

ANIMATRONICS CONTROL USING THE UNREAL ENGINE'S ANIMATION FEATURES

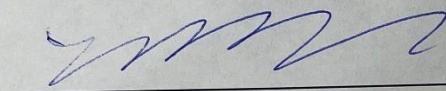
ANIMATRONIC THESIS  
Cat Alvord - May 2024

Presented to the Design and Production Faculty of the University of North Carolina School  
of the Arts in partial fulfillment of the requirements for the Degree of Master of Fine Arts

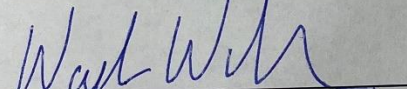
Accepted by:

  
Thesis Committee Chairman – Eric Hart

5-2-24  
Date

  
Committee Member – Yu-Ang Teng

02/05/2024  
Date

  
Committee Member – Wade Wilson

5/2/24  
Date





# Table of Contents

<b>Chapter 1: Thesis Intent.....</b>	<b>2</b>
Unreal Engine.....	3
<i>What is a game engine? .....</i>	<i>3</i>
<i>What's so special about Unreal?.....</i>	<i>3</i>
<i>Who else is benefitting from Unreal outside of game developers? .....</i>	<i>4</i>
Animatronics .....	6
<i>What are Animatronics?.....</i>	<i>6</i>
<i>How do you make them move? .....</i>	<i>6</i>
<i>What's so important about the history of controls?.....</i>	<i>7</i>
Combining Unreal and Animatronics .....	8
<b>Chapter 2: Interactive Demonstration .....</b>	<b>9</b>
How are these animation tools being demonstrated? .....	10
Meet Bean Bag the Penguin.....	11
The Animatronic Figure .....	11
The Pebble Game .....	12
<b>Chapter 3: How it Works .....</b>	<b>13</b>
Linking the Digital and Physical World .....	14
<i>Joystick to Unreal.....</i>	<i>14</i>
<i>Maya to Unreal.....</i>	<i>14</i>
<i>Animation onto Player Character.....</i>	<i>15</i>
<i>Digital Figure to Physical Figure.....</i>	<i>17</i>
Game Layout and Flow of Data .....	22
Using Unreal's Animation Features.....	23

<i>State Machines</i> .....	24
<i>What is this feature?</i> .....	24
<i>How is this feature normally used in game development?</i> .....	25
<i>What problem does this solve for Animatronics?</i> .....	26
<i>How was this used in my thesis demonstration?</i> .....	26
<i>Animation Blending</i> .....	27
<i>What is this feature?</i> .....	27
<i>How is this feature normally used in game development?</i> .....	27
<i>What problem does this feature solve for Animatronics?</i> .....	27
<i>How was this used in my thesis demonstration?</i> .....	28
<i>Aim Offsets</i> .....	29
<i>What is this feature?</i> .....	29
<i>How is this feature normally used in game development?</i> .....	30
<i>What problem does this feature solve for Animatronics?</i> .....	30
<i>How was this used in my thesis demonstration?</i> .....	31
<b>Chapter 4: Conclusion</b> .....	<b>32</b>
Glossary of Terms.....	<b>34</b>
Works Cited.....	<b>35</b>



## Chapter 1: Thesis Intent

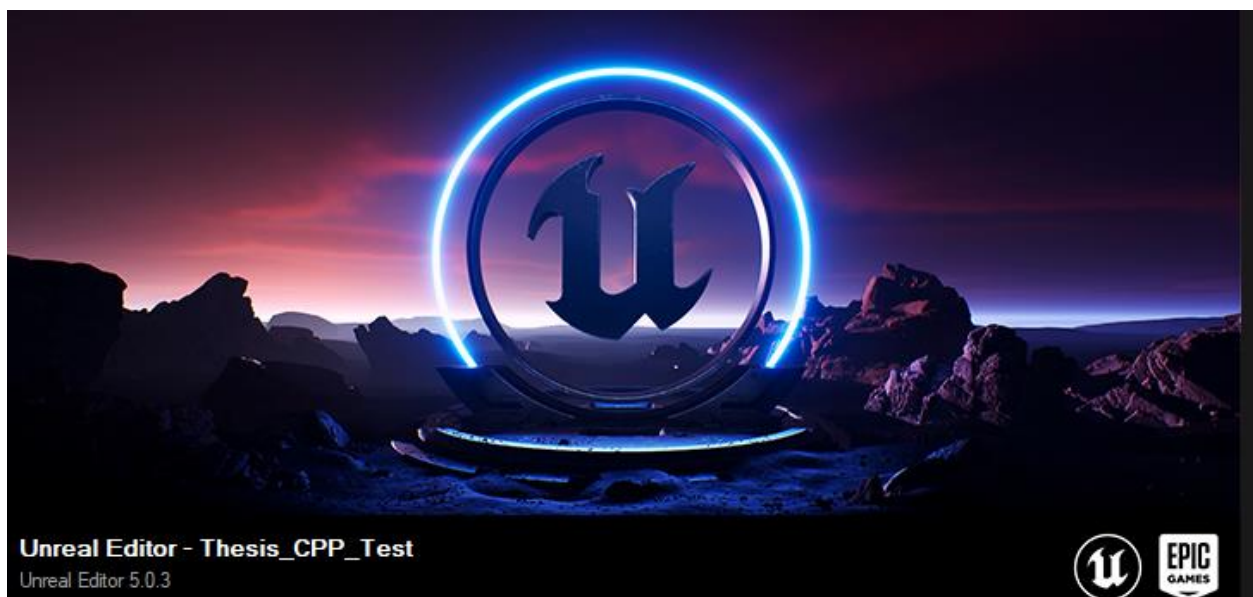
## Unreal Engine

### What is a game engine?

A *game engine* is a software framework that assists developers in creating video games. One way I like to describe it is “what Adobe Photoshop is to digital artists.” Photoshop provides the canvas and all the tools needed for a digital artist to paint a picture. It handles all the coding and computation under the hood so that the artist doesn’t need to worry about *how* the brushes work (or even how to code one from scratch every time). Instead, the artist has the freedom to focus on creating art.

Game Engines essentially provide the same for game developers. They give the developer the environment and tools needed to create a video game. However, unlike Photoshop, game engines also give developers more flexibility by allowing them to go in and edit or add code to make the game exactly what they want.

For many years now, game engines have been an integral part of video game development. Although most game engines are proprietary and only available for use by the game studio that created them, some game engines are made available for anyone. One such engine is the Unreal Engine created and developed by Epic Games.



*A splash screen from the Unreal Engine*

### What's so special about Unreal?

The Unreal Engine - which I will refer to as “Unreal” from here on out - has experienced a dramatic uptick in popularity over the last couple years. Unreal has many

desirable features that aid in its popularity, such as: easy to learn software with plenty of tutorials and documentation on the web, extremely advanced lighting and rendering capabilities, and an affordable price. For most users looking to learn Unreal, it is completely free, making it accessible to hobbyists, students, and professionals.

All of these features, combined with hitting the market at just the right time, have given Unreal another edge over its competitors: Unreal has extended beyond the reach of the gaming industry.

### Who else is benefitting from Unreal outside of game developers?

Although Unreal was created to solve challenges faced with creating a video game, many other industries often find themselves facing the same challenges. Whereas these industries typically have their own specialized software, Unreal has become an accessible, one-stop-shop software that brings all of these features into one easy to use program. Filmmakers can use Unreal to create VFX, animations, or even complete animated films.



*A still from a short film created in Unreal 4 by Weta Digital*

*Unreal is used to create the background on LED panels surrounding the set of the Mandalorian*

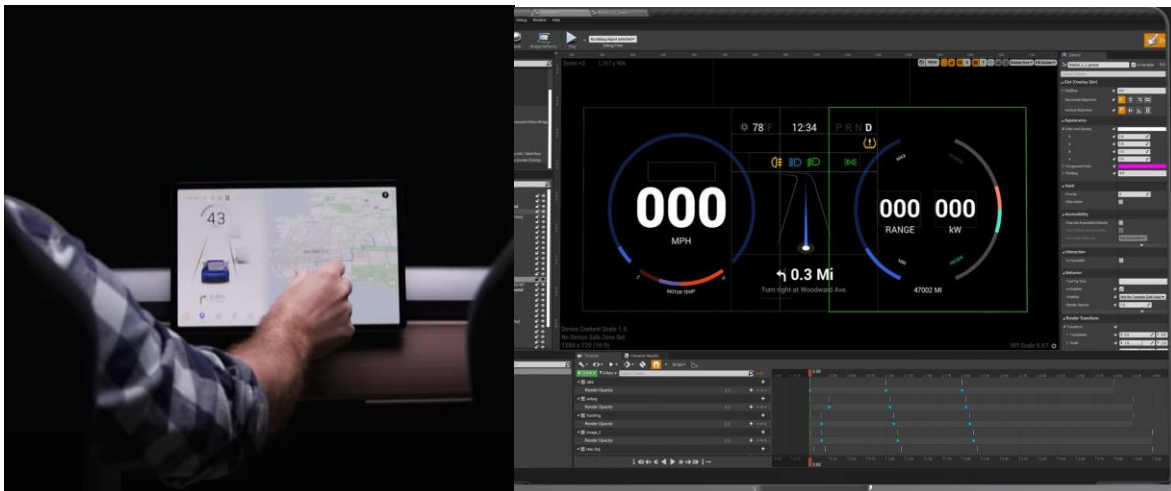
Architects can use Unreal to create astonishingly realistic renderings of spaces as they tweak and edit the scene on the fly.





*An architectural rendering showcasing Unreal's Lumen (lighting) technology*

Even some in the automotive industry have started using Unreal to create a vehicle's interactive center-console display.



*Left and Right: The new Hummer EV by GMC will use Unreal for its center console display*

These are just some of many ways that Unreal is being used creatively to solve industry-specific problems in the real world. I have no doubt that the engine will continue to grow and develop to benefit a wider array of needs as the demand for it rises. What interests me most, however, is the current impact that Unreal can have on the future of animatronics.

