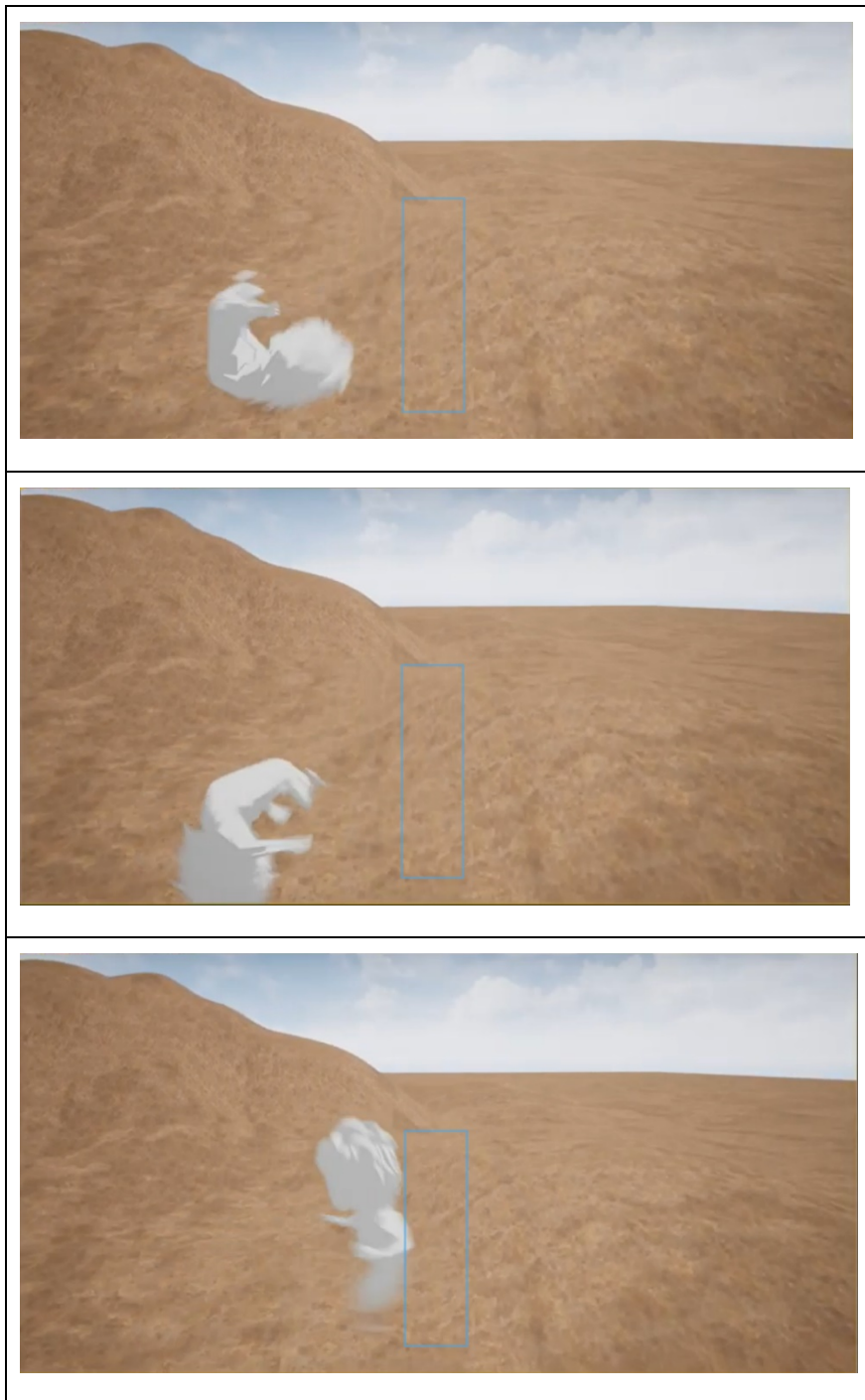


Figure 11. Character model position during jump flip.

As such when playing the game, the player would see the character high in the air, but the collision would in reality be in a different place. This would make the character look like it hit seemingly invisible obstacles and cause other unexpected behaviour. When this problem was identified, it was also noticed that the collision for the character was too large during the jump animation, **anim notifies** were used to control it during the jump animation to scale it to be a suitable size. Figure 12 shows how the collision is detached



from the model, and Figure 13 illustrates how the collision was halved. The black capsule is the collision component drawn using the debug tools in UE4.



	<p>At the start of the jump the character is where the collision is.</p>
	<p>When jumping up the character moves higher than the actual collision.</p>
	<p>After having landed from the jump the character stays higher than the collision.</p>

Figure 12. Character collider during the jump animation.

	<p>At the start of the jump the collision height is the height of the character.</p>
--	--



	<p>When jumping up the collision height made smaller.</p>
	<p>When falling after jump the collision is returned to its original size.</p>

Figure 13. Halving the height of capsule component.

After modifying the collision and the position of the model in the animation, the problem, however, still existed. After using debugging tools to draw the collision, it became apparent that the model would either be slightly above the collision or slightly below it. The reason for this behaviour remains unknown.

The next possible solution that was tested to fix the behaviour was to set the model to where the capsule is through code for the duration of the jump flip animation. This worked, but it was visually quite unattractive. The model was instantly moved to the new position and this transition was plain to see as well. Figure 14 represents visual explanation for this.

	<p>Just before the jump flip animation starts.</p>
--	--







	<p>Instantly moving the model to the collision at the start of the jump flip.</p>
	<p>After moving the model and the jump flip is playing.</p>
	<p>Instantly moving the model to the collision at the end of the jump flip animation.</p>
	<p>Right after the model has been moved to the collision position at the end of the jump flip animation.</p>

Figure 14. Moving the character model to the collision position during the jump flip.

Lerping the model to the new position would look visually more attractive since the movement would be smooth instead of a sudden disappearing and reappearing. In order for the player movement logic to remain independent

from the animation script, the moving of the model had to be done in the **Animation Blueprint**. As such, when trying to implement the lerp for the model, it was discovered that the lerp function in UE4 needed an alpha value that would grow from 0 to 1 over time, and that this alpha value should come from a timeline component. However, these timeline components are not allowed in the **Animation Blueprint**. This is how UE4 works and it cannot be changed, at least for the time being.

Making a custom timer for this was also tested, but the result was the same as before trying to lerp the model to the correct position. It would go over or below the actual collision. As such, the version where the model instantly moves to the proper position for the duration of the jump flip was kept. It was more important to have the collision and model be in their correct places rather than the animation to look perfect.

It is even stated by Cooper (2019, 42) that gameplay wins over the most beautiful and fluid animations if they interfere with gameplay, and such animations will be cut or scaled back.

To fix many of the problems in the jump animations and behaviour, custom 3d models and animations for the characters are needed. Another solution would be to not use the UE4's own character component, but to make a custom one for the main character.

#### **6.5.4 Attacking state**

The player character can also attack. It has different spell attacks and two different melee attacks, one when a weapon is equipped and one for when no weapons are equipped. It possible for the character to attack from idle and movement states. To attack the character must be on the ground, if it is in the air it cannot attack. For the project this is okay as there is not going to be a lot of combat and no aerial combat at all. Melee and spell attacks are both implemented using **Conduit States**.

For the melee attacks after the **Conduit State**, it is tested whether the character has a weapon equipped or not, and if it is moving or still. Based on the result the correct melee attack is chosen. After the animation has played

the character transitions back to the movement state and from there to idle it is not moving anymore. Figure 15 below illustrates the melee attack system.

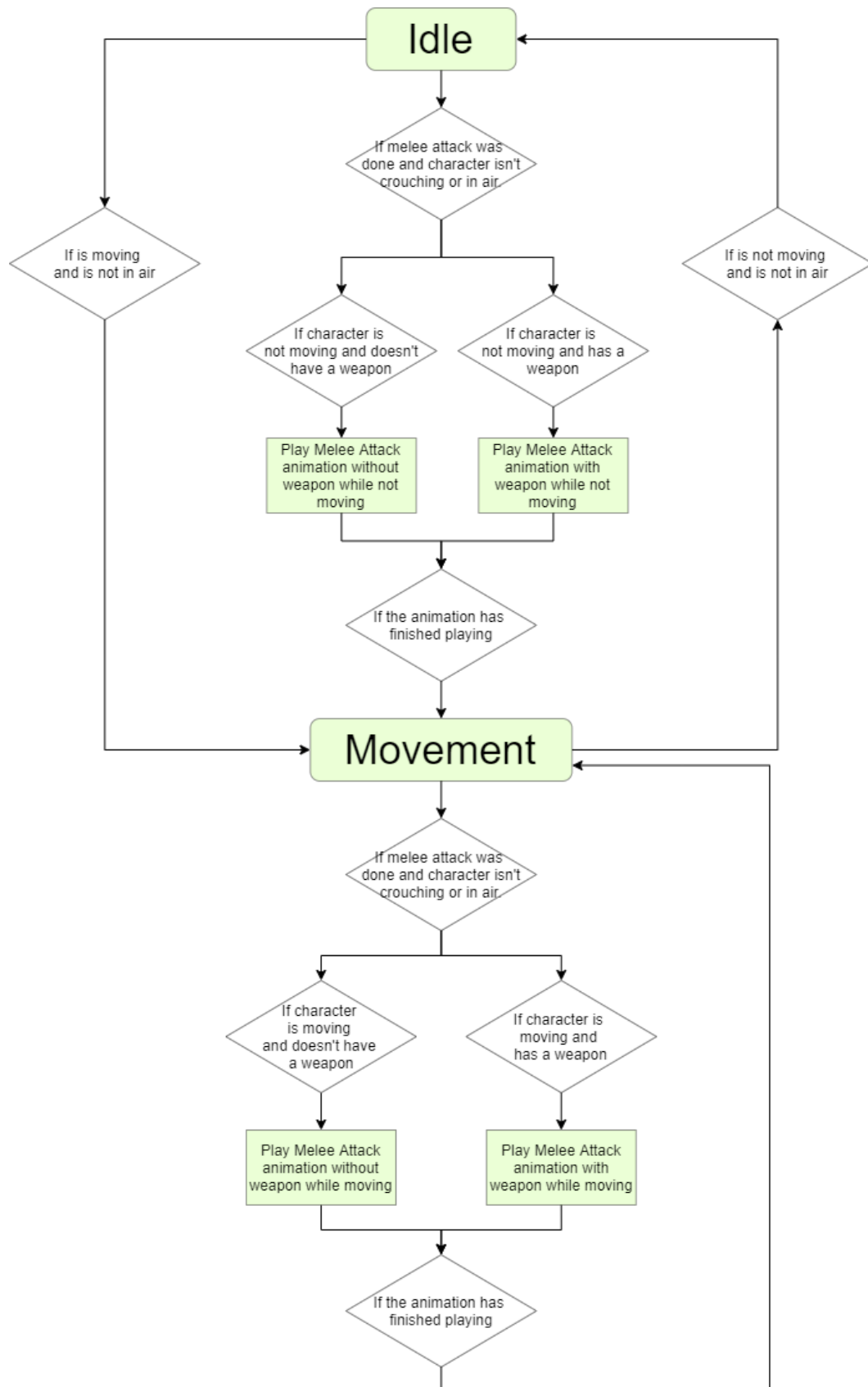


Figure 15. Melee attack system.

The spell attacks work similarly. From the **Conduit State** multiple different spells are available. The correct one is chosen by getting an enumeration for the used spell from the **Player Blueprint**. Figure 16 below illustrates the spell system. Once the spell animation is over the character transitions back to the movement state the same way as in melee attacks.

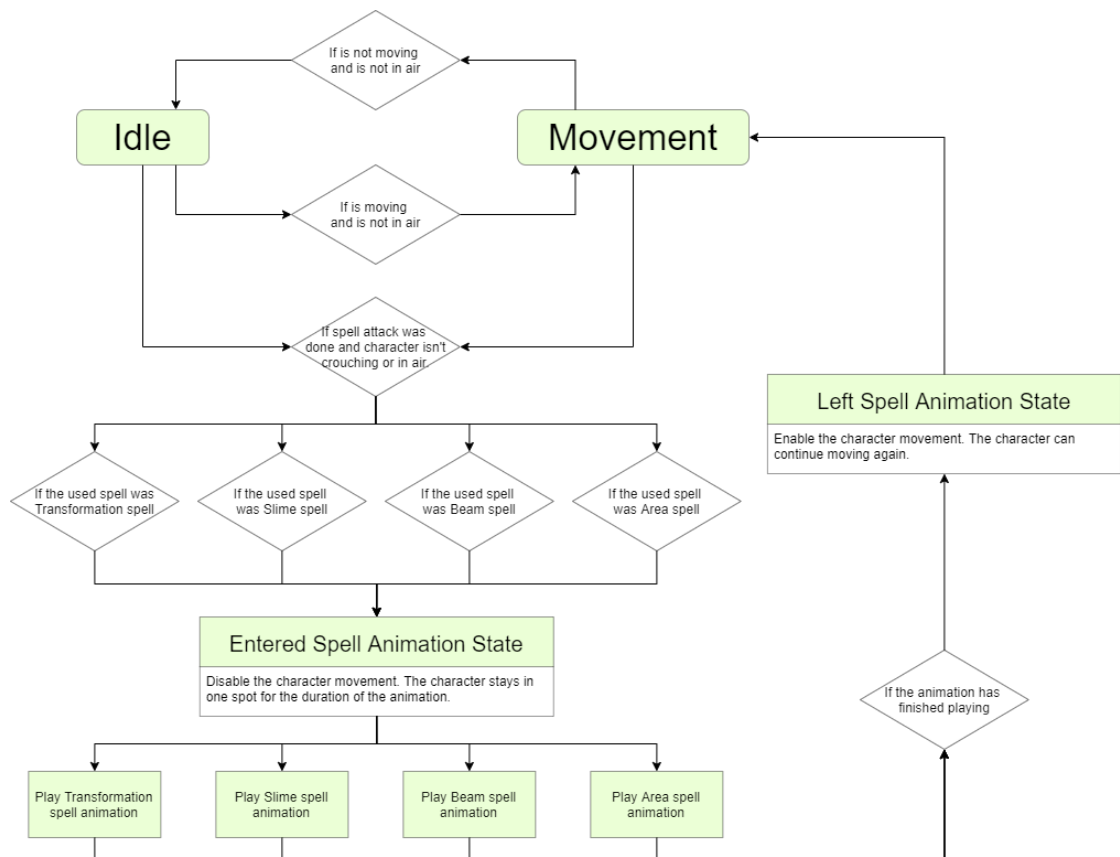


Figure 16. Spell attack system.