## Animatronics

### What are Animatronics?

“Animatronic” or “Animated Figure” is a generic term often used to describe a puppet controlled by electromechanical means. In 1961, Walt Disney coined and later trademarked



*Walt Disney admiring one of the famous Enchanted Tiki Room animatronic figures*

the term “Audio-Animatronic” to refer to the robotic characters being created at WED Enterprises for Walt Disney’s Enchanted Tiki Room attraction. Although these were not the first animatronic figures, they were the first to have their movements programmatically synced with sound. This was also the first of many revolutionary innovations that the Disney company has pioneered in the field of animatronics.

### How do you make them move?

Syncing audio with motion may not seem that impressive today, but it was no easy feat for engineers at the time.

Animators had to painstakingly hand-carve notches into discs for a cam

and follower mechanism to trigger the birds’ motions. Later the system was updated so that motion triggers were marked onto magnetic tape instead. Although these methods of animation worked for their intended purpose, they were tedious and provided little flexibility for changes.

The 60s were very prosperous for animatronics as Disney Imagineers tried various control systems to find one to fit all their needs. Not long after the Tiki Room opened, WED



Enterprises debuted another unique control system in “Great Moments with Mr. Lincoln” at the World’s Fair. This animation method involved rigging animator Wathel Rogers into a primitive motion capture harness to record his movements onto the figure. The end result was an incredible 5 minutes and 15 seconds of Abraham Lincoln speaking before an audience with the motion and subtleties of a real human being.

In 1969, advancements in computer technology allowed for the creation of the Digital Animation Control - or DAC - system. Now, animators could use knobs and buttons to record, edit, and playback animation onto the figure in real-time. This was an efficient and effective way of perfecting animatronic animation and formed the basis of what Disney has used until today.



What’s so important about the history of controls?

*Animator Wathel Rogers demos the live motion-capture system used on Walt Disney's Carousel of Progress*

Over the past 50 years, many animatronics companies have made a name for themselves in the industry, but the Disney company has always been the gold-standard. One of the ways that Disney historically set itself apart was based on a principle shared by all of its control systems: good animation is the driving force behind every figure’s movement. Nothing is random and nothing is out of time. Every movement or pause is intentionally crafted by someone who understands the nuances of animation.

In the past decade, physical knobs and buttons have mostly given way to screens and graphs but that defining principle has remained the same. It has become the industry standard to use Maya to create animations on a digital figure which will then be replayed on a physical figure. Each company may have its own unique system for exporting animation curves onto a figure’s motors, but they generally do everything they can to uphold the integrity of the original animation.

## Combining Unreal and Animatronics

Although this heavy reliance on man-made animation results in more lifelike figures, using pre-recorded Maya animations alone creates a figure that must follow a pretty strict routine. In order for a figure to cleanly loop through a show in front of an audience, either the final frame of animation must match the beginning frame of the next — or a computer must figure out how to smoothly move the figure between its final and initial poses. For figures that need to be interrupted at irregular intervals (like a boat ride where vehicles may not always move at the same speed) the animatronic must be able to smoothly transition to the main show animation from any point during the idle animation. Interactive animatronics also presents a unique challenge.

Many companies have developed in-house solutions to these issues, but the animatronics industry is not alone in facing these challenges. For years, the video game industry has also faced the same problems when creating programming digital characters. Nobody wants to see abrupt or janky animations in games, so developers have spent years perfecting tools to fix this.

Of course, with Unreal being one of the most popular game engines on the market today, it is filled with various tools to aid developers in programming complex digital characters. With this thesis, I intend to explore the many ways Unreal can be used to create a life-like animatronic character through its advanced animation capabilities.



## Chapter 2: Interactive Demonstration

## How are these animation tools being demonstrated?

In order to explore Unreal's animation capabilities on a physical figure, I decided to create an interactive demonstration that would highlight not only the actual functionality of the animation tools - but create a compelling experience to show just how much real-time animation can improve the audience's emotional connection with the figure.

To accomplish this, I decided to create a video game where the playable character was a physical animatronic. Players would be prompted to aid the character in their task and would control the figure much like a regular video game character. This means that instead of actually puppeteering the individual movements of the character, the player merely guides the character's actions in the same way that one would control any regular video game character. To create the game around the animatronic, the environment is projected on screens around the character in real-time. This helps create the illusion of a video game character that has come out of the screen and exists in our world.

To create the final experience, I needed a computer capable of running Unreal at a high enough framerate, some projectors, and screens, and of course, an animatronic figure. However, in order to tell if my program was working and if the data coming out of Unreal was even usable, I couldn't use just any moving figure. I needed an animatronic that could move in such a way that would make sense for the game and, most importantly, had enough resolution in its motors to move smoothly. I also needed the figure to have a one-to-one digital version that was rigged exactly how it moved in real life. Since I did not already have one available, I decided to create an animatronic character specifically for the purpose of my thesis and this game.

## Meet Bean Bag the Penguin



Bean Bag is a Gentoo penguin living a happy and peaceful life with his girlfriend in Antarctica. He's a little clumsy and he's certainly not in the best athletic shape, but he makes up for it with his big heart and loveable attitude. As is customary with Gentoo penguins, he has decided to propose to his girlfriend with the nicest pebbles he can find. You -- the player of this game -- have been tasked with helping Bean Bag find and collect the best pebbles to bring back to his girlfriend.



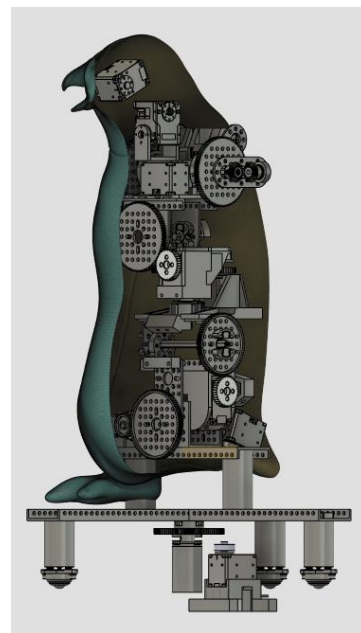
*A Gentoo penguin at the Edinburgh Zoo bringing a pebble to his mate.*

### The Animatronic Figure

Bean Bag in real life is a 12-function animatronic figure that uses Dynamixel motors and GoBilda aluminum parts. He is about 25 inches tall and designed based on real-life Gentoo penguins. I used gears

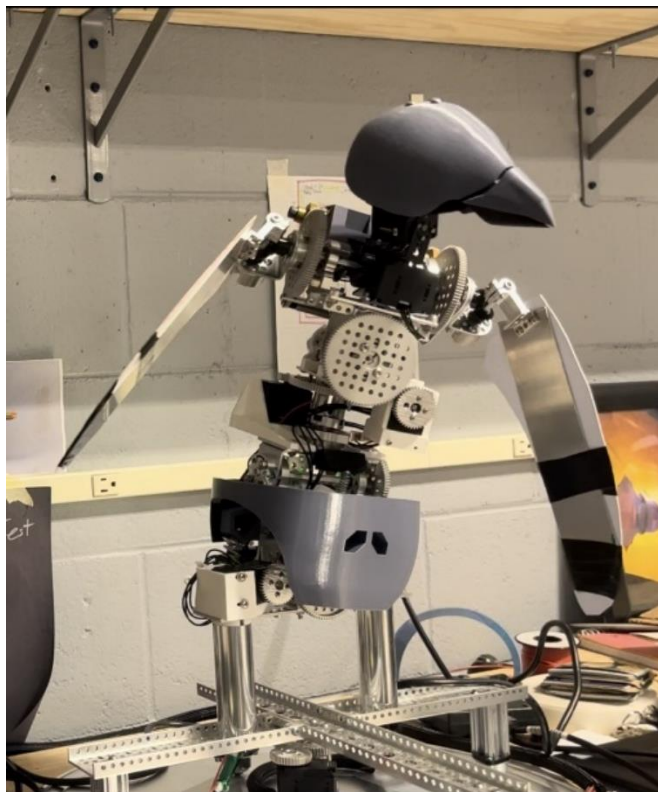
to increase the resolution and the torque of his motors so that all of his motions are as smooth as I can make them.

To create the final figure, I started by sculpting him in Zbrush and then used Fusion to create his mechanical design and Maya for rigging and animating him. By using this digital pipeline, I was able to have a one-to-one digital copy of my physical figure. This is crucial for programming a figure via animation data. His 12 functions are: full figure spin, body tilt, tail waggle, body foresway, chest turn, chest tilt, wing rotate, wing flap, head turn, head tilt, head nod, and jaw open/close.



*Bean Bag's mech design in Fusion*





*Bean Bag as an animatronic figure.*

For the figure finishing, I created a pattern based on a 3D printed maquette and sewed together a stretchy fleece skin. Under that skin is a second skin containing stuffing and weighted beads that help create his plushy look. The use of the beads and the overall effect of these “fat sacks” is where he gets his name.

## The Pebble Game

Players are tasked with aiding Bean Bag in his quest to find the perfect pebble for his lover. He is controlled via a Logitech Extreme 3D Pro joystick and can perform various actions such as idling, walking, braying, and interacting with pickups.

The player uses the joystick to maneuver Bean Bag around the level to find each of the glowing pebbles to bring back to the nest. Upon finding these pebbles, the player can press the trigger button to pick them up. This causes Bean Bag to play his “pickup” animation and the pebble disappears from the screen. The player then maneuvers Bean Bag back to the nest and deposits the pebble he has just picked up. The pebble appears in the nest and the game is completed once all of the pebbles have been found. Standing beside the nest in the virtual world is Bean Bag’s girlfriend. She interacts with the physical Bean Bag figure, helping tie together the digital and physical world.