### 6.5.5 Crouching state

The player was not originally meant to crouch, but it was decided to add crouching, in case there came the need to make obstacles that required the player to go under them. The character crouches if the crouch button is being pressed. This input is checked in the **Player Blueprint** and checked in the **Animation Blueprint**. Script for the input is seen in Figure 17.

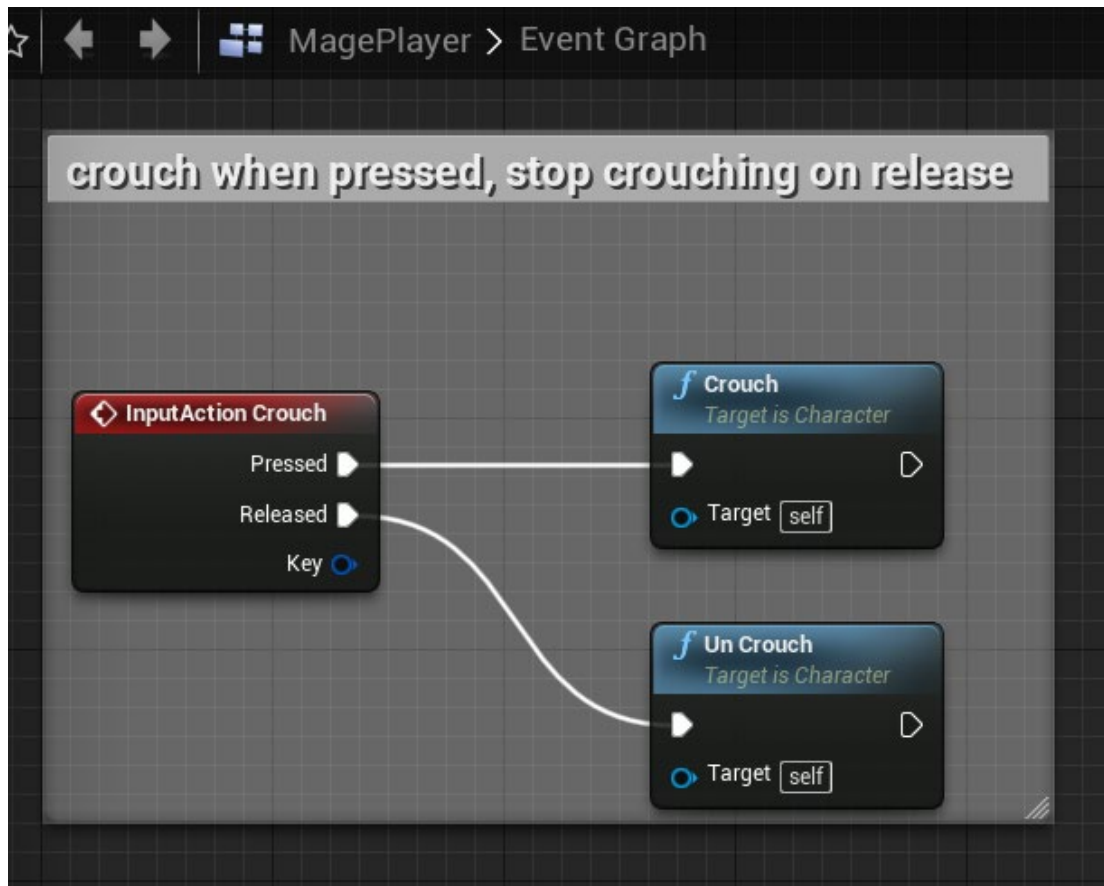


Figure 17. Input for crouching.

Crouching has three states. Starting the crouch (**StartCrouch**), being in the crouch (**CrouchBlend**) and ending the crouch (**EndCrouch**). Starting the crouch plays an animation where the character crouches down, during the crouching down movement gets disabled and stopped when the character enters the state. Movement will be enabled after the crouching down animation has left the **StartCrouch** state. As such, when entering and leaving the **StartCrouch** state **Anim Notifies** are being used to disable and enable movement. During the crouching **CrouchBlend** state is being played. **EndCrouch** state works the same way as the **StartCrouch**, the animation for it is one where the character stands back up. During **EndCrouch** movement is disabled and stopped. The **CrouchBlend** state is a **Blend State** where the character is either crouched still or moving in crouched state. The state takes in the speed and direction of the character. From the speed it checks whether to use walking animation or idle crouch animation. The direction can be used to blend in leaning to sides or crouch walking by facing front, but moving towards the sides, also called strafing. Figure 18 and Figure 19 show the crouching blend implemented for the project.

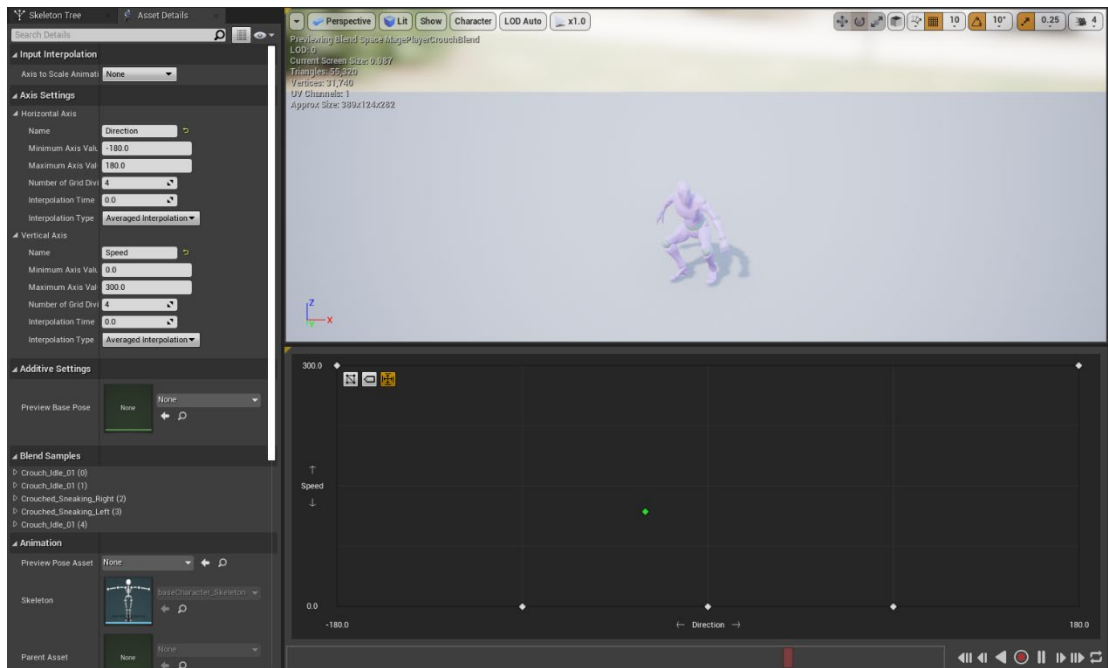


Figure 18. Crouch Blend.

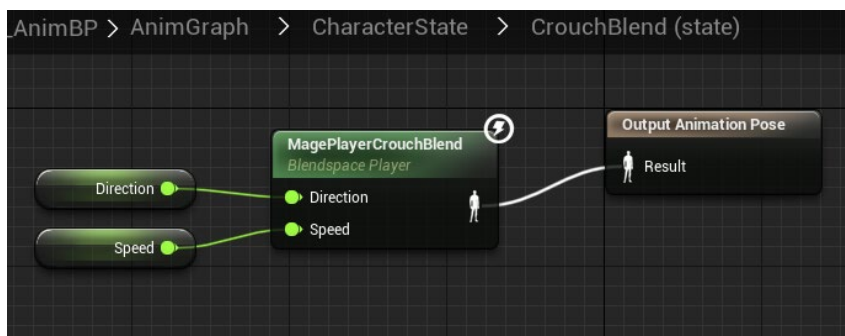


Figure 19. CrouchBlend state in the Anim Graph.

During the crouching the camera would suddenly teleport a notch lower, this is part of UE4's readymade features when using crouching from the **CharacterMovement**, the movement of the camera was made slightly smoother by selecting the **Spring Arm** the camera is attached to, in the **Player Blueprint**, and enabling **Camera Lag**. The settings for the **Spring Arm** are shown in Figure 20. Other options for better camera movement during crouch would be to make a custom crouch system but it was not possible at the time.

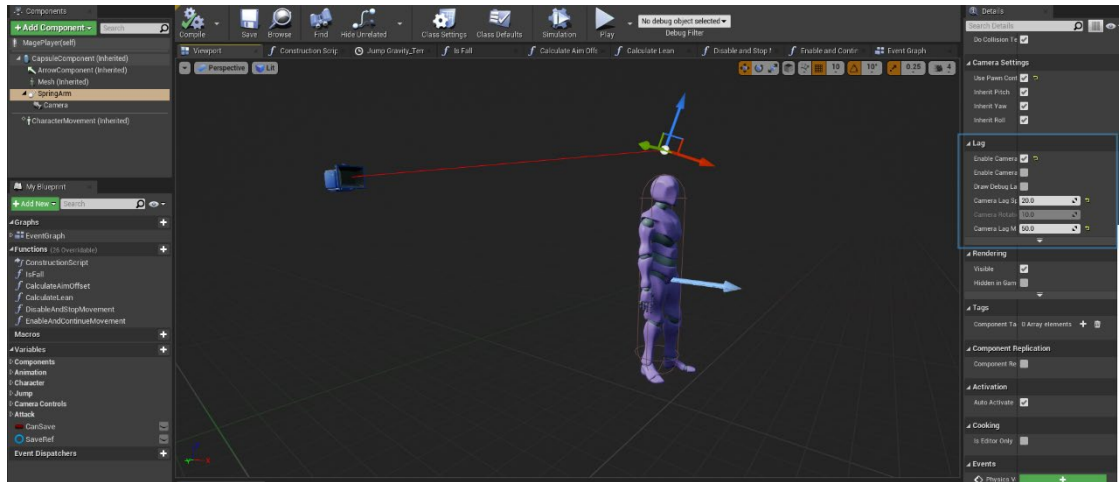
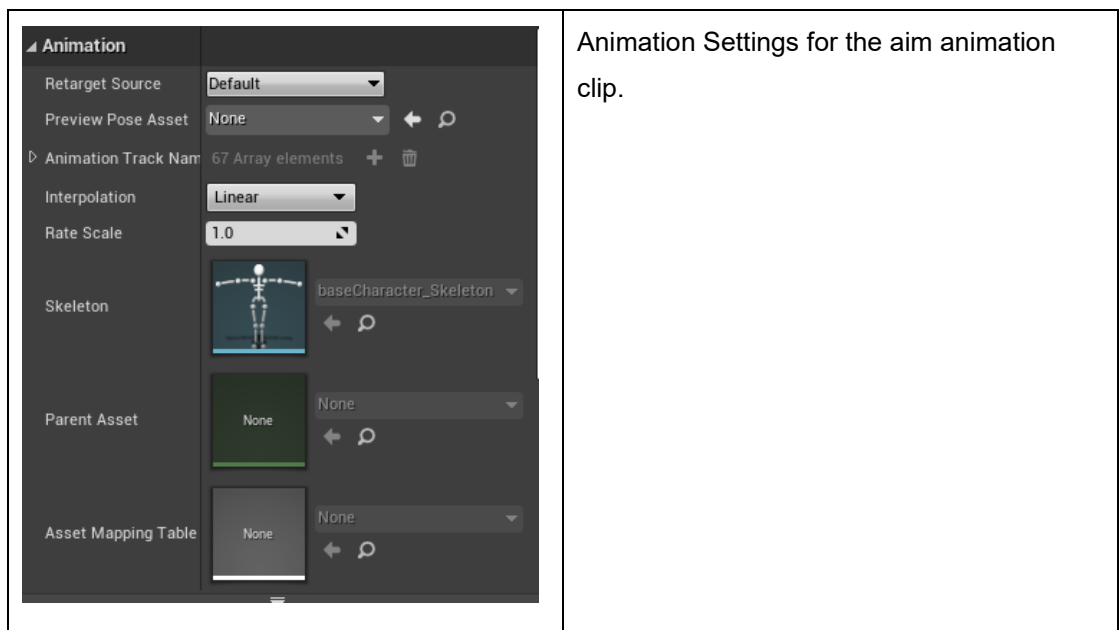


Figure 20. Spring arm settings to smooth camera movement during crouch.

### 6.5.6 Aim offset

**Aim Offset** is a kind of **Blend Space** meant solely for the use of blending aiming animations, for characters that can use a weapon. It can also be used to create animation blend where the character's head turns toward the direction the player is turning the camera to. For **Aim Offset** it is recommended to have 9 animations or poses, where the character looks to the different direction for the result to be satisfactory. This is because there are 9 main directions to look to: the front, up, down, left, right, up left, up right, down left and down right. The setup for each of these animations needs to be set as seen in Figure 21 below.



Animation Settings for the aim animation clip.