*Bean Bag's girlfriend, nest, and game environment as seen through the Unreal Editor.*



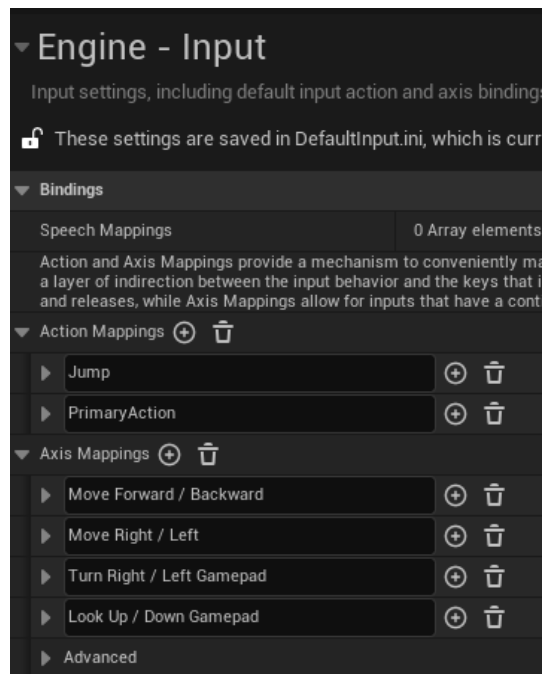
## Chapter 3: How it Works

## Linking the Digital and Physical World

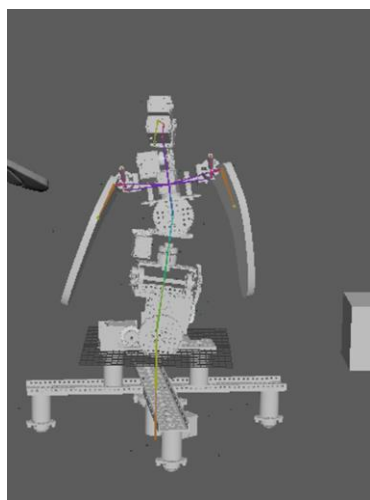
### Joystick to Unreal

Unreal's Third Person template already has complete controller functionality attached to a humanoid character. These controller inputs have been mapped to let the player character run around, jump and interact with items. Since my flight stick controller differs in function and layout from a generic, two thumb stick controller, I had to alter the settings in order to map my joystick to the functions I wanted.

These settings can be found in the Project Settings under 'Engine - Input' and are split into two categories: *Action Mappings* and *Axis Mappings*. Action Mapping refers to button presses, and Axis Mapping refers to the movement of the joystick. Here, I assigned the trigger button as "Interact" and then assigned the appropriate axes for moving my character around.



*The default input settings for the Third Person template*



*My mech design rigged in Maya*

### Maya to Unreal

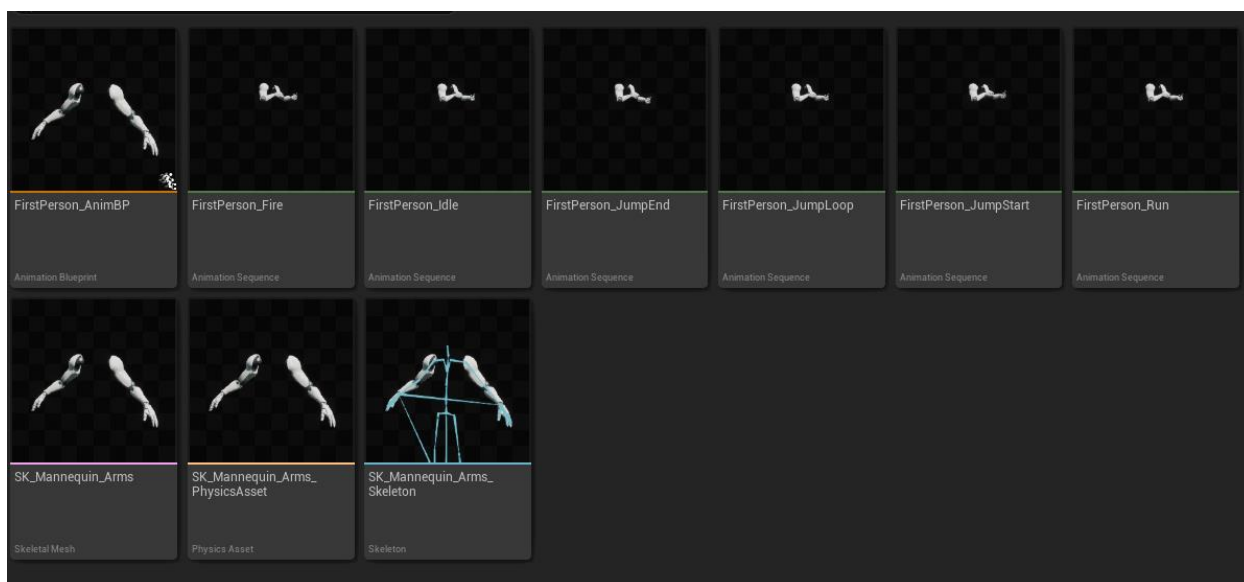
Although Unreal provides robust tools for applying animation assets onto a character, its tools for creating animation from scratch are rudimentary. In order to uphold the intention of placing clean and professional animation onto a figure, I used Maya to rig and animate my character.

I uploaded my mechanical design from Fusion into Maya and rigged it so that the joints would precisely match the movements of the motors. Each motor on my physical figure corresponds with a joint in Maya that has limits set to match the motor's min and max position. Then, the animation data (angle of each joint at each frame) can be mapped onto the servos in order to control the figure. This is the industry standard for controlling animatronics for themed entertainment.

A programmer will then usually use a script to export the animation curves as data that can be read by another program for playback on to motors. Since I want to use the actual animations and not just the curve data, I will instead export the animation clips and rigged character as FBX files. These files can be imported into Unreal and the animation assets can then be associated with and used on my player character.

### Animation onto Player Character

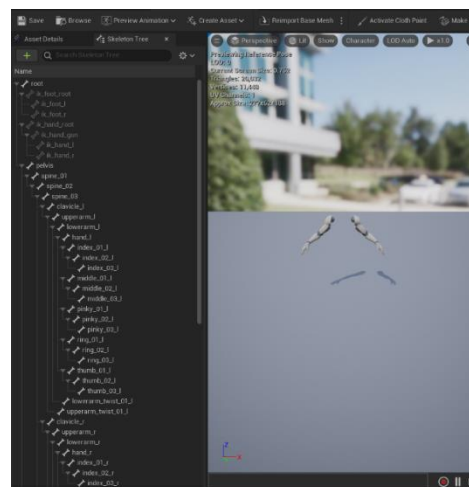
When importing one of these FBX files into Unreal, it creates a *Skeletal Mesh*, *Skeleton*, *Physics Asset*, and - if there is animation present - an *Animation Sequence*. To keep things simple, I am only focused on using the Skeletal Mesh and the Animation Sequences.



From Left to Right: The Animation Blueprint, Animation Sequences, Skeletal Mesh, Physics Asset, and Skeleton created for the default Third Person character

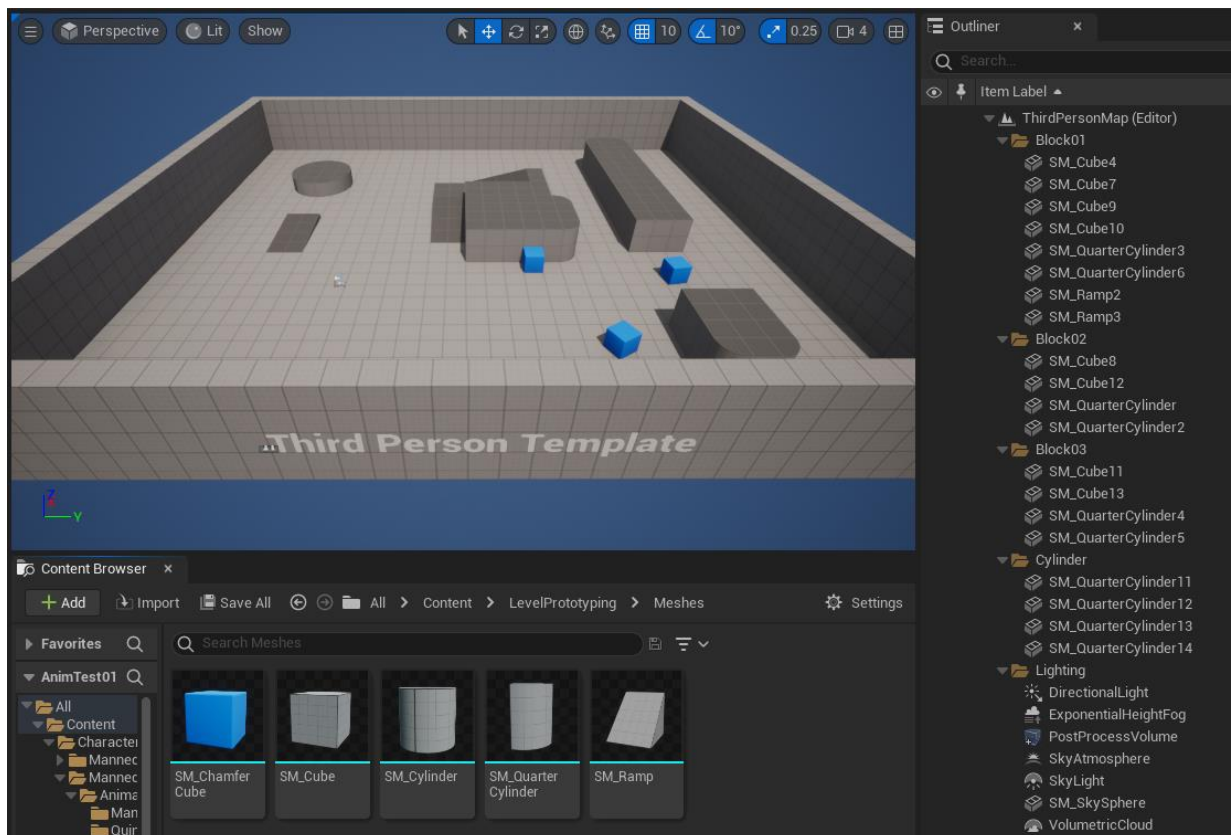
The Skeletal Mesh contains the rigged character. The Animation Sequence is an animation that is associated with a skeletal mesh and can be played on that skeletal mesh. Oftentimes, a character will have many individual animation sequences for the various actions that the character can perform. These often include idles, running, jumping, and more. An Animation Blueprint is then responsible for playing those Animation Sequences on a Skeletal Mesh.