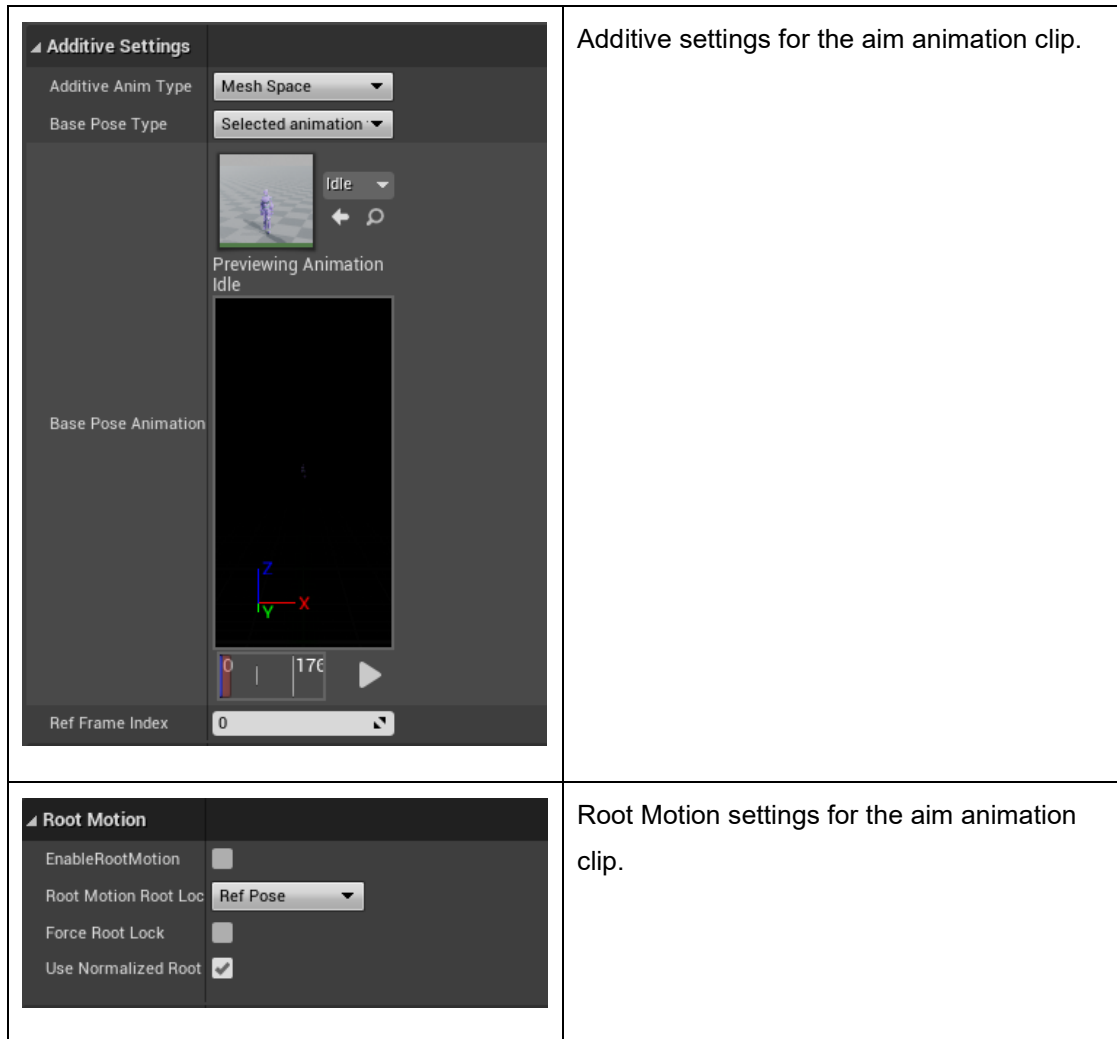


Figure 21. Settings for animation to be used in aim offset blend.

In the **Asset Details** panel in additive settings the **Additive Anim Type** is set to **Mesh Space** and **Base Pose Type** to **Selected animation**. In the thesis that would be the base idle animation. These settings allow for the head turn animation to be added to the idle animation, instead of having to also create 9 different idles with the head turns.

During the implementation of the **Aim Offset** it was noticed that using animations where the head was turned to a good direction, but the rest of the animation was very different from the base idle animation the blending of the **Aim Offset** animation would not work as was supposed to. For example, the arms of the character would be stiffly raised up instead of being relaxed by the character's sides, the model would also be disfigured. This was fixed by making custom animations from the base idle animation by making the head turn to where the character was supposed to look at and then exporting only



that one frame as its own animation. Figure 22 shows the wrong behaviour and Figure 23 the correct one.



 A screenshot from a 3D software interface showing a purple humanoid character model standing upright and looking directly forward. The character is positioned in the center of a grey ground plane against a light grey background. The software's interface, including a top toolbar and a bottom status bar, is visible.	Looking straight ahead.
 A screenshot from a 3D software interface showing the same purple humanoid character model in a dynamic, crouching pose, looking towards the left side of the frame.	Looking left.
 A screenshot from a 3D software interface showing the purple humanoid character model in a dynamic, jumping pose, looking directly upwards.	Looking straight up.
 A screenshot from a 3D software interface showing the purple humanoid character model in a dynamic, jumping pose, looking upwards and to the right.	Looking up right.

Figure 22. Aim offset wrong behaviour.


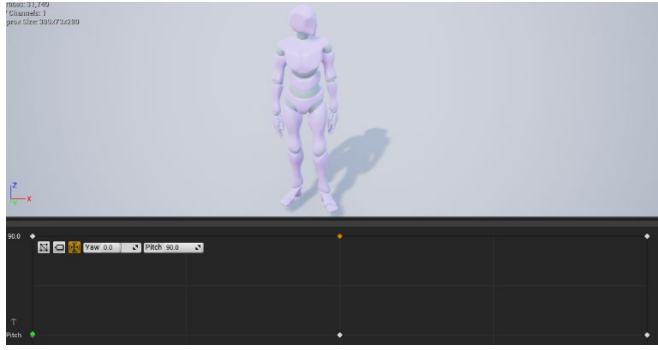

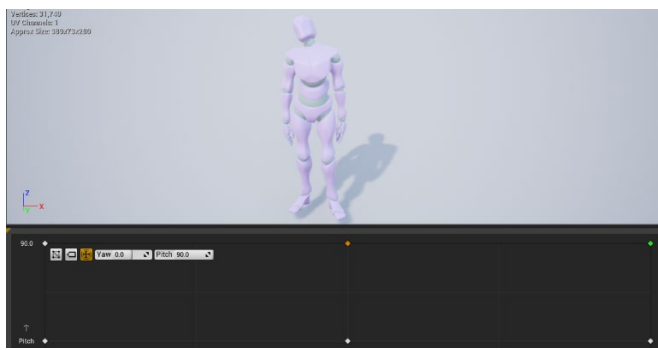
	<p>Looking straight forward.</p>
	<p>Looking left.</p>
	<p>Looking straight up.</p>
	<p>Looking up right.</p>

Figure 23. Aim offset correct behaviour.

## 6.6 Retargeting animations to another character

UE4 has inbuilt **Animation Retargeting** tools for sharing animation assets between different characters. **Animation Retargeting** is a feature that is

meant to be used to share animations between differently proportioned characters that use the same **Skeleton asset**.

It is possible to prevent animated skeletons losing their proportions or becoming deformed when using animations from a differently shaped character via retargeting. Animations can also be shared between characters that use different **Skeleton assets**, if they have a similar bone hierarchy and share an asset called a **Rig** to pass animation information from one **Skeleton** to the other by using **Animation Retargeting**. (Unreal Engine 2020b.)

For the project **Animation Retargeting** was used because the two characters chosen from Mixamo could not be identified by UE4 to share the same skeleton. The reason for this is unknown as when inspecting the models with Blender the bone hierarchy and naming conventions were the exact same. After trying to re-import the second character multiple times with different settings and UE4 flipping parts of the character to point wherever and disfiguring it the idea of using the same skeleton had to be given up. Retargeting was used to share the set of animations meant for the main character with the NPC. Figure 24 shows the weird behaviour.

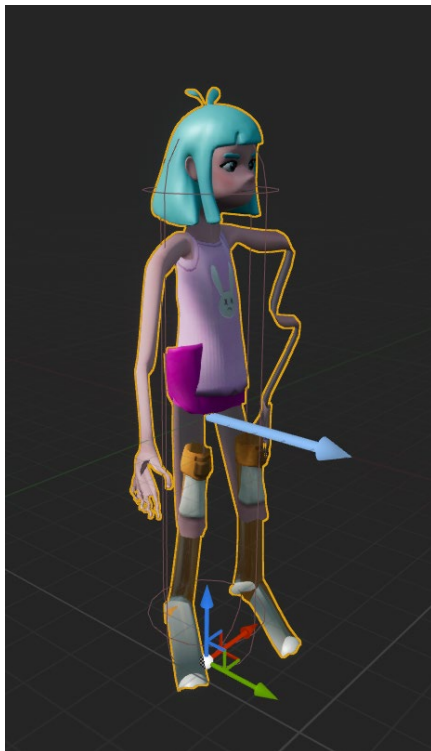


Figure 24. Weird behaviour of the NPC model when trying to use the same skeleton as the base character.

Alternative way for acquiring animations for the NPC would have been to individually download them from Mixamo, but this would have taken hours of more work and so was an option left to be. As stated in UE4 documentation (2020b) animations are bound to a **Skeleton asset**. The **Skeleton asset** is a list that consists of bone names and hierarchical data. Furthermore, the initial proportions, which define the **Skeleton asset**, of the original **Skeletal Mesh** are stored in the **Skeleton asset**. This data is stored as bone translation data. Only the bone's translation component is targeted by the retargeting system, and the bone's rotation is obtained from the animation data.

This is the reason why different sized meshes trying to use the same skeleton directly end up disfigured. In the project the NPC is slightly shorter than the base character so at first it was tested to retarget the animations to the same skeleton following UE4 documentation and then use them with the NPC, but this also made the NPC behave weirdly. Figure 25 shows how the NPC behaved unsatisfactorily.

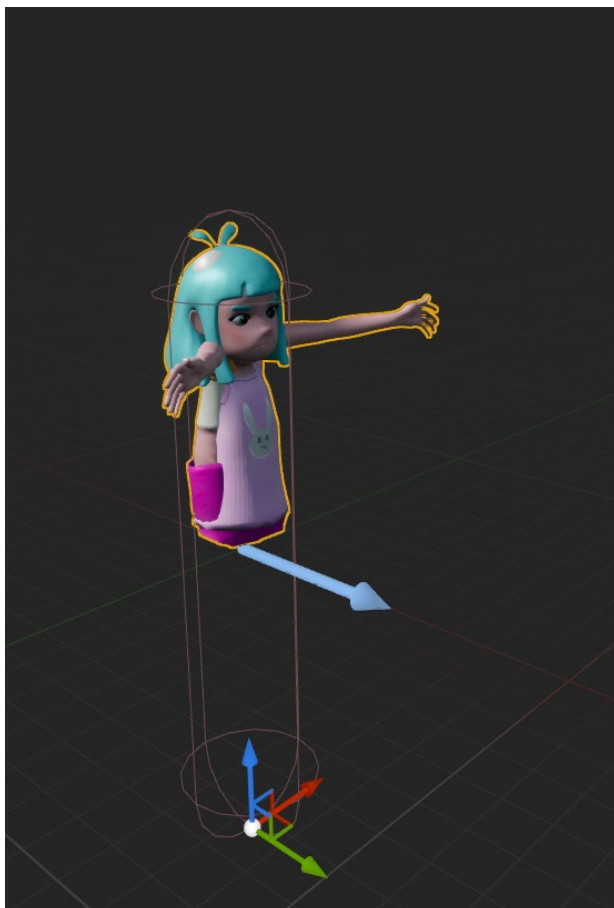


Figure 25. NPC with animations retargeted to the same skeleton.

The final solution was to import the NPC with its own skeleton and use that skeleton with it. After which, the animations were retargeted for the new skeleton. This worked very well and saved hours of work finding and downloading individual animations for the NPC.

To actually retarget the animations what first needed to be done was to set up the base skeleton correctly. This was done by opening the skeleton wanted to use as the base for the new one and using the **Retarget Manager** to set up a rig for it. In the project this base skeleton is the base character's skeleton. Figure 26 and Figure 27 show the **Retarget Manager** and the settings used.

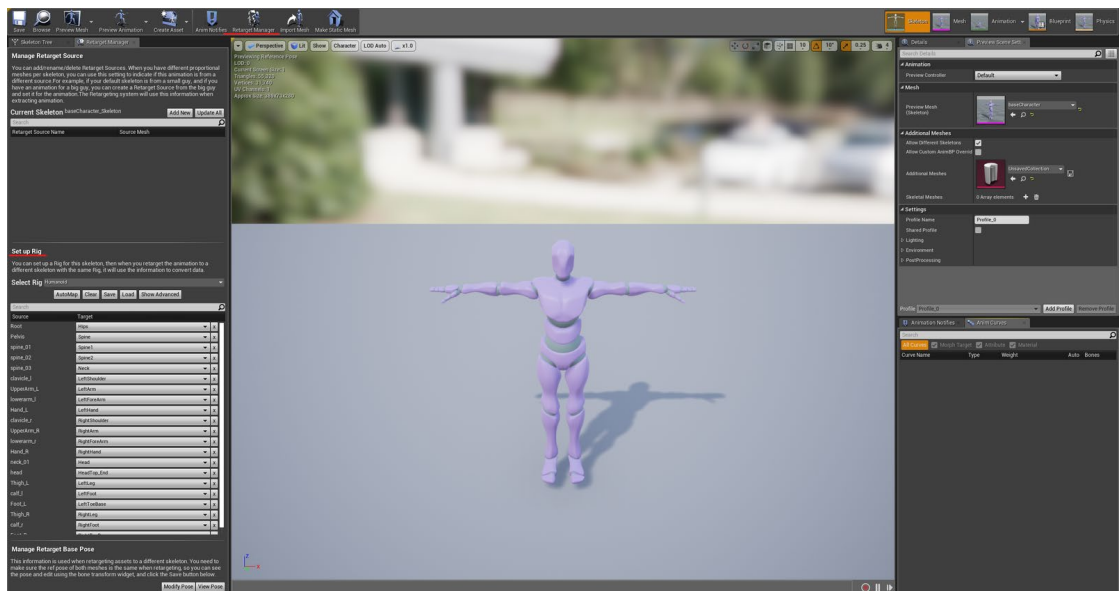


Figure 26. Base skeleton with Retarget Manager options.