In order to connect my character to both the player's inputs and the Animation Blueprint, I created a *pawn*



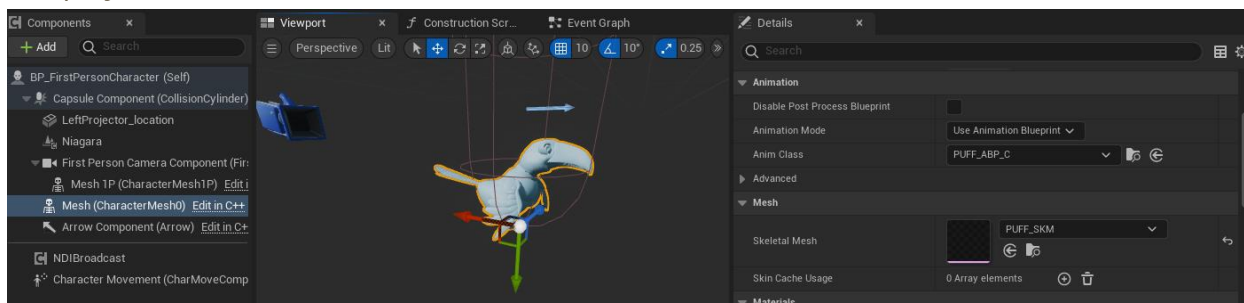
The default Third Person Skeletal Mesh

and placed it in the level. A pawn is a kind of actor that can be possessed and controlled by a player. An *actor* is an object that can be instantiated by being placed in a level. It may help to think of blueprints, actors, and pawns in an Object Oriented Programming sense. The assets in the content browser are the classes, while the assets that have been placed in the level are their instances.



*The assets in the Content Browser above are like classes, and the assets in the Outliner to the right are like instances.*

Inside the pawn, I associated the “CharacterMesh” with the skeletal mesh of my animatronic. This is also where I linked the Animation Blueprint I created earlier to the character. After placing the pawn in the level, it is able to be possessed and controlled as our playable avatar.



*The player pawn containing the skeletal mesh of my animatronic*



## Digital Figure to Physical Figure

For this project, I used Dynamixel motors. These “smart servos” have internal computers that store settings such as min and max position, motor ID, current position, and are controlled via serial communication. In order to move them alongside the digital figure’s joints, I needed to get the rotational values of the joints every game tick and map those numbers to the servo’s ranges. Then the new servo positional data could be packed into an instruction and sent over USB to the motors’ control board.



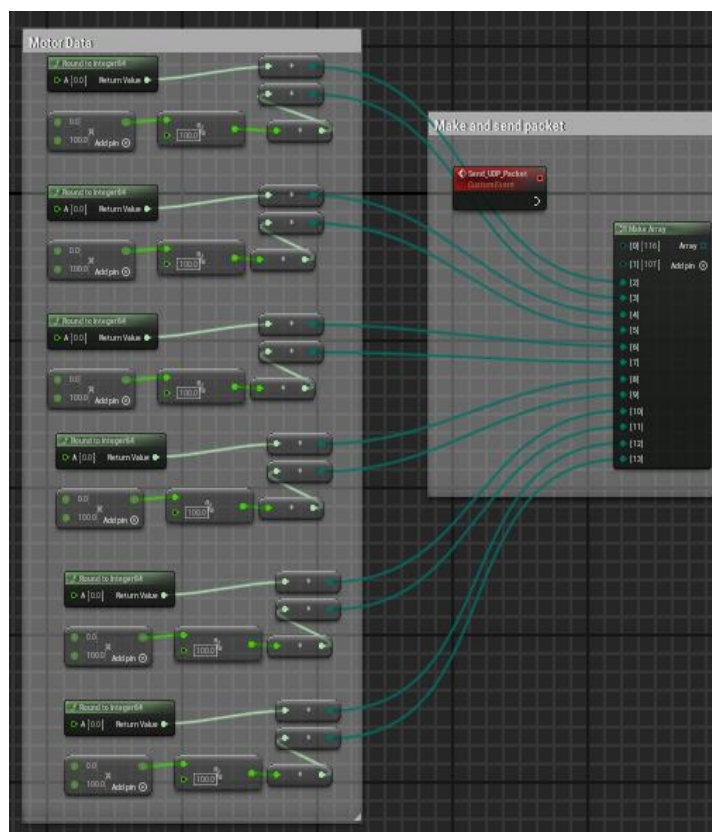
*An XL430-W250-T servo motor, one of the servos I am using on my figure*

I tried two different methods to get data out of Unreal and on to the motors. First, I used UDP to send data to a separate “in-between” program and the second, I created a direct serial link between Unreal and my motor control board.

### UDP Communication

UDP (User Datagram Protocol) is a protocol for sending data over a network. Unlike its counterpart TCP, UDP is useful for broadcasting and streaming data since it needs no confirmation that the data has been received. It merely sends data packets over the network without waiting for a connection to be established and confirmed by the listener. This is a method I have used in my internships to get data between Unreal to helper programs and is originally how I controlled my physical figure for this project.

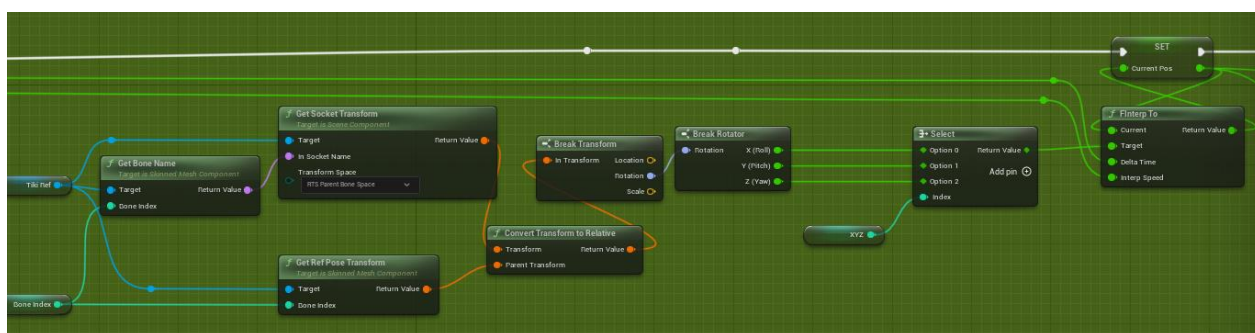
One of the benefits of using UDP is that your in-between program can easily live on a different computer than the one that is running the Unreal Project. This can be especially useful if your program is managing inputs and outputs across multiple systems. For example, if you were using a resource intensive input program (such as using camera input to send data back



*Data being packed into a byte array to send out of Unreal*

and forth to Unreal) your camera program could live on one computer and use UDP to talk to Unreal on a different computer. Since I am not really doing anything resource intensive in my in-between program, I let it live on the computer running Unreal and had two programs talk over the localhost (127.0.0.1) network.

To get the data out of Unreal, I used a UDP Plugin to stream the joint angles as I read them off of the character. In my level blueprint, I read the angle of the joints every game tick. As I iterated through each joint, I passed the angle through the “Finterp To” node to interpolate from the last read value based on the amount of time that has passed since the last tick. This helps to smooth the data and helps with framerate inconsistencies.



*Reading the angle of a joint and interpolating the desired position from the last tick*

Those numbers were then packed into an array of unsigned 8-bit integers and sent over the network. The final packet was structured like the table below and was a total of 14 bytes long.

Header 2 Bytes	Servo 1 Data 2 Bytes	...Servo N 2 Bytes
746B	[Byte 1 of servo 1 position] [Byte 2 of servo 1 position]	[Byte 1 of servo N position] [Byte 2 of servo N position]

My in-between program was listening on the designated port and expecting to receive packets exactly matching the standard above. Once it received a packet and verified that the header matched what it would expect, the program would break apart the data and send it to the servos using the Dynamixel SDK’s “Bulk Write” function. This function allows the user to write to multiple servos at a time using only one instruction packet.

```

currentPos = (float)udpClient.buffer[buffIndex];
currPosDec = (float)udpClient.buffer[buffIndex + 1] * 0.01;
buffIndex += 2;
currentPos += currPosDec;
currPosMap = Wing_R_Servo.convertAnimNumToServo(currentPos);
std::cout << currPosMap << std::endl;

// Allocate goal position value into byte array
param_goal_position[0] = DXL_LOBYTE(DXL_LOWORD(currPosMap));
param_goal_position[1] = DXL_HIBYTE(DXL_LOWORD(currPosMap));
param_goal_position[2] = DXL_LOBYTE(DXL_HIWORD(currPosMap));
param_goal_position[3] = DXL_HIBYTE(DXL_HIWORD(currPosMap));
servoID = Wing_R_Servo.ID;

// Add parameter storage for Dynamixel#1 goal position
dxl_addparam_result = groupBulkWrite.addParam(servoID, ADDR_GOAL_POSITION, LEN_GOAL_POSITION, param_goal_position);
if (dxl_addparam_result != true)
{
    fprintf(stderr, "[ID:%03d] groupBulkWrite addparam failed\n", Wing_R_Servo.ID);
}

// Bulkwrite the positions to the servos
dxl_comm_result = groupBulkWrite.txPacket();
if (dxl_comm_result != COMM_SUCCESS) printf("%s\n", packetHandler->getTxRxResult(dxl_comm_result));

// Clear bulkwrite parameter storage
groupBulkWrite.clearParam();

```

*A code block from my in-between program responsible for processing, packing, and sending the data received from Unreal*

This technically worked for getting data onto the motors but did not produce smooth enough results for my desired outcome. Although UDP is the fastest network communication protocol, it still introduces some delays into the overall data path from joint to servo. Also, since there are no checks, data can be lost. Another problem comes from Unreal's inefficiencies in regard to "Event Tick" and Blueprint Programming in general. This results in stuttered movements on the physical servos. This could be remedied by interpolation in my C++ program, but I decided to try to eliminate as much data loss from the source as possible. This led me to try a direct connection between Unreal and my servos.

### Serial Communication

In an effort to reduce any and all delays in communication, I wanted to create the most direct path from joint to motor. To do this I decided to integrate my motor controller with my Unreal project and avoid using blueprints as much as possible. Unreal lets you program with C++ instead of blueprints, but if you want to communicate between the two, you need to use Unreal's requirements for coding. These limit the kinds of data types you can use and although Unreal often provides alternatives to these types, they also can present frustrations and slowdowns for me. This was especially challenging when I needed

to incorporate Dynamixel's CRC (checksum) calculation code into my own.

```

unsigned short SerialSender::updateCRC(unsigned short crc_accum, unsigned char* data_blk_ptr, unsigned short data_blk_size)
{
    unsigned short i, j;
    unsigned short crc_table[256] = { ... }

    for (j = 0; j < data_blk_size; j++)
    {
        i = ((unsigned short)(crc_accum >> 8) ^ data_blk_ptr[j]) & 0xFF;
        crc_accum = (crc_accum << 8) ^ crc_table[i];
    }

    return crc_accum;
}

```

*Dynamixel's CRC calculation code. It contains data types unsigned short (16-bit int), arrays, and other data types native to C++ but not Unreal.*

Since I still wanted to keep my blueprint that reads joint angles intact but also wanted to use types and functions that Unreal would not allow, I had to find a work-around. I created a public C++ file that would serve as the bridge between Unreal and the regular C++ code I needed. I created a C++ module named "DataBridge" for this purpose. This file contains a list of joint data that is updated by Unreal every tick.

I then created a C++ module called "SerialSender" and placed it in the directory of my DataBridge files. SerialSender does not interact directly with my Unreal project, so I'm able to use all the data types and functions I need. It merely receives data from DataBridge as it is updated by Unreal.

Another method I employed to speed up the flow of instructions to my motors was multi-threading. SerialSender runs on a dedicated thread and interprets and sends data to the servos as fast as it can. It checks every cycle to see if the motor's data in DataBridge has been updated. If so, it stores that data as the new

```

#include "DataBridge.h"
#include "SerialSender.h"

// Sets default values
ADataBridge::ADataBridge()
{
    PrimaryActorTick.bCanEverTick = true;
}

// Called when the game starts or when spawned
void ADataBridge::BeginPlay()
{
    Super::BeginPlay();
    _sSend = new SerialSender();
    _sSend->startThread(1);
}

// Called every frame
void ADataBridge::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}

void ADataBridge::stopThreadSender() {
    _sSend->StopThread();
}

void ADataBridge::updateAnimPosArr(int ID, float animPos) {
    _sSend->updateAnimArr(ID, animPos);
}

```

*My DataBridge file that updates SerialSender's target servo positions*

target position and begins incrementing the servo towards that target. Since it is running on its own thread and can make up for missing data, the flow rate of data-to-servos is independent of Unreal's tick.

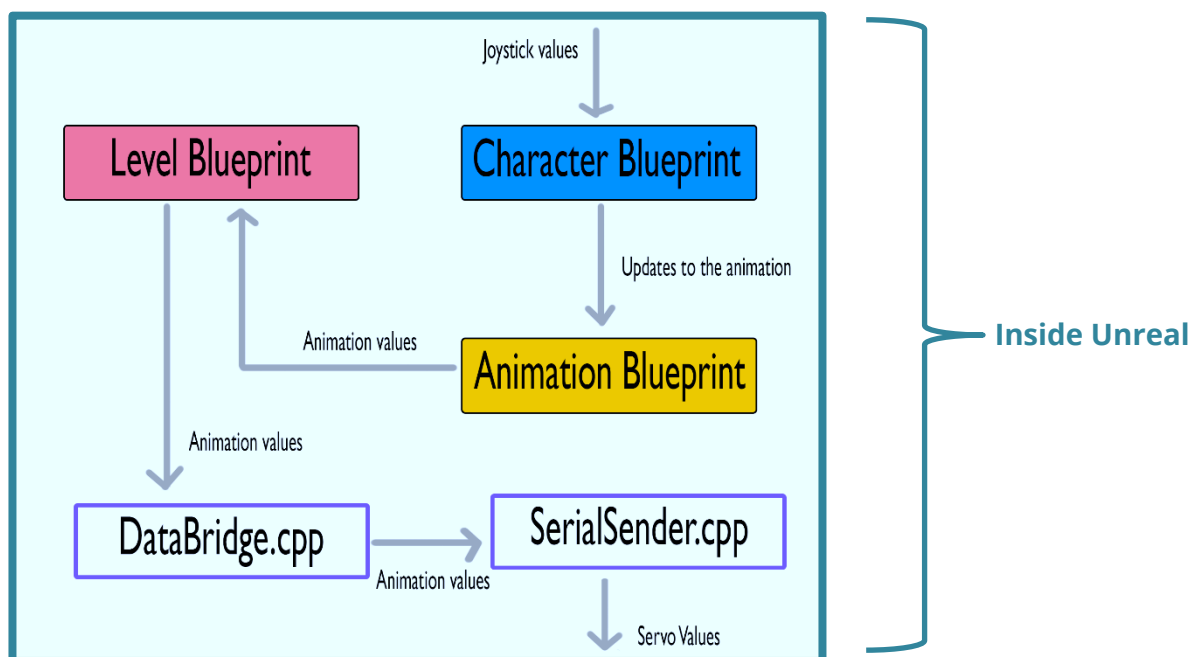
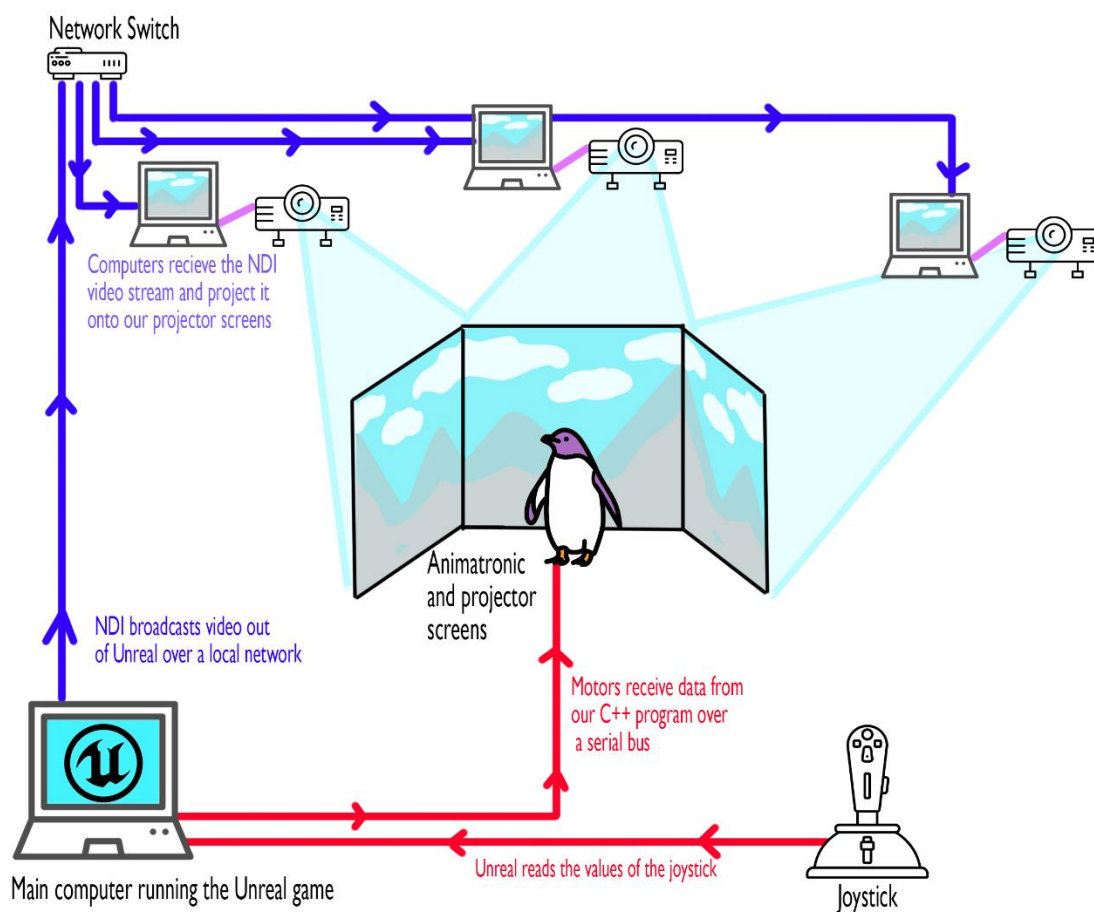
Due to some technical issues, I decided to forgo the Dynamixel SDK and simply create the serial data packet myself. I only needed to write the goal position of the servos and therefore only needed to create one kind of instruction packet. This is similar to what I was doing before with the UDP packet, but this time with a stricter standard and more data to be sent. Luckily, Dynamixel provides extensive documentation that makes this process easier. Here is the structure of the Dynamixel "Bulk Write" instruction packet that I used:

Header 2 Bytes	Reserved 1 Byte	Packet ID 1 Byte	Length 2 Bytes	Instruction 1 Byte	Parameters N Bytes	CRC Checksum 2 Bytes
0xFFFFFD	0x00	0xFE	[Length of Parameters]	0x93	[Servo ID, Instruction Addr, Data Length, Data to be written] repeated for every servo	[Checksum calculation]

The end result is a 118-byte packet containing all of the 12 servo's positional instructions being sent as fast as the program can run. This resulted in much smoother motion and is the method I used in my final product.