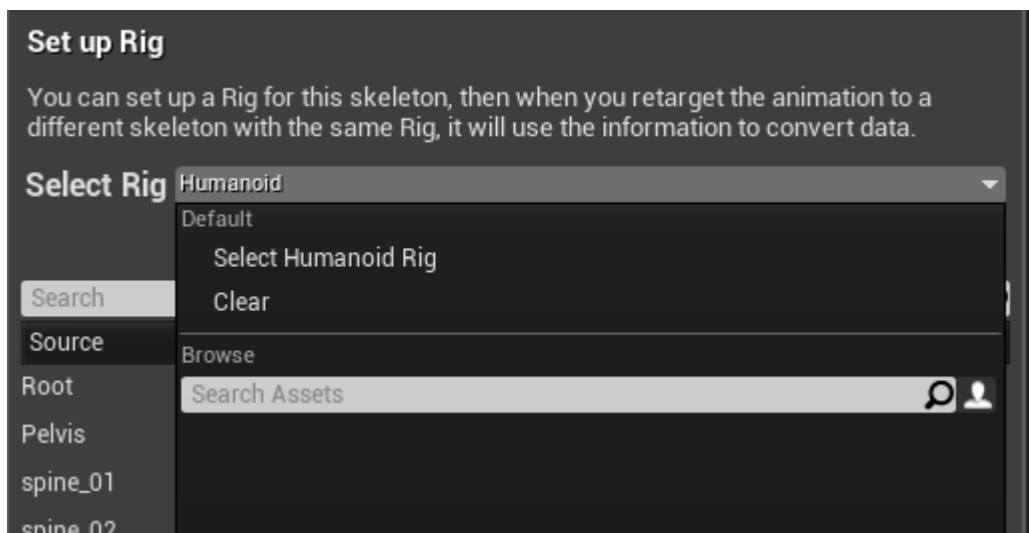


Figure 27. Rig options in the Retarget Manager.

When retargeting a humanoid character, the new one also needs to be a humanoid. As such, a **Humanoid Rig** was chosen. After this selection has been done the list for retargetable bones becomes visible and the correct bones can be chosen. Figure 28 shows the **Set up Rig** panel and how the bones were set to their corresponding targets.

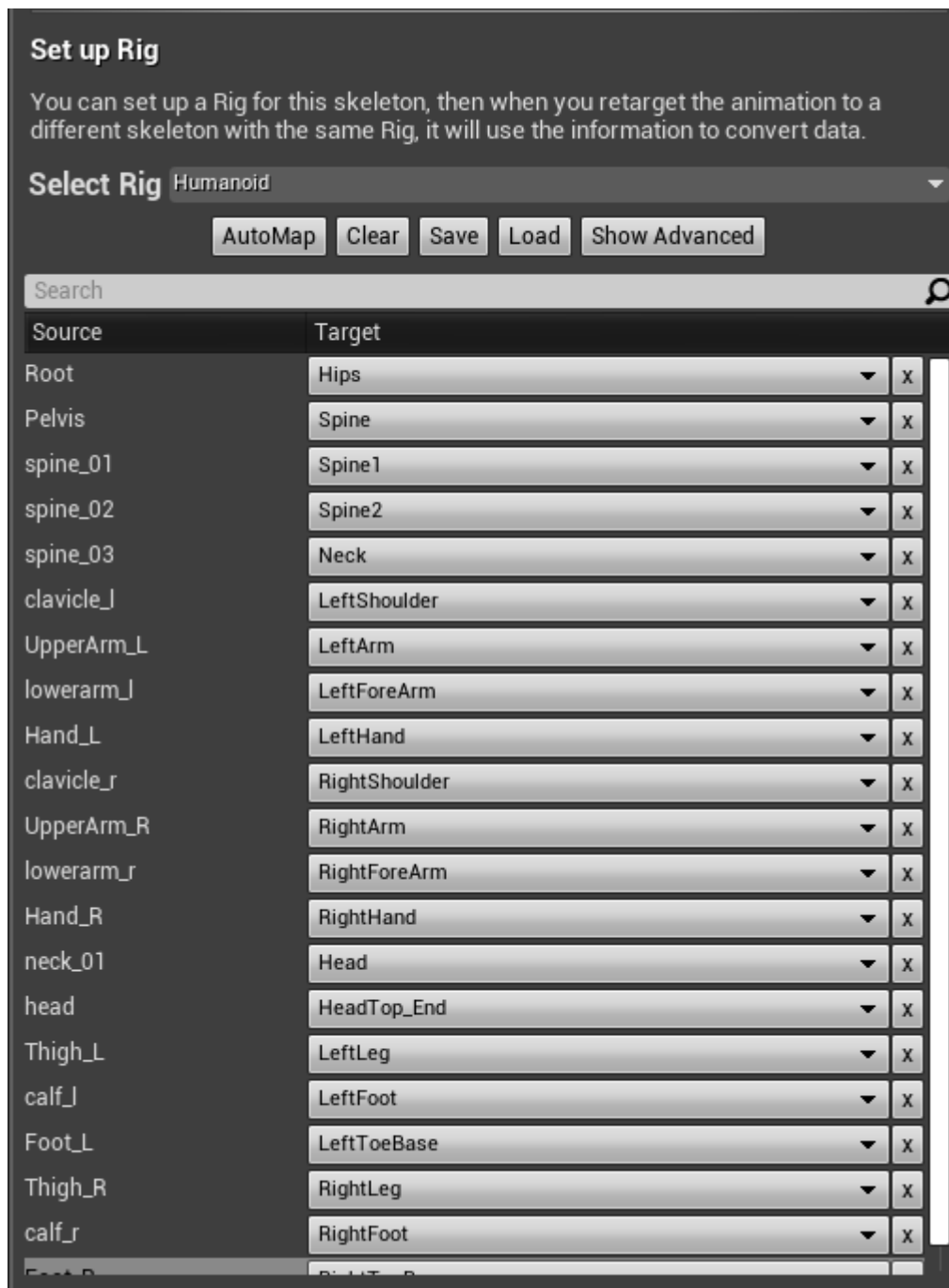


Figure 28. Retargeting bone names in the Retarget Manager.

The same settings must be set for the new skeleton as well. Very similarly to the base skeleton, in the **Retarget Manager** selecting a new **Humanoid Rig** in the **Set up Rig** tab. Then setting the corresponding bones in the list that is underneath. The corresponding bones would be those that are in the same place on the new skeleton as the ones in the base skeleton. Clavicle left could, for example, be called Left Shoulder on the new skeleton. Figure 29 shows the NPC's retargeting.

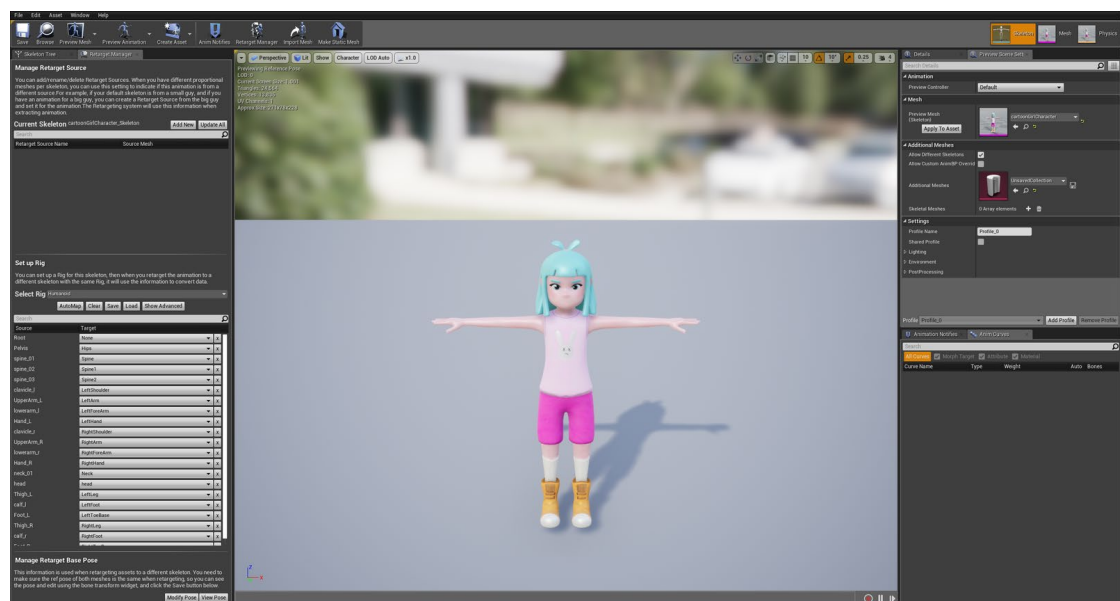


Figure 29. Retarget options for the NPC.

When the setup is done for the skeletons, what remains is choosing the animations to retarget in the **Content Browser**. This happens by right clicking and choosing to retarget assets from the tab that opens. Shown in Figure 30.

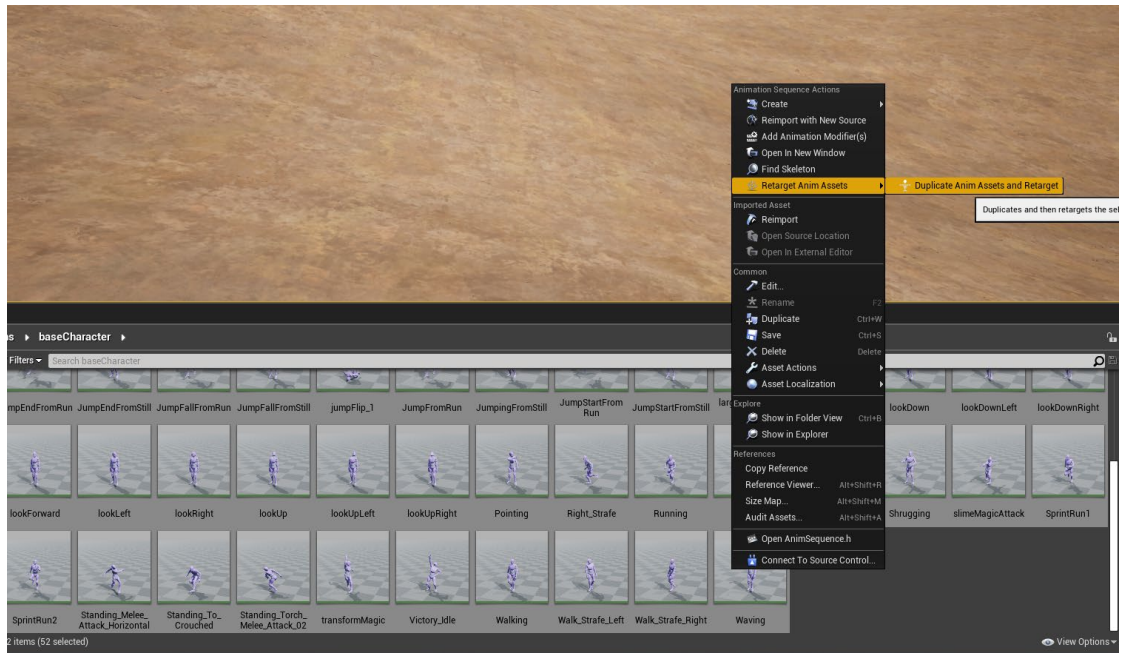


Figure 30. Retargeting selected animations.

After this a window for previewing the result opens up. In this preview window it is important for the both characters to be in the same pose. A T-pose is good for this. Figure 31 shows the review window with both characters in a T-pose.

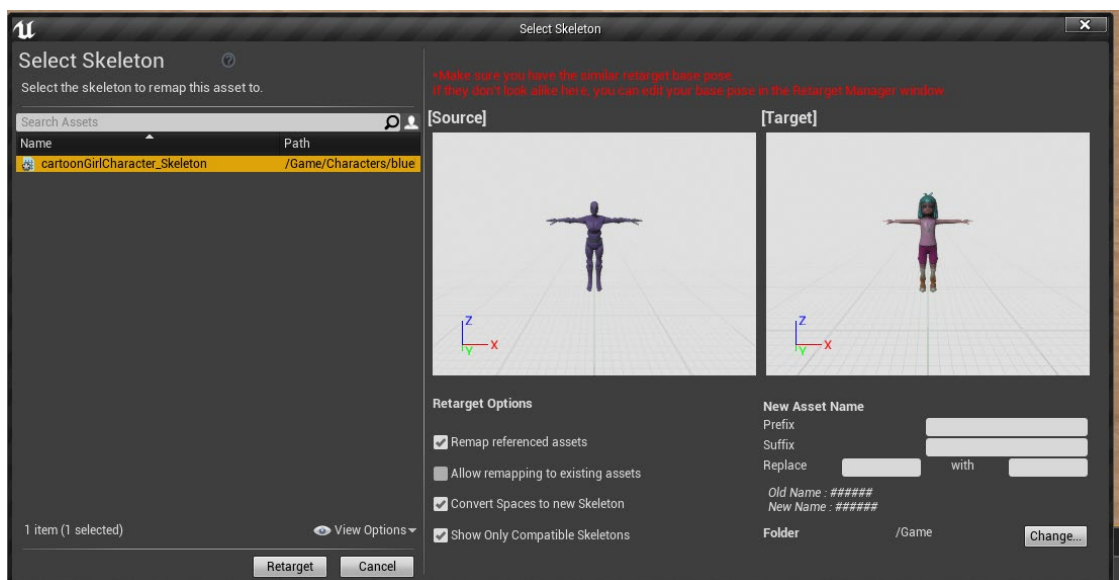


Figure 31. Duplicating animations skeleton selection.



However even if in the preview window everything seems to be properly set, it might not still go right on the first try. Figure 32 shows how the legs of the NPC are disfigured.



Figure 32. Retargeted skeleton on NPC with the NPC's legs disfigured.

If disfigurement happens it is probably because the bone names were not set correctly. This can be fixed by going back and re-setting the bone names in the **Retarget Manager**. Figure 32 shows the fixed NPC.



Figure 33. NPC's retargeting fixed.

For reusing the animations, it would have been best if the models were made for the same skeleton by using an external 3D program. Even if something seems to be the same on downloaded third-party made models, there may be some settings or changes that are not visible. As such, features or components needing them to be the exact same will not work or they may suffer in quality. It is possible to work with, but at worst will create extra work as happened with the NPC model during the implementation. Although this is also a question of knowledge before starting the actual work. Someone very experienced in 3D modelling might have the knowledge to fix the problems showcased here or may know what to do beforehand to avoid them completely.

## 6.7 Other implemented features

During the implementation there arose the need to add a few other features and change how some components worked. These had little to do with the animations themselves but were needed to help with the testing of **the Animation Blueprints**.

### 6.7.1 Camera

Originally the camera was fixed behind the character and the character's front could not be seen. However, during the development came a need to see how the character moves and works in all perspectives so that errors could be found more easily. For example, in the jump flip animation the model would move forward, but when only viewed from behind it was hard to realise why it seemed to lag weirdly at times. Unlocking the camera revealed the problem, the model moved forward during the animation, but it was not supposed to do so.

The following figures show the settings that were changed to modify the camera movement. Figure 34 shows the **Player Character Blueprint** overview where the settings for the **Player Character**, **CharacterMovement** and **Camera** can be found. In Figure 35 and Figure 36 it is shown how to set the **Player Pawn Settings** to enable the camera's movement settings, which are shown in Figure 37 and Figure 38. Lastly the settings for the **CharacterMovement** are shown in Figure 39 and Figure 40.

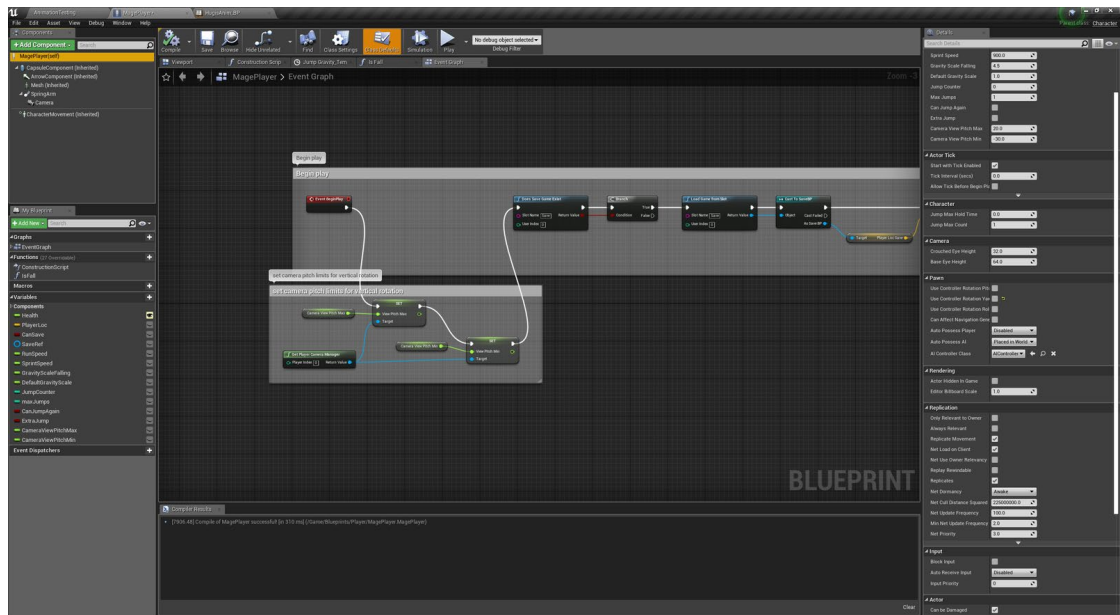


Figure 34. Player Character Blueprint Event Graph view.

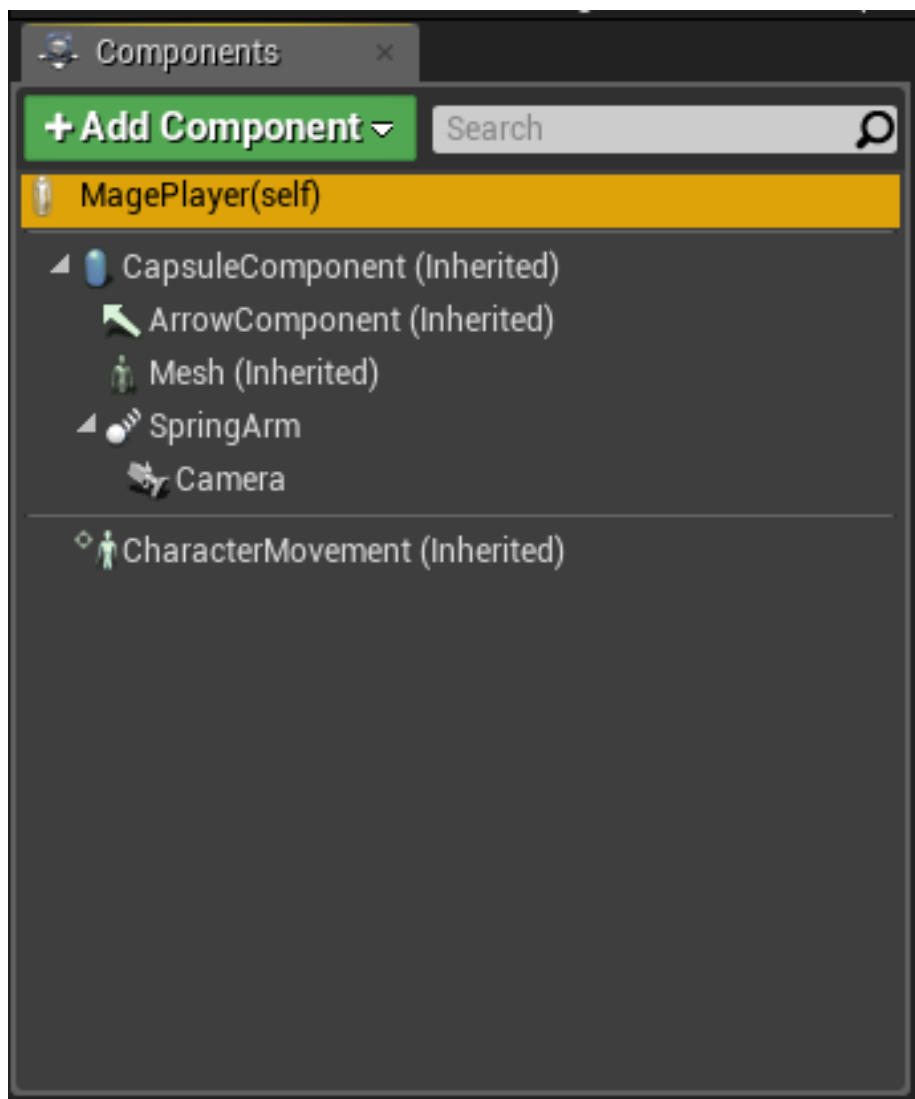


Figure 35. Components panel with the player selected.

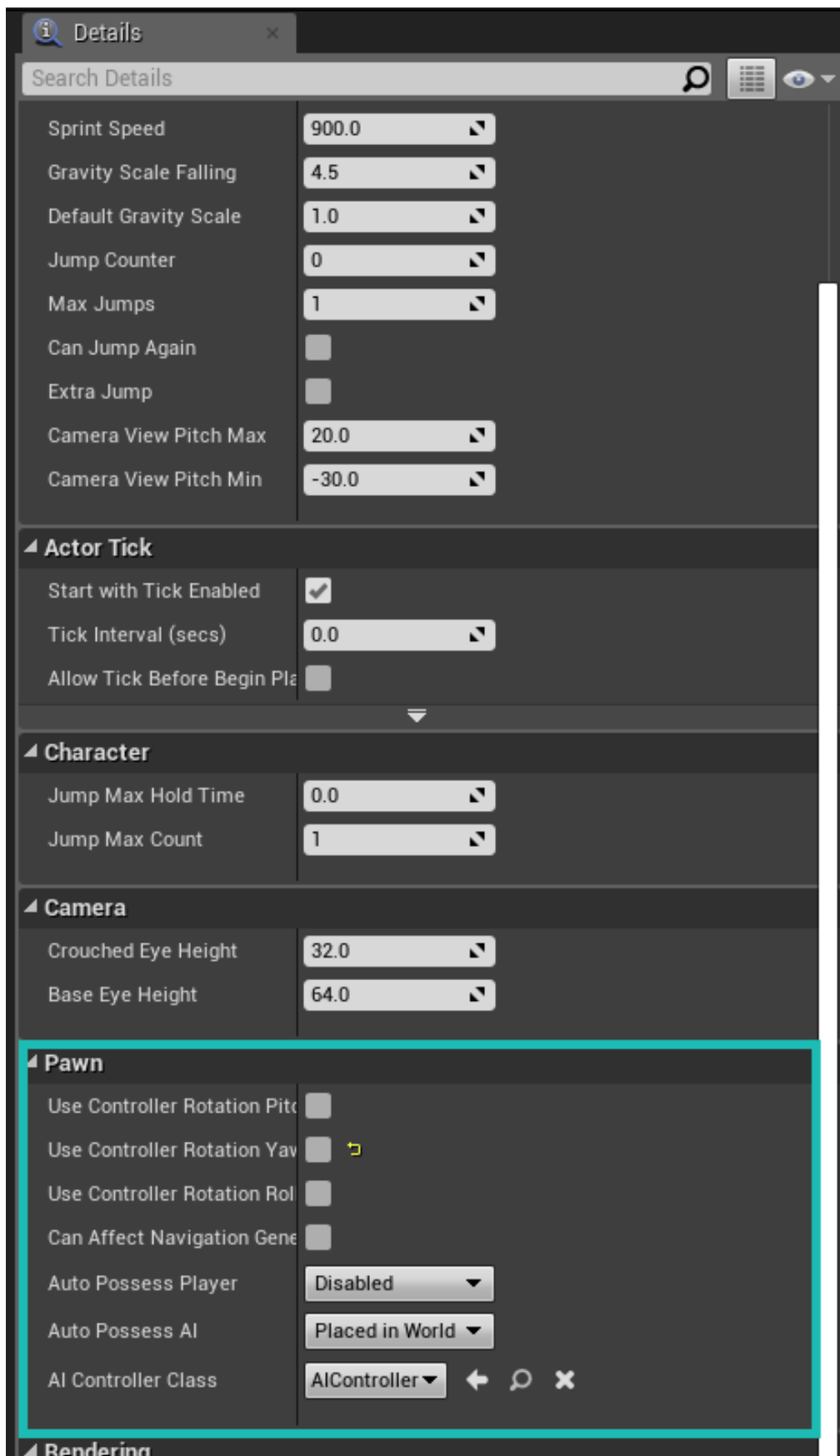


Figure 36. Player Details panel opened with the Pawn tab highlighted.

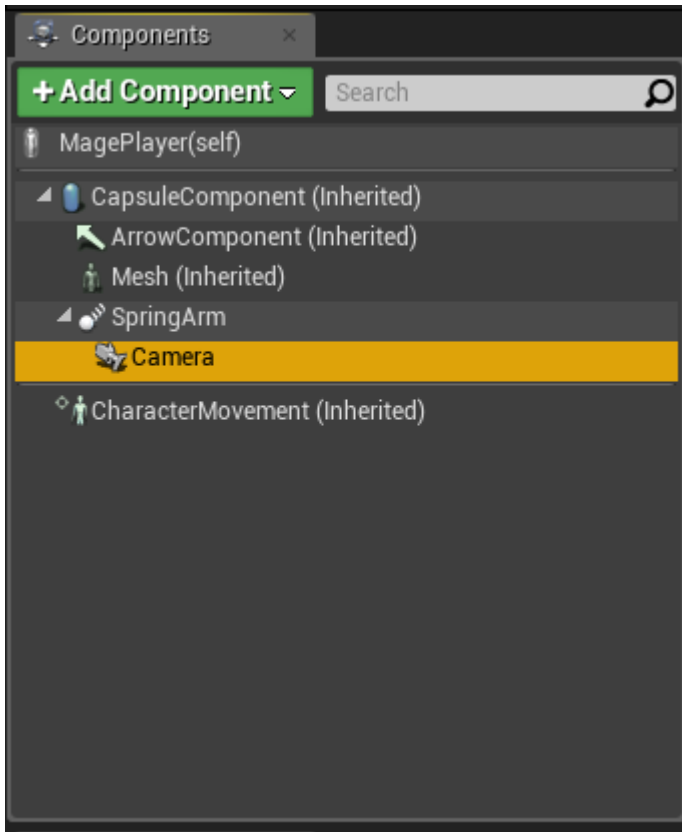


Figure 37. Components panel with Camera selected.

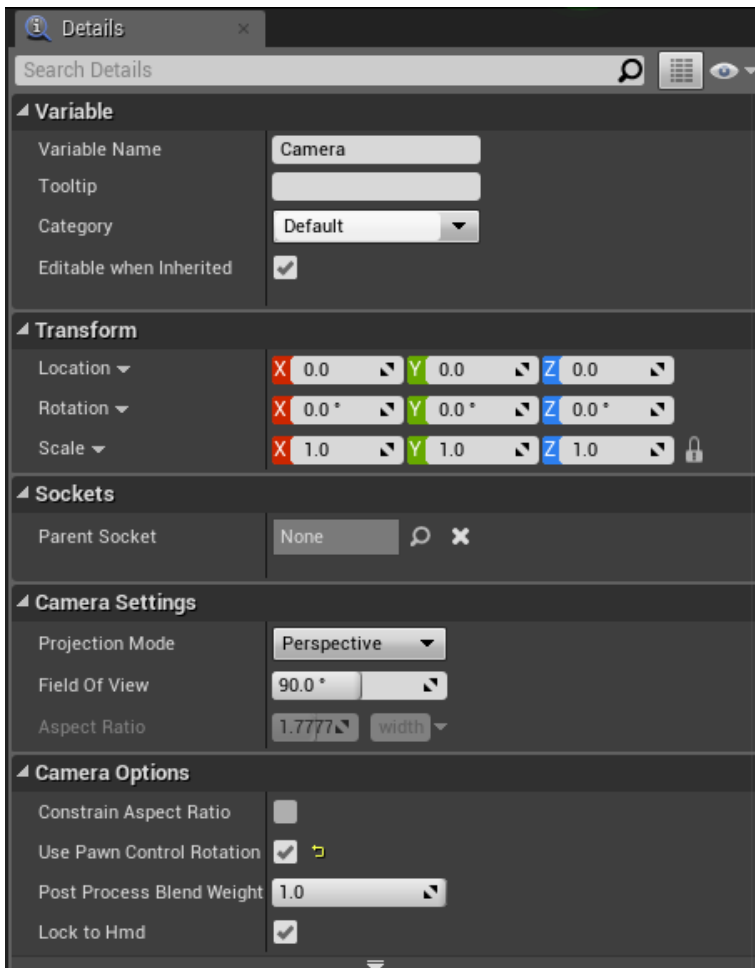


Figure 38. Camera's Details panel open.