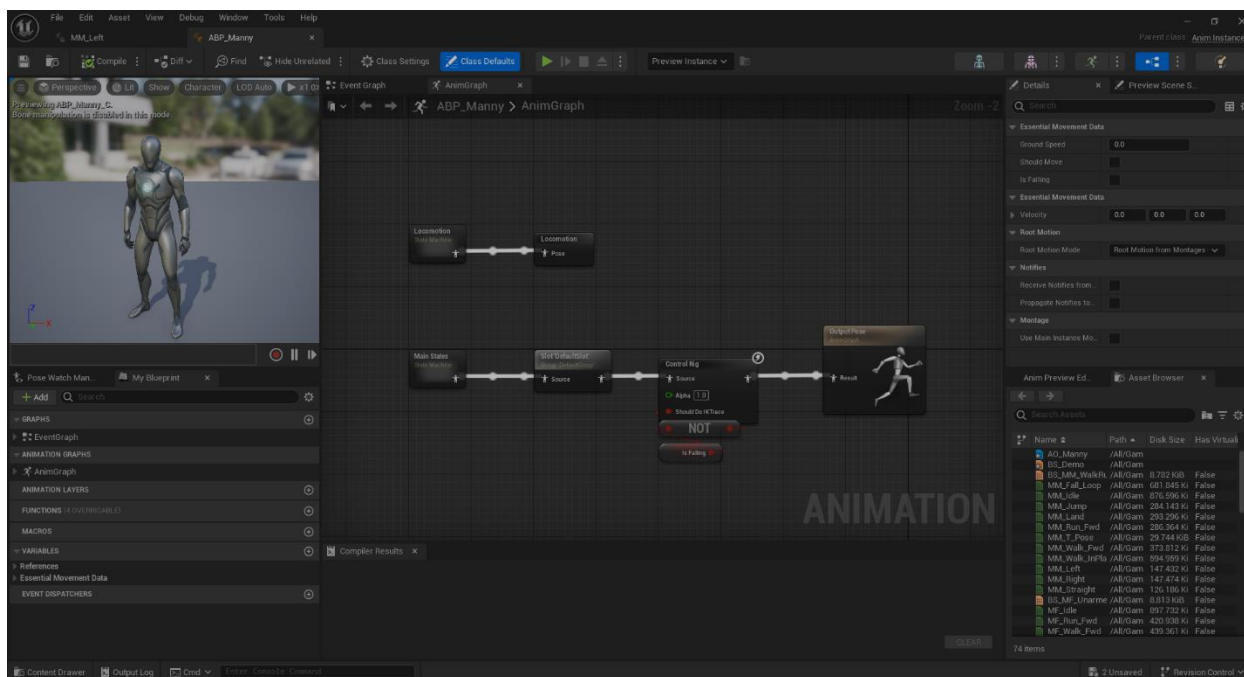
## Game Layout and Flow of Data



## Using Unreal's Animation Features

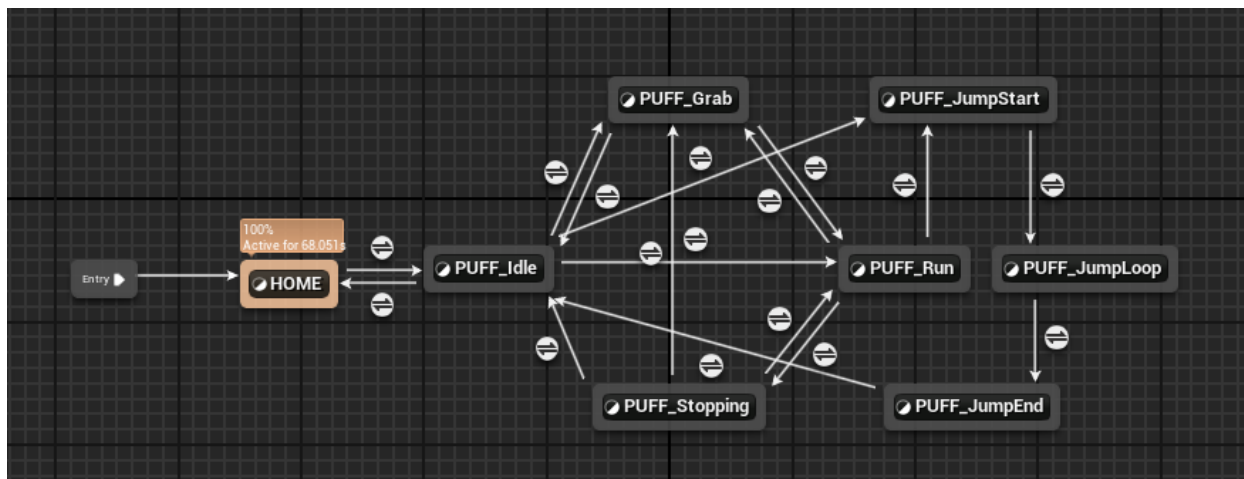
I used an Animation Blueprint to create the behavior of my character. The Animation Blueprint is responsible for playing the various Animation Sequences associated with the Skeletal Mesh and provides various tools for creating realistic and responsive motion. It also provides a “brain” for the character, changing the character’s behavior depending on different inputs.

To produce these results, the Animation Blueprint makes use of Unreal’s various advanced animation tools. These tools include State Machines, Animation Blending, and Aim Offsets. This section will cover each of these tools, what they are, how they are normally used, and what specifically makes them beneficial for animating animatronics.



*The default Animation Blueprint for Unreal's mannequin character*

## State Machines



*A State Machine within an Animation Blueprint containing various actions for a playable character*

### What is this feature?

State Machines make up the “brain” of the Animation Blueprint and contain the roadmap for which animation to play and when. This allows the character to respond in real-time to various inputs. A State Machine lives inside the Animation Blueprint and consists of a network of *states* and *transition rules*. Each state contains an output node labeled “Output Animation Result”.

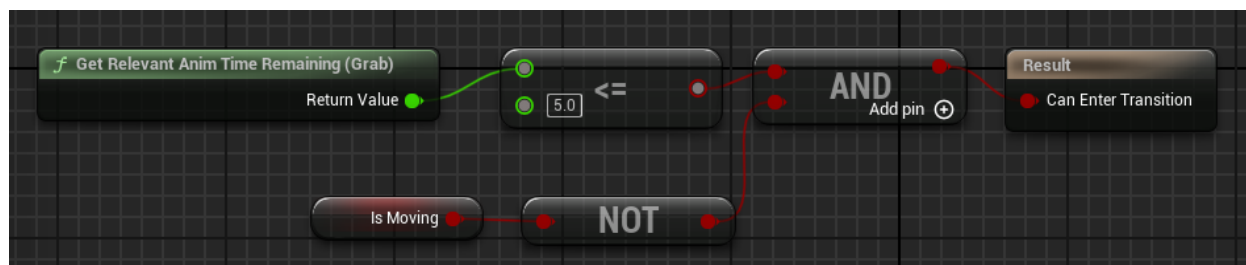


*Each state contains the animation that will play while in that state*

Here we can connect an animation sequence from our list of available sequences to the output node. This means that while the character is in that state, it will be playing that linked animation.

We can also change the behavior of the animation by selecting it in the grid and editing the settings in the Details panel to the right. This includes behaviors such as starting frame, playback rate, and whether or not to loop.

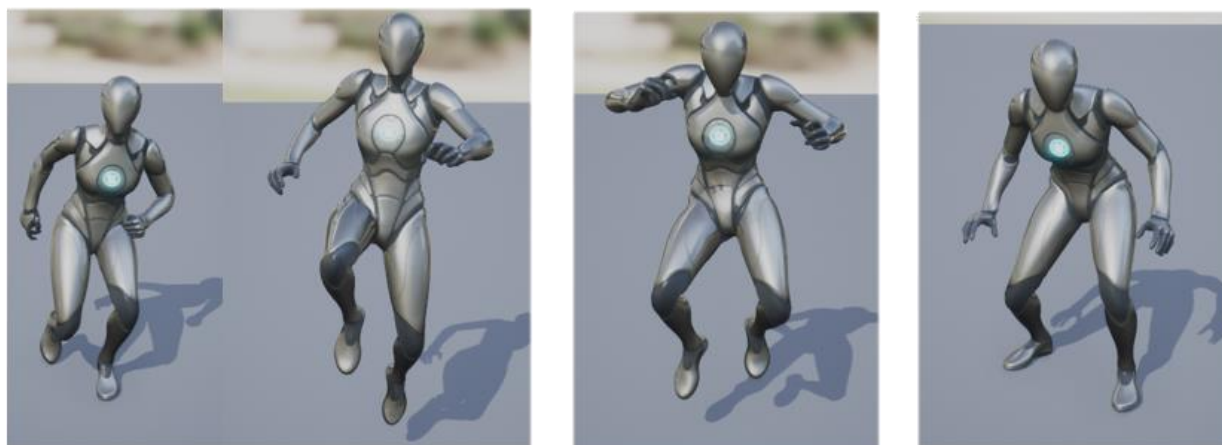
Transition rules contain the logic to switch between animation states. These often include Boolean variables that are manipulated by other blueprints to indicate when it is time to switch states. Once the parameter “Can Enter Transition” has been passed a “true” value, the character will switch to the next linked state in the direction of the Transition Rule.



*A Transition Rule that will return true if the player character is stationary and the current animation has finished playing*

How is this feature normally used in game development?

State Machines are often used to link the playable character's animations with the input received from the player. This can include playing running and jumping animations when the player uses their game controller to move the character around. It is also useful for handling transitional animations. This often occurs when an animation loop of variable time needs to play. A common example is a jump animation. This animation looks like one smooth motion, but it's actually made up of three separate Animation Sequences: jump, falling, and land. The amount of time that a character is falling and, therefore, needs to loop the falling animation, will depend on how high of a platform they jumped from.



*From left to right: two frames of **jump** animation, one frame of **fall**, and one frame of **land***

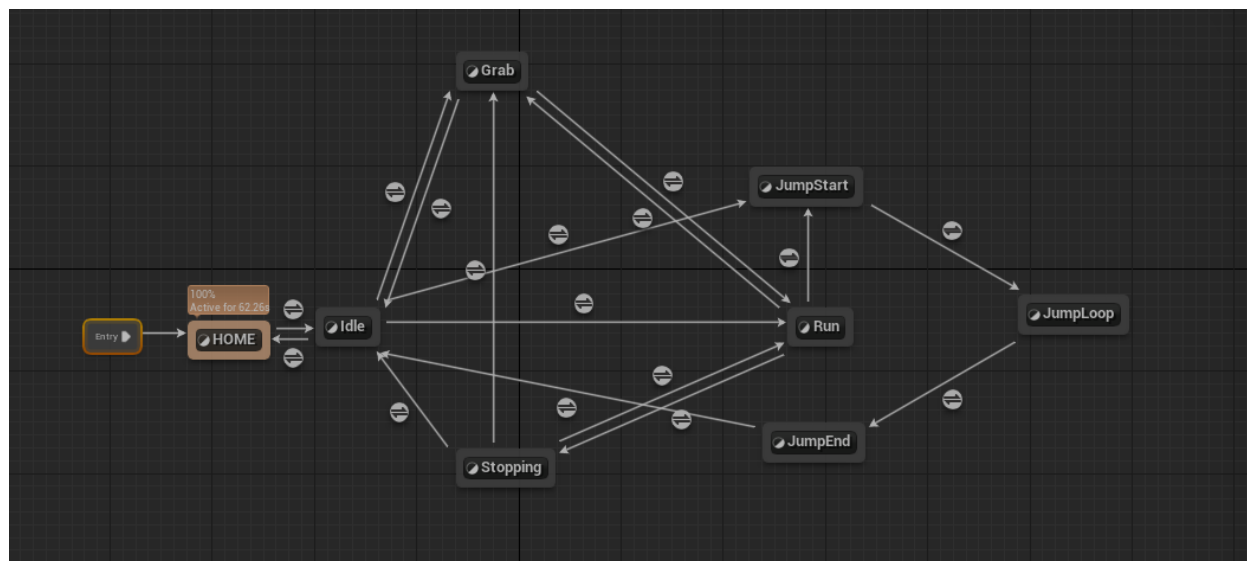
State Machines allow developers to smoothly transition and loop the fall animation until the character hits the ground. Jump, fall, and land exist as separate states often with the flags "isJumping" and "isFalling" set by the pawn's blueprint in the Transition Rules.

## What problem does this solve for Animatronics?

By giving our character a brain, we're able to interact with them in real-time. Like most animatronics controls of the modern age, each company has their own way of creating these animation features – such as state machines. This often involves a developer(s) programming a state machine the traditional way. It's not too difficult to do but since Unreal provides the necessary grunt work of doing the math to blend and play the animation sequences, it allows developers to focus more on the final outcome behavior rather than the mathematical way to achieve it.

## How was this used in my thesis demonstration?

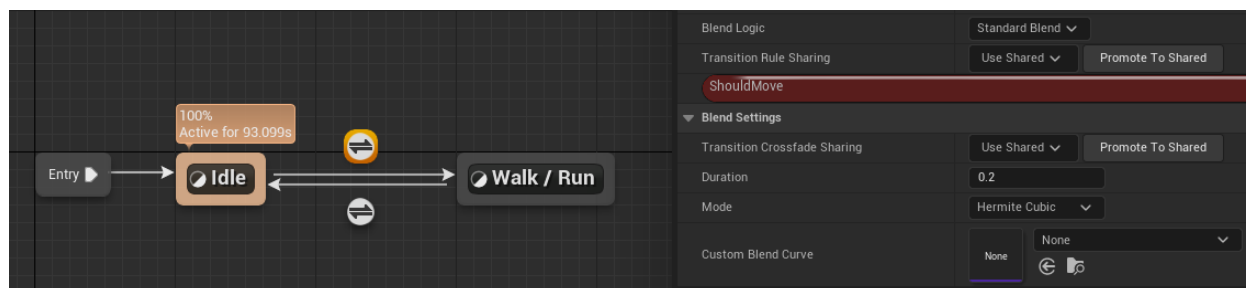
For the Pebble Game, I used a state machine to determine Bean Bag's animations based on the joystick's inputs. As the player maneuvered Bean Bag around the level, the level blueprint read the players inputs and relayed that information back to Bean Bag's animation blueprint via its Boolean variables. As those variables updated, the state machine would transition to the appropriate state and thus play the corresponding animation on our character.



*Bean Bag's State Machine*



## Animation Blending



*A simple state machine showing the blend settings between Walk/Run and Idle*

### What is this feature?

Animation Blending is a simple tool that allows the developer to smoothly transition between the end of one animation and the start of the next. This includes looping between animations and interrupting one animation to begin another.

The blend settings are found by selecting one of the Transition Rules in the State Machine and looking in the “Details” panel to the right. There are plenty of settings to customize blends, but I chose to use the sinusoidal mode for my character. This results in a blend that accelerates between positions at a sinusoidal rate and is one of the most common mathematical ways of blending animatronics’ motion in general. I also varied the duration of the blend depending on the animation. I tweaked this number by trial and error until I found a realistic result.

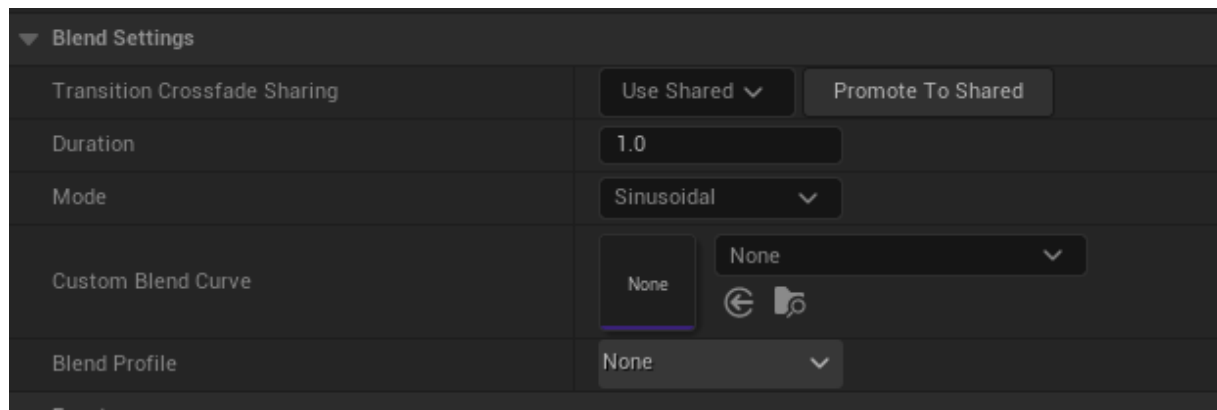
### How is this feature normally used in game development?

Animation Blending is one of the most commonly used animation tools. Any time the player character jumps, sprints, stops, and crouches, an Animation Blend is used to smoothly transition between those actions. It’s also commonly seen when a player approaches and talks to an NPC. Blending lets the NPC seamlessly transition from their idle animation to their talking animation.

### What problem does this feature solve for Animatronics?

Just as an NPC would “jump” between animations in an unnatural manner without Animation Blending, an Animatronic figure would jerk. Not only does this result in an unnatural movement, but it can lead to motor damage if the difference between starting and ending positions is great enough. Usually if an Animatronic controller doesn’t have a blending function, then the character must start and end in the exact same pose. This is fine if a character is merely looping through one animation but can present challenges if that animation needs to be interrupted at any given point.

As with State Machines, developers will often code their own blend functions to achieve the same results. However, just as Unreal provides the necessary work to provide State Machine functionality, Unreal has also already done all of the necessary work to create blend functions – thus saving precious time for the developer.

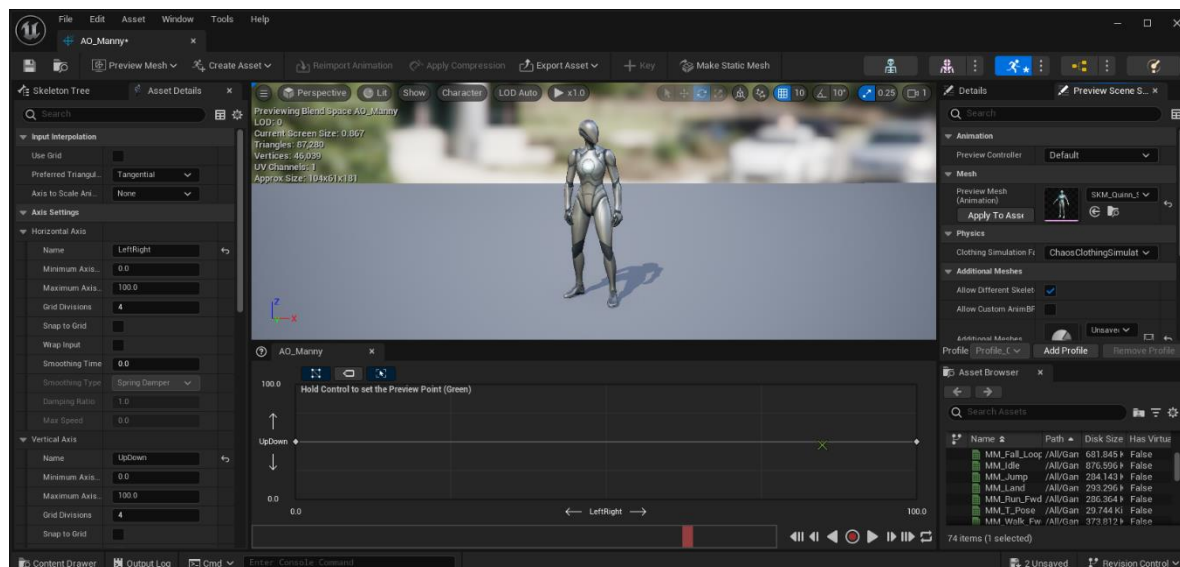


*Blend settings used in one of Bean Bag's transition rules.*

How was this used in my thesis demonstration?

Every time Bean Bag started and stopped moving, picked up a pebble, deposited a pebble, and switched between any other action, an animation blend was used to smooth the transition between those animated states. Each blend was created and tweaked via the details panel each transition rule in the animation state machine.

## Aim Offsets

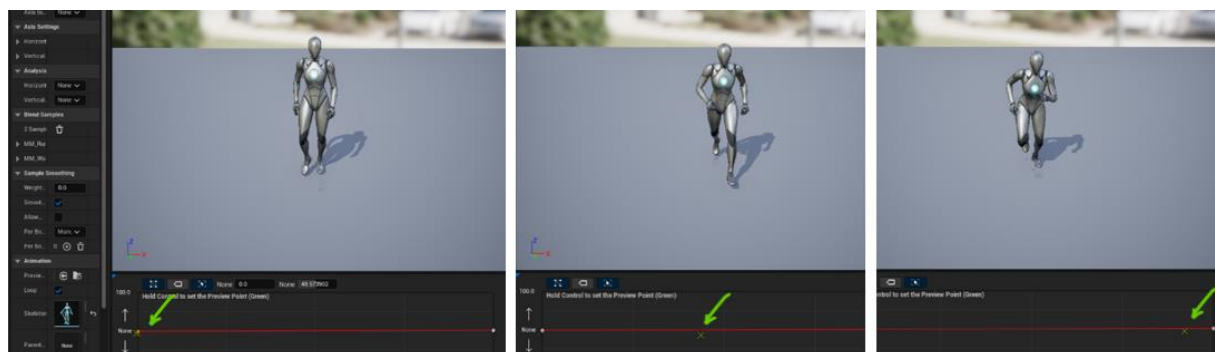


*An Aim Offset affecting the direction in which a character is looking while their idle animation plays*

What is this feature?