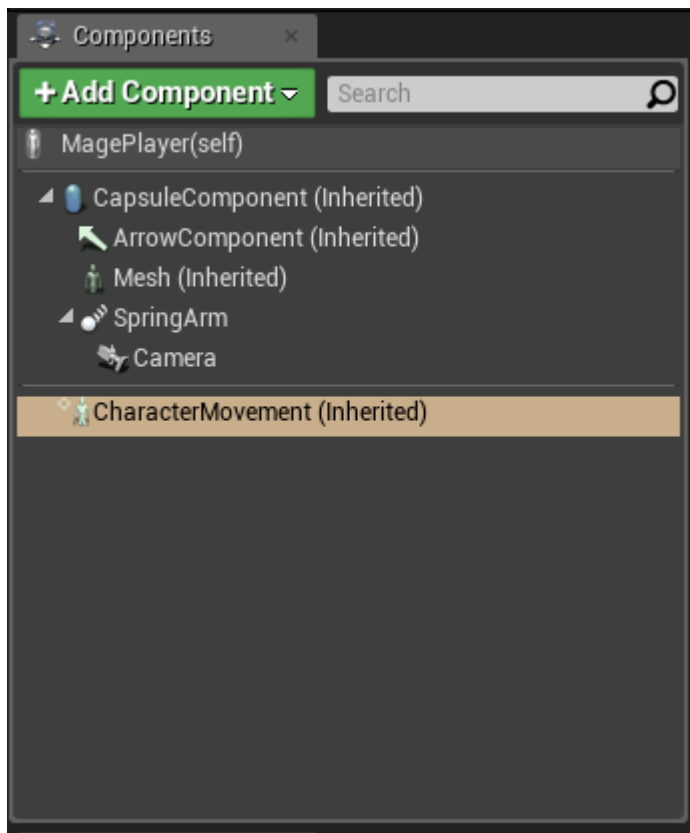


Figure 39. Components panel with CharacterMovement selected.

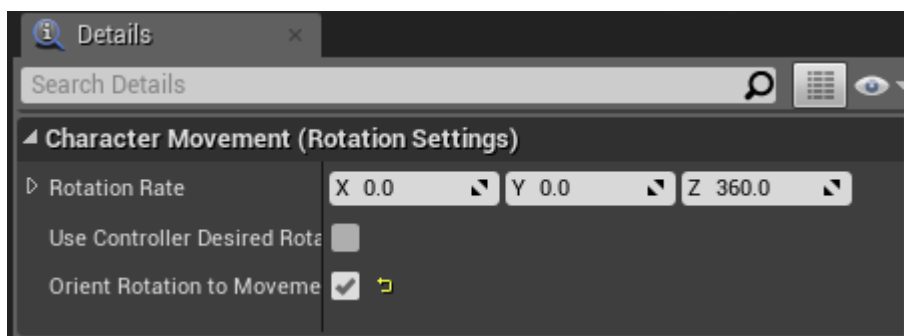


Figure 40. CharacterMovement details panel open.

## 6.7.2 Simple AI for NPC

To showcase and test the re-targeted animations an NPC was needed. A proper AI with human lifelike behaviour had not been implemented in the project by the time the re-targeted animations were made and needed to be tested. As such, there was a need for a simple AI. The simple implemented AI has one feature. It moves the NPCs in a designated area to new random

locations. The NPC needed two new **Blueprints**, one for the **AI Controller** and one for the **Character Pawn**. Figure 41 and Figure 42 show the creation of the **AI Controller** and Figure 43 shows the creation of the **Character Pawn**.



Figure 41. Making a new Blueprint Class.

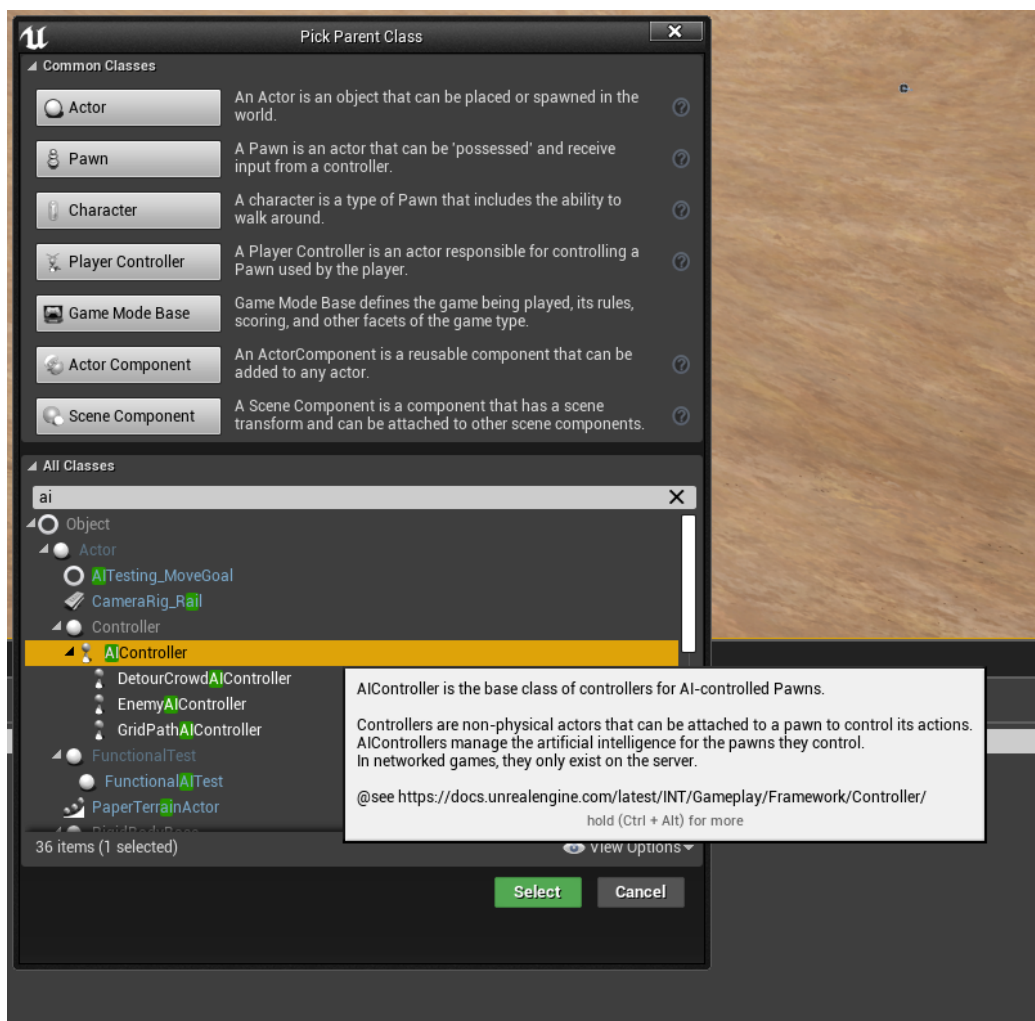




Figure 42. Making the new Blueprint into an AI Controller for the NPC.

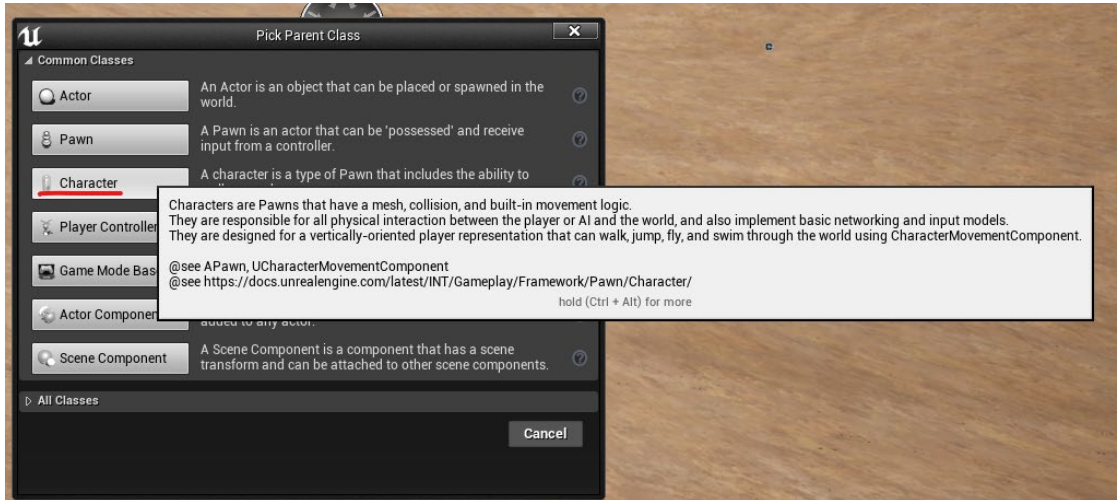


Figure 43. Making the Character Pawn for the NPC from the Blueprint Class.

In the following Figure 44 the **NPC Character Blueprint** is shown. There under the **Details** the **Skeletal Mesh** has been changed to the skeleton of the NPC. This is because as stated previously the animations in UE4 are tied to their skeletons and only models rigged with the exact same skeleton can use the animations for that skeleton. As shown previously the animations used for the base character were re-targeted for the cartoon girl character that is used for the NPC.

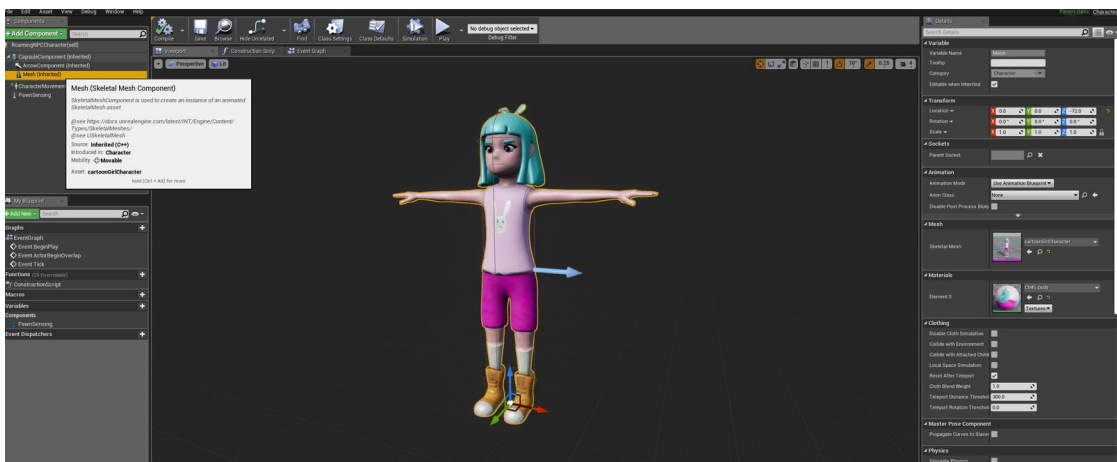


Figure 44. Changing the mesh for the NPC Character pawn.

In Figure 45 below the **Pawn Sensing** component has been added to the **NPC Character Pawn**, this enables the NPC to be aware of its surroundings. Features such as seeing or hearing the player are found in this component.

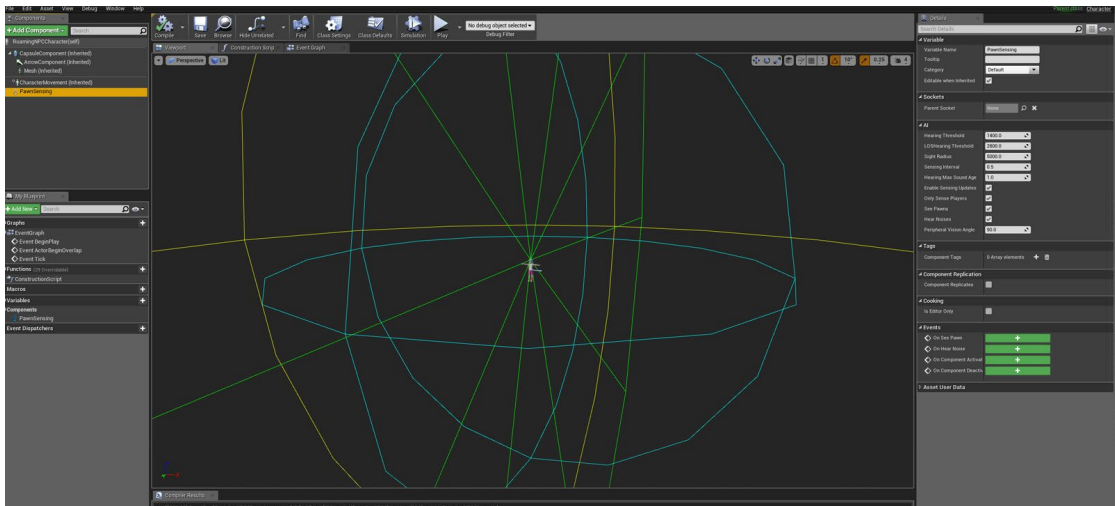


Figure 45. Pawn Sensing component on NPC Character Pawn.

In the following Figure 46 a custom event has been made for the NPC which enables it to move to a new random location. Figure 47 after that shows the custom event in a more detail. After the NPC has moved it waits for a random time between zero and fifteen seconds and then moves again.