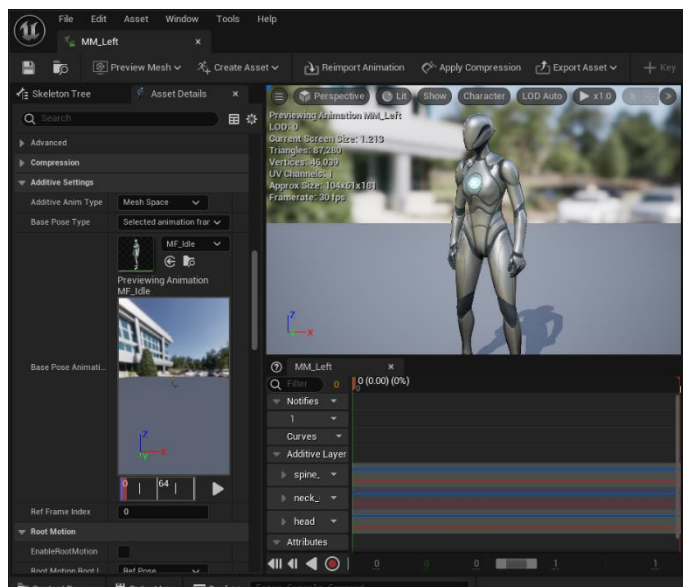
Aim Offsets provide a quick and easy way to control the direction in which an animation is focused. An Aim Offset asset is essentially a graph containing various animation sequences with each axis of the graph controlling the weight of influence that the closest animation asset has on the main animation currently playing on the character. Unlike a *Blend Space* — in which the graph is meant to serve as a gradient between entirely different animations — Aim Offsets use *Additive Animation Blending* to merely influence the way that a certain animation asset is behaving.

For example, a Blend Space may be used to adjust how much of the “walk” versus “run” animation should be playing based on the speed that a character is moving. An Aim Offset may be used to control the direction in which a character is aiming their gun while that walk or run animation is playing.



A Blend Space with a “walk” and “run” animation showing how the value of the X axis on the graph (noted by the green arrow) affects the output animation.

Also, unlike a Blend Space, an Aim Offset often uses animation assets consisting of only one frame. To create these single frame assets, developers often make duplicates of the base animation that they want to affect and delete all but the first frame. They then manipulate and key frame the desired bones to create the poses that will influence the final character's animation. Oftentimes, this means creating a pose for facing the left direction, and one facing the right. These poses – or single frame animation assets – are then given the correct additive settings in the Asset Details panel and added to our Aim Offset. The picture to the right shows an example of this single frame animation using the default mannequin character. In this asset, the mannequin has been posed looking to the left with another asset created for the mannequin to look to the right. Both of these assets are then additively blended onto the mannequin's idle animation via the Aim Offset. The final result is an idle animation that can be naturally and dynamically directed to face a desired way.



A single frame animation of a character looking to the left. The additive settings have been set to "Mesh Space" with our desired base animation asset selected below

How is this feature normally used in game development?

One of the most common uses for Aim Offsets, and the feature they were named for, is allowing a character to aim a gun in a direction independent of the currently playing animation. This means that no matter if the player is running, jumping, or standing still, the character can look around and aim their gun in any direction.

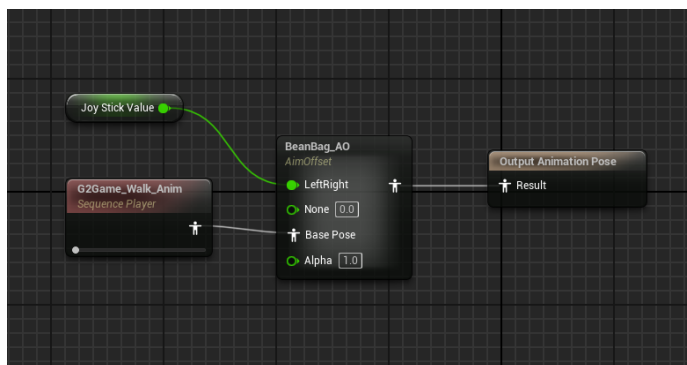
Another common use for an Aim Offset is to allow character's heads to follow certain directions. This can often be seen when approaching NPC's in modern games. The NPC's head will usually follow a player as the player moves around them.

What problem does this feature solve for Animatronics?

It's extremely useful to be able to direct an animatronic's animation towards various points in real-time. If someone is supposed to be interacting with a figure, it would be quite unnatural for the figure to not be looking at them while they interact. In the past, some animatronic programmers have resorted to creating entirely separate animations for each

of the directions that the animatronic needs to look. This is extremely inefficient and can be easily accomplished by using Unreal's Aim Offsets.

How was this used in my thesis demonstration?



Bean Bag's Aim Offset implemented in his Animation


As Bean Bag moves around the level, his head and body twist to point in the direction of the joystick. This is done using an Aim Offset affecting his head turn and chest turn functions. The Aim Offset axis values are updated in the Animation Blueprint by the values read from the joystick's twist axis. I also used an Aim Offset to aid in his

girlfriend's reactions, allowing her to respond in his direction when he is near. This helps create a natural connection between the two when they interact.



## Chapter 4: Conclusion





With this thesis I was able to explore several ways that the Unreal Engine could aid in animatronic controls--specifically by providing robust animation tools to aid in creating the final movement of my figure. The animation tools I focused on include State Machines, Animation Blending, and Aim Offsets. While these animation tools provided me with everything I needed to create my interactive game, there are plenty of other animation tools and capabilities that Unreal provides. I didn't end up using them for my thesis, but they could be very useful for other kinds of animatronic projects.

Aim Offsets let me point Bean Bag in the direction that he was moving, but Blend Spaces and the "Layered Blend by Bones" could also be used to dynamically influence a base animation. These could be used to create an interactive gradient between animation assets or alter an animation only on a selected part of the figure. An example use could be incrementally using more of a running animation instead of a walking animation depending on how fast the character is going.

While there are many more tools in Unreal to explore, there are even more possibilities for their uses. One of these uses could be to aid in live puppeteering. Instead of the traditional way of controlling each individual motor to create the final look of the figure, a puppeteer could use something like a game controller and Unreal to influence the motions and behavior of the puppet in a much more intuitive and natural way. The inputs from the game controller, instead of directly altering the motion of a single motor, could be used as inputs for the Animation Blueprint. Simple motions like moving a joystick left and right or pressing a single button could result in complex motions from the figure. This makes it much easier to learn and perform on the puppet and can lead to much more complex figures being controlled by a single puppeteer.

Unreal provides so many potential improvements for the field of animatronic controls and I believe the possibilities will continue to grow as the technology improves. Perhaps Unreal's physics simulations could be used to create even more realistic reactions on a real-world figure. Maybe Unreal and NDI's real-time video streaming could greatly aid in creating reactive emotions on figures with projected faces. As the themed entertainment industry looks towards expanding the stories we can tell with animatronics, I believe that Unreal can provide the necessary tools to make those stories a reality.

# Glossary of Terms

**Actor** – An Unreal object that is instantiated by being placed in the level.

**Animation Blueprint** – A roadmap in Unreal for which animation to play on a character, how to play it, and when.

**Blueprint** – A term Unreal uses to refer to their visual programming files.

**Blueprint Grid** – The window panel displaying nodes and their connections in a blueprint.

**Bulk Write** – A function provided by Dynamixel to write to multiple servos using only one instruction packet.

**Checksum** – A sequence generated by running an algorithm on a piece of data in order to verify that the data is complete and unaltered.

**Dynamixel** – A brand of servo motor that uses serial communication to read and write values to its memory.

**Event Tick** – The “heartbeat” of the Unreal game

**Level Blueprint** – The main blueprint associated with the level (or “map”) of your game.

**Mesh** – Geometry of a virtual object.

**Node** – A visual piece of code that is added to a blueprint. These can include functions, variables, operators and more.

**Non-Player Character (NPC)** – A character in a video game that is not controlled by the player.

**Pawn** – An actor that can be possessed and controlled by a player.

**Serial Communication** – a communication protocol in which data is sent one bit at a time, often over a bus of wires.

**Skeletal Mesh** – An asset in Unreal containing a rigged character.

**UDP** – A kind of network protocol often used to stream or broadcast information.

## Works Cited

Epic Games. *Unreal Engine 5 opens new doors for architectural visualization*. Unreal Engine.  
<https://www.unrealengine.com/en-US/blog/unreal-engine-5-opens-new-doors-for-architectural-visualization> :

Epic Games. *USA's largest car maker builds HMI systems in Unreal Engine* Unreal Engine.  
<https://www.unrealengine.com/en-US/blog/usa-s-largest-car-maker-builds-hmi-systems-in-unreal-engine>

Industrial Light & Magic. "The Virtual Production of the Mandalorian Season One" *YouTube*,  
 23 February, 2024, <https://www.youtube.com/watch?v=gUnxzVOs3rk&t=1s>

Parzival. "Meerkat Short Film from Weta Digital | Unreal Engine 4" *YouTube*, 23 February,  
 2024, <https://www.youtube.com/watch?v=Pf4TwsBggdY>

"P-p-pick up a Perfect Penguin Pebble | Edinburgh Zoo." Edinburgh Zoo, 3 Feb. 2017,  
[www.edinburghzoo.org.uk/news/article/12549/p-p-pick-up-a-perfect-penguin-pebble](http://www.edinburghzoo.org.uk/news/article/12549/p-p-pick-up-a-perfect-penguin-pebble)

Walt Disney Company. "Walt Disney Interacts with an Audio-Animatronics Tiki Bird"  
 ResearchGate, [www.researchgate.net/figure/Walt-Disney-interacts-with-an-Audio-Animatronics-Tiki-bird-in-The-Enchanted-Tiki-Room\\_fig1\\_220475304](http://www.researchgate.net/figure/Walt-Disney-interacts-with-an-Audio-Animatronics-Tiki-bird-in-The-Enchanted-Tiki-Room_fig1_220475304).