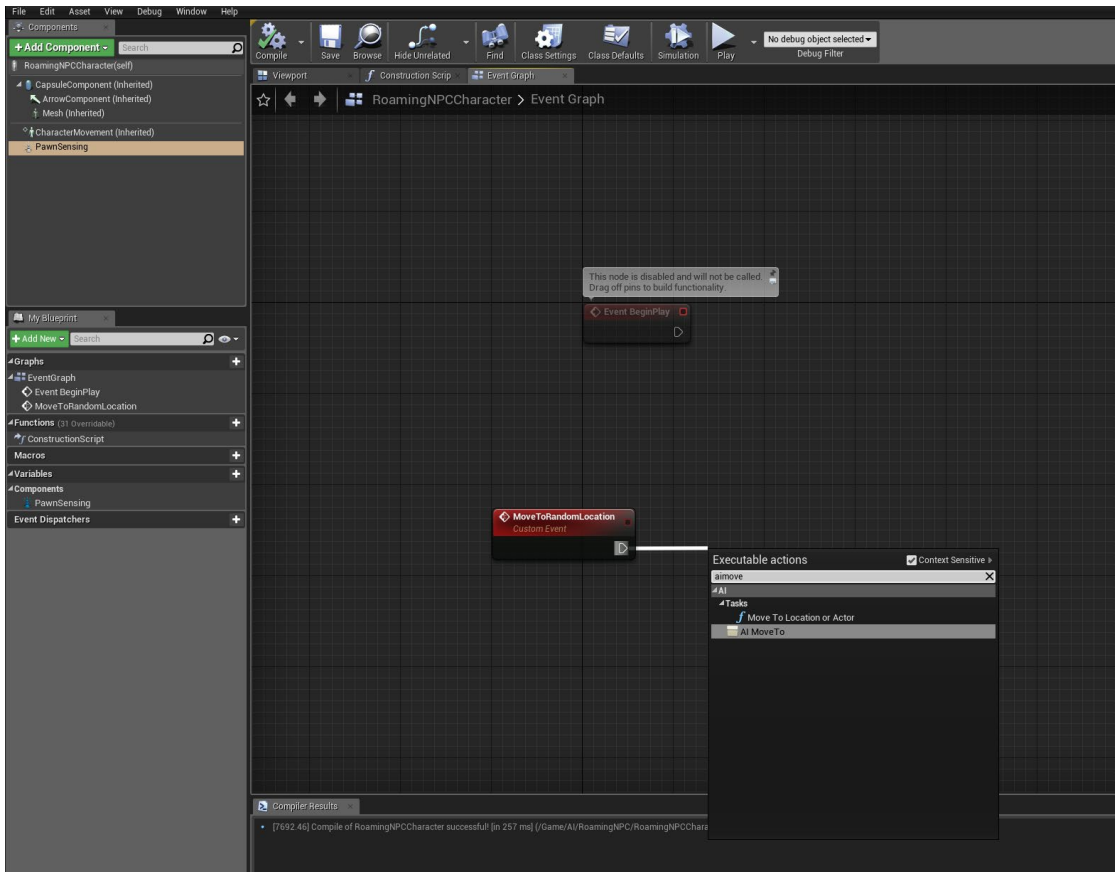


Figure 46. Custom Event for moving the NPC to a random location.

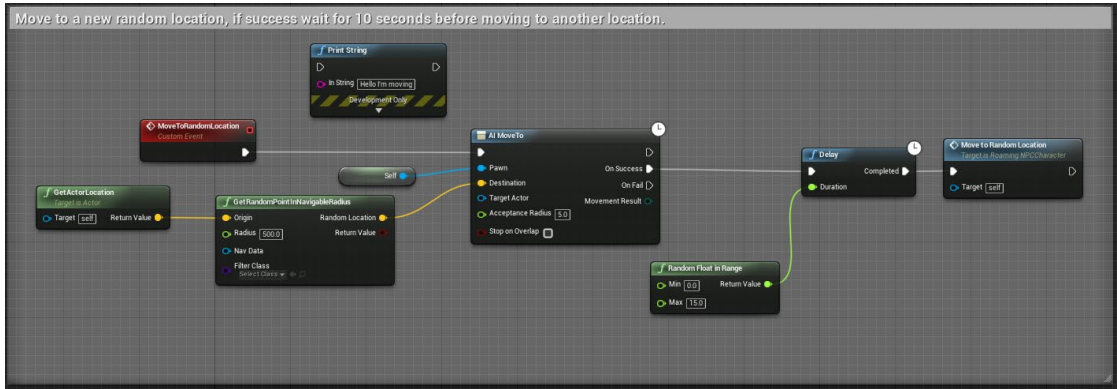


Figure 47. Custom Event MoveToRandomLocation as currently implemented in the project.

Figure 48 shows how the event is called to move the NPC.

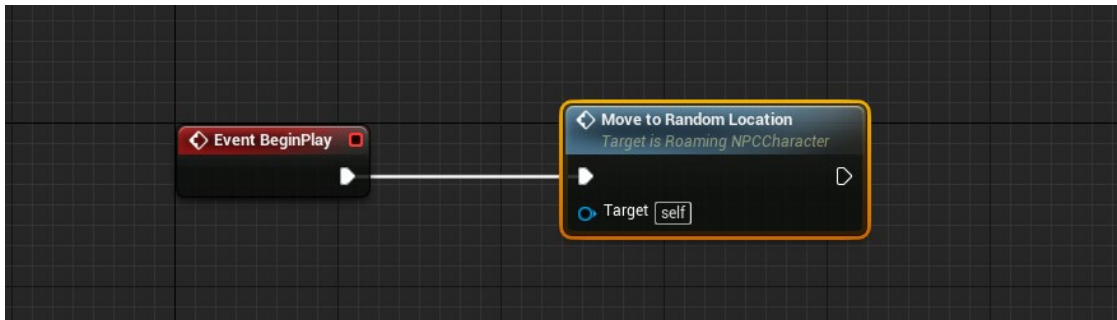


Figure 48. Calling the new event from the Begin Play.

Figure 49 and Figure 50, both below, show that the NPC has been added to the game map and that UE4's another ready component **Nav Mesh Bounds Volume** has been added as well. This component enables **AI Pawns** to move in the area where the component is.

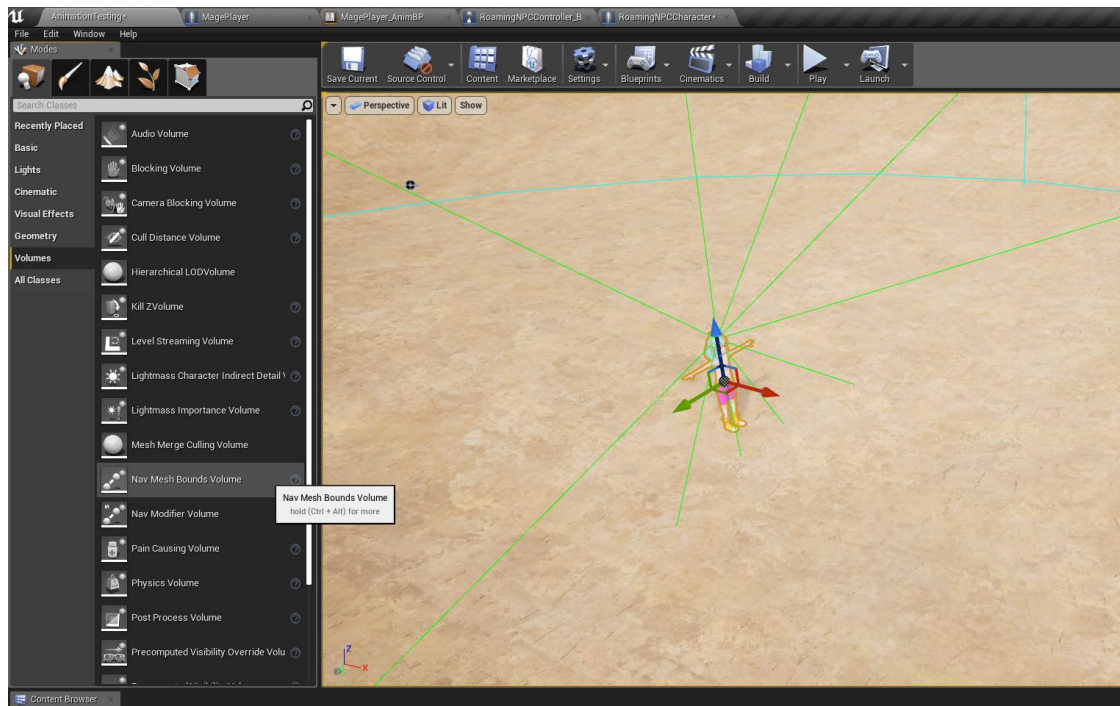


Figure 49. NPC put to the level and Nav Mesh in the side menu.

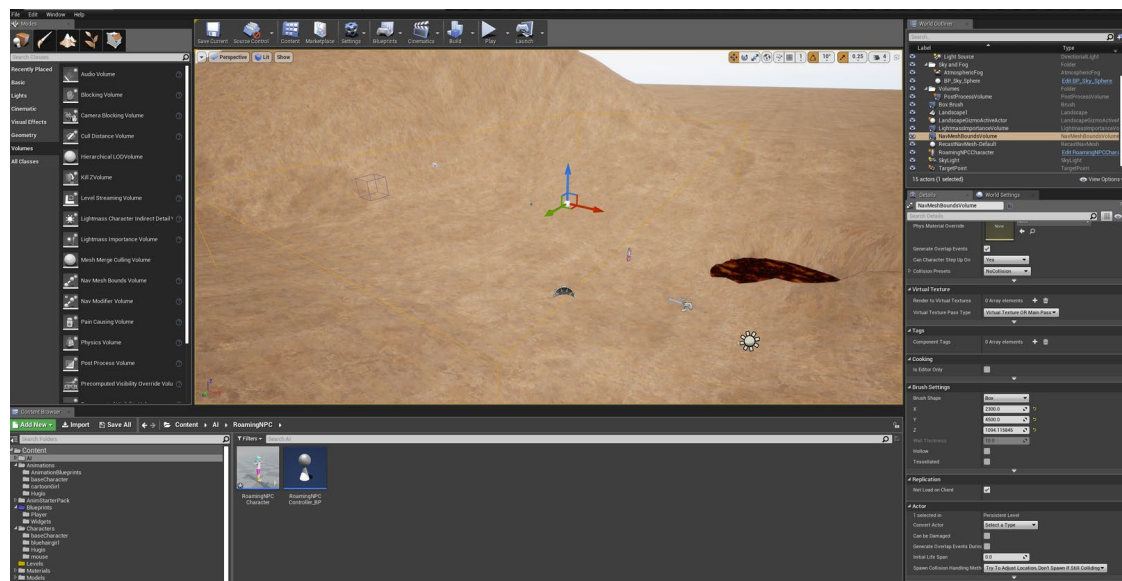


Figure 50. Nav Mesh Bounds Volume added to the level and scaled to cover a large area.

As the 3D model used for the NPC uses a different skeleton from the base character it needs its own **Animation Blueprint**. The NPC was given states for idle, walk and run. Similarly, to the player character, the NPC has three extra idle states that randomly happen if the NPC stays still for long enough. For the testing of the re-targeted animations and the simple AI these stages were enough. It is possible to expand it later. The animation states for the



NPC are shown in Figure 51 below. Controlling the animations via the **Animation Blueprint** for the NPC are shown in Figure 52, Figure 53 and Figure 54, also below.

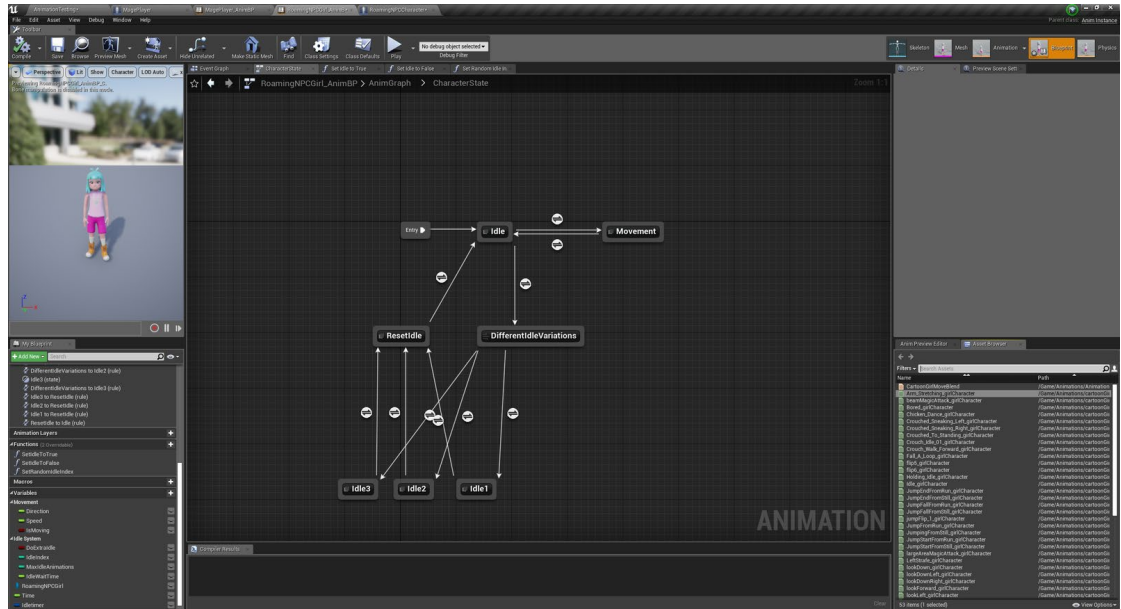


Figure 51. NPC animation states.

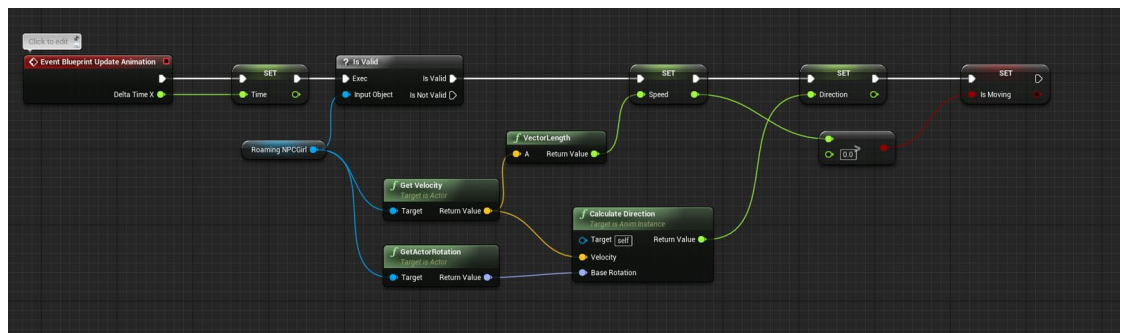


Figure 52. NPC animation update in the Animation Blueprint Event Graph.

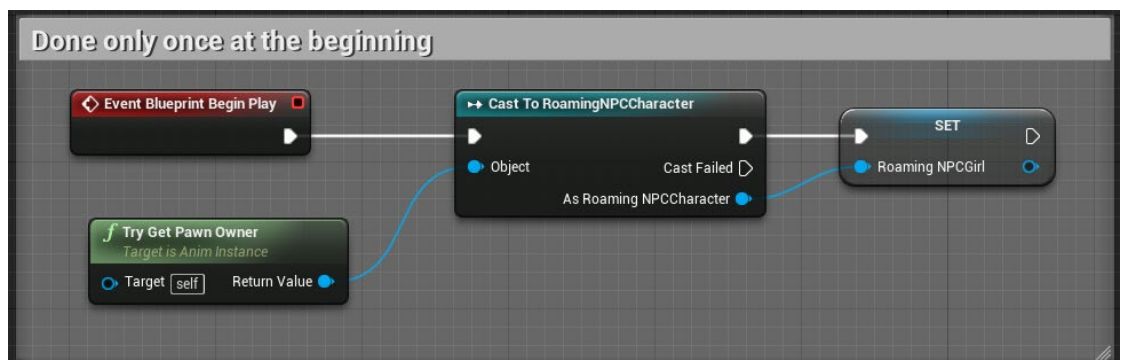


Figure 53. NPC Animation Begin Play in Animation Blueprint Event Graph.

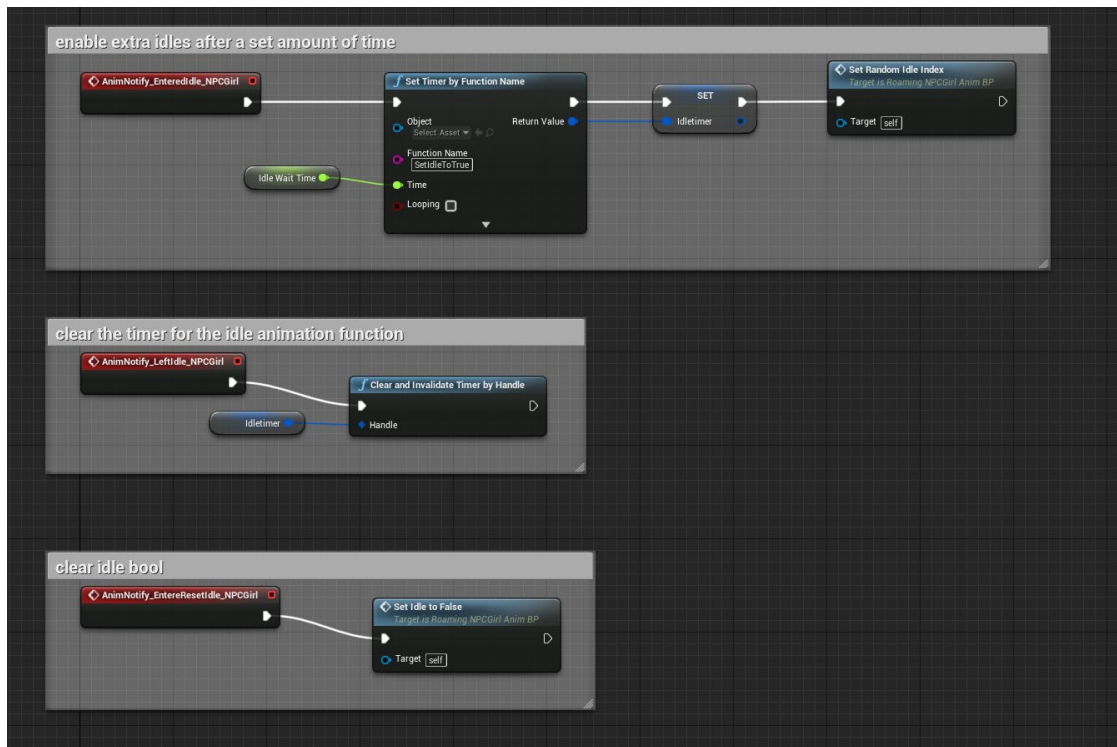


Figure 54. Idle system for the NPC using AnimNotifies in Animation Blueprint Event Graph.

## 6.8 Future development

Many features and parts of the project need to be improved in the future. They were left as they are because the timeframe meant for the thesis was too short to get everything optimal on the very first time working with the animation systems.

The most important part that must be done first is to make custom 3D models or choose a set of models that can be used throughout the game. The models need to be made from the same skeleton, otherwise the animation work will be strenuous. When the custom models have been made it will be easier to fix the jump flip problem. This is undoubtedly going to take a lot of time, and there is a possibility that it needs a custom implementation instead of using the **CharacterMovement**. After the problem with the models is solved it becomes possible to make the **Animation Blueprints** inheritable. Another important improvement is to make the **Animation Blueprints** easier to read. As they are now all of the states are in one view and it is hard to find the correct transition lines or states at times. Using **Sub Anim Instances** may help with this. It would be good to look into them more to find out what is the proper way to use

them. If it turns out they are not a suitable solution for the bloated **Animation Blueprint**, then other ways of tidying it up should be searched for.

## 7 CONCLUSIONS

The main objective for this thesis was to make inheritable Animation Blueprint for the player character. The blueprint was then supposed to be inherited by the NPC, and some of the key animation were to be overridden to give more personal feel for the NPC. Other objectives were to learn how to use UE4's animation tools and to understand the workflow with animations in UE4.

The implementation was the part of this thesis where most was learned. It must also be stated that even when following official documentation not everything will work out as shown in the examples. More so if the assets and scripts already in the project are not exactly setup the same way as what the examples would prefer them to be.

While working on this thesis many new procedures were learned and some old previously learned ones were reaffirmed. The most important aspect to have learned was the reusing of the animations and **Animation Blueprints**, and what all would have been needed for them to properly work. Also experimenting with different ways to get the animation states to work has given a better understanding of the kind of problems that may come up later.

This thesis may be good for others who wish to know more about animating with UE4, and what should be considered before starting the actual work. It might also help others not working directly with animation to understand it more, and to see what all should be considered at the start of a project. This thesis will also work as a documentation to help with the animation systems in Chicken Quest, so that the information is not lost over time. Hopefully, this will help with possible errors and problems that may arise in the future if parts of the project are changed.

The largest problems encountered while making this thesis were the jumping animation system and the transitioning after attacking and jumping back to being idle or moving states. The need for 3D models made from the same skeleton for reusing the **Animation Blueprints** also hindered the implementation. This however can be fixed in the future.

The first objective was not accomplished. It is not detrimental to the project as it helped gain important experience and information on the proper workflows. However, working on this thesis has revealed what all assets would have been needed before starting to work with the **Animation Blueprints**. This helps in the future when the 3D models are acquired and can be added into the game. They will be easier and faster to set up in the game. The other objectives were accomplished and by the end of the thesis it was noted that they were as important as the concrete goal of having made a reusable **Animation Blueprint**. This thesis has proven that when some goals are not met a lot more can be learned from the failure, than from accomplishing the objectives on the first try.



## References

2D Animation: Everything You Should Know About it. 2019. Renderforest. Blog. Available at: <https://www.renderforest.com/blog/2d-animation> [Accessed 9 March 2020].

Adobe. 2020. Mixamo WWW document. Available at: [https://www.adobe.com/devnet/author\\_bios/Mixamo.html](https://www.adobe.com/devnet/author_bios/Mixamo.html) [Accessed 2 February 2020].

Blender. No year. About. WWW document. Available at: <https://www.blender.org/about/> [Accessed 2 February 2020].

Chang, A. 2020. The Process of 3D Animation. WWW document. Available at: <https://www.media-freaks.com/the-process-of-3d-animation/> [Accessed 1 April 2020].

Collins, T. 2018. 3D Modelling Pipeline. Updated 22 June 2018. Article. Available at: <https://medium.com/@homicidalnacho/3d-modelling-pipeline-bd9be7dba136> [Accessed 1 April 2020].

Cooper, J. 2019. Game anim: video game animation explained. Boca Raton, Florida, USA: CRC Press.

How animation for games is different from animation for movies. Pluralsight. 2014. WWW document. Updated 4 March 2020. Available at: <https://www.pluralsight.com/blog/film-games/how-animation-for-games-is-different-from-animation-for-movies> [Accessed 25 March 2020].

Sito, T. 2013. Moving innovation : a history of computer animation. Cambridge: The MIT Press.

Unreal Engine. 2020a. Animation Blueprints. WWW document. Available at: <https://docs.unrealengine.com/en-US/Engine/Animation/AnimBlueprints/index.html> [Accessed 17 March 2020].

Unreal Engine. 2020b. Animation Retargeting. WWW document. Available at: <https://docs.unrealengine.com/en-US/Engine/Animation/AnimationRetargeting/index.html> [Accessed 24 March 2020].

Unreal Engine. 2020c. Animation System Overview. WWW document. Available at: <https://docs.unrealengine.com/en-US/Engine/Animation/Overview/index.html> [Accessed 13 March 2020].

Unreal Engine. 2020d. Blender to Unreal tools, Part 1 | Live from HQ | Inside Unreal. WWW document. Available at: [https://youtu.be/c3\\_xUMQ6hhs](https://youtu.be/c3_xUMQ6hhs) [Accessed 17 April 2020].

Unreal Engine. 2020e. Features. WWW document. Available at: <https://www.unrealengine.com/en-US/features> [Accessed 2 February 2020].

Unreal Engine. 2020f. Importing Static Meshes. WWW document. Available at: <https://docs.unrealengine.com/en-US/Engine/Content/Types/StaticMeshes/HowTo/Importing/index.html> [Accessed 17 March 2020].

What is 3D Animation Compared to 2D Animation? The Core Differences. 2019. Bloop Animation. WWW document. Updated 25 March 2019. Available at: <https://www.blopanimation.com/what-is-3d-animation/> [Accessed 10 March 2020].

White, T. 2006. *Animation from Pencils to Pixels: Classical Techniques for the Digital Animator*. Burlington: Focal Press, Elsevier.

## List of Figures

Figure 1. Idle system flowchart.....	25
Figure 2. Idle system animation states in the project.....	26
Figure 3. Custom events from Idle animation state set to enable using them. ....	28
Figure 4. Calling EnteredIdleBC as an AnimNotify to add logic from the Event Graph. ....	28
Figure 5. Reset idle AnimNotify for resetting the boolean for playing extra idles animations. ....	29
Figure 6. Movement Blend Space. ....	29
Figure 7. Movement Blend state in the Anim Graph.....	30
Figure 8. Transition between idle and movement state. ....	30
Figure 9. Jump script in the project. ....	31
Figure 10. Jump system in the project.....	32
Figure 11. Character model position during jump flip. ....	38
Figure 12. Character collider during the jump animation. ....	39
Figure 13. Halving the height of capsule component.....	40
Figure 14. Moving the character model to the collision position during the jump flip.....	41
Figure 15. Melee attack system.....	43
Figure 16. Spell attack system. ....	44
Figure 17. Input for crouching.....	45
Figure 18. Crouch Blend.....	46
Figure 19. CrouchBlend state in the Anim Graph.....	46
Figure 20. Spring arm settings to smooth camera movement during crouch.....	47
Figure 21. Settings for animation to be used in aim offset blend.....	48
Figure 22. Aim offset wrong behaviour.....	49
Figure 23. Aim offset correct behaviour.....	50

Figure 24. Weird behaviour of the NPC model when trying to use the same skeleton as the base character. ....	51
Figure 25. NPC with animations retargeted to the same skeleton.....	52
Figure 26. Base skeleton with Retarget Manager options. ....	53
Figure 27. Rig options in the Retarget Manager.....	54
Figure 28. Retargeting bone names in the Retarget Manager.....	54
Figure 29. Retarget options for the NPC. ....	55
Figure 30. Retargeting selected animations. ....	56
Figure 31. Duplicating animations skeleton selection.....	56
Figure 32. Retargeted skeleton on NPC with the NPC's legs disfigured. ....	57
Figure 33. NPC's retargeting fixed. ....	58
Figure 34. Player Character Blueprint Event Graph view. ....	60
Figure 35. Components panel with the player selected.....	60
Figure 36. Player Details panel opened with the Pawn tab highlighted. ....	61
Figure 37. Components panel with Camera selected.....	62
Figure 38. Camera's Details panel open. ....	63
Figure 39. Components panel with CharacterMovement selected. ....	63
Figure 40. CharacterMovement details panel open.....	63
Figure 41. Making a new Blueprint Class. ....	64
Figure 42. Making the new Blueprint into an AI Controller for the NPC.....	65
Figure 43. Making the Character Pawn for the NPC from the Blueprint Class. ....	65
Figure 44. Changing the mesh for the NPC Character pawn. ....	65
Figure 45. Pawn Sensing component on NPC Character Pawn.....	66
Figure 46. Custom Event for moving the NPC to a random location. ....	67
Figure 47. Custom Event MoveToRandomLocation as currently implemented in the project.....	67
Figure 48. Calling the new event from the Begin Play.....	67

Figure 49. NPC put to the level and Nav Mesh in the side menu. ....	68
Figure 50. Nav Mesh Bounds Volume added to the level and scaled to cover a large area. ....	68
Figure 51. NPC animation states. ....	69
Figure 52. NPC animation update in the Animation Blueprint Event Graph. ....	69
Figure 53. NPC Animation Begin Play in Animation Blueprint Event Graph. ....	69
Figure 54. Idle system for the NPC using AnimNotifies in Animation Blueprint Event Graph. ....	70

## List of Tables

Table 1. Transition rules for idle system.....	26
Table 2. Transition rules for jumping from idle.....	34
Table 3. Transition rules for jumping from movement. ....	35