Authoring Procedural Quadruped Gait in Game Engines With Input

Trent Riek

## Abstract

Video Game environments & interactive media are increasingly becoming more complex with their animation systems; Game engines now support systems that include built in logic, interpolation, and additive animation that capture the complexities of characters. Yet some challenges remain; In most cases animations are still constructed externally outside of the game engine, instead being created in a separate animation program. This creates a disconnect from interactive gameplay with predefined animations. In this project, I explore animation in the Unreal Game engine and create game engine solvers to author realistic & interactive movement for a quadruped (a Llama) within the game engine with no external help.

## Acknowledgements

I would like to thank my committee members for being patient with me as I discovered what my final goal of this project was and all of the requirements needed to complete it. In particular I thank Tim for Introducing me to procedural animation; it sprouted an interest in understanding the kinematics of animals, and furthered my hobby of learning about taxonomy.

# 

# Introduction

Video Games aesthetic quality & complexity has evolved over the last two decades dramatically. Video Games went from simple 3d models made of a dozen geometric objects, to simulating huge environments with detailed surfaces, lighting, physics, and AI systems. In fact, video games are often referred to as simulations, to the point where it is hard to tell the difference. Both (can) include a virtual environment which simulates real-world phenomena, and players who can interact with said environment and each other[1]. This is particularly true for the visual aspects - games can support many realistic effects from refraction of glass to the forces acting on cloth in the wind. One such aspect is animating creatures - animal’s bodies can move correctly to walk around a gameworld(the environment). To move around the gameworld often means the player controlling an avatar, either a human or some other animal, that can walk and run or otherwise move themselves around the level. NPCs (Non-Playable Characters) must also walk and move around the world. Today Video games simulate character movement very well. Games can make player avatars interact so closely with the environment that they can walk up stairs with their feet touching each step, put their hand on a doorknob and turn it to open a door, and so on. *Video games can simulate organisms* - at least pretty well.

But, this animation has limits. Not everything can be simulated perfectly, in particular the movement of *live organisms*. Live creatures have multiple modes of locomotion, each with a sophisticated musculature built from eons of evolution. The movement of organisms is incredibly complex and must be abstracted in video games. Video games are limited by both hardware constraints (the amount of calculations that can be done) and *gameplay* constraints. Gameplay is the engagement of a player in a game - games are defined by constraints/rules, and gameplay happens when a player makes choices based on these rules for a desired outcome [2]. When video game developers design a game, they must make the environment follow the game rules - often directly contrasting the results of a pure simulation. In a first person shooter game the player must switch between running, crouching, strafing and shooting a gun at the click of a button, even if switching between these movements in milliseconds is biologically impossible. The gameworld is filled with these switches, making the dynamic simulation contrast the game’s rules. This means animations have to somehow obey rules of the simulation, along with the rules of gameplay. Typically, gameplay trumps simulation - the avatar previously mentioned must follow the gameplay before following the virtual environment [3]. This poses a hard challenge to make animation look as correct as possible.

In this paper, I attempt to create correct animations that both are reactive to the environment and gameplay input. I dissect the key components of quadruped movement and choose methods that abstract the process to smaller elements. I abstracted the body movement into three separate sections; leg movement, body movement, and a gait system. I create solvers that procedurally create poses for the leg and body based on foot locations. I then create an independent gait system, designed to drive the foot movement based on the current gait and character speed as well as update the current gait. This produces natural transitions, and a character that can interact with the environment well.

# 

# Background & Related Work

## Current animation model

Normally to solve the aforementioned problem, video games borrow from their neighbor field of 3D animation. Games are built in game engines, and animated films are made in animation software. For most games, the character animations are designed in an external 3d software and then imported into the game engine. These animations are then played based on the current game input. This solves the problem for gameplay, but is not very effective for the simulation. Thus in many games, additive animation occurs. An animation is loaded, and then modified based on environment parameters. For humanoid characters. Still the effectiveness of this approach is limited. For bipedal characters, this works fine; by finding simple rules that the majority of the animations can follow which when added up creates emergent behavior that is realistic [3]. But this is only for bipeds. The animal kingdom is vast and has a plethora of body types, and implementation of these different characters often is lacking. Multiple games do feature complex tetrapod animals and have taken great lengths to animate them correctly. A great example is Red Dead Redemption 2 - this game features horseback riding, and the developers took great care in animating the horses body movement, leg position, and gait[4]. Even in this last approach, the base movement of the horse was driven by imported pre-made animations.

An ideal system would be able to have all animation be driven by the *game engine alone* with no pre-made animations, that would allow the character to all possible scenarios of both gameplay and environment simulation. In essence, the animation is procedurally generated, which means the animation is created based on given inputs which can change. In this paper, I make an attempt at accomplishing this goal. I focus on procedurally animating an ungulate animal. Ungulates are the taxonomic clade that refers to hoofed mammals and are often featured in games (such as horses and cows). These animal’s animations are often overlooked in development (the focus is on the human characters). In particular, I animated a Llama. I chose a Llama because I like Llamas (and they are not animated nearly as frequently as horses).

## Ungulate Gaits

There are multiple components to analyze for Ungulate movement. This includes the type of locomotion (what the animal does to move around) and the movement of the body parts to achieve this locomotion. The type of locomotion is the gait - A *gait* is a pattern of walking for an animal, in what order they move their appendages in order to move in a direction. Almost all creatures in the animal kingdom have a gait. Animals with four feet, two feet, multiple feet, or even animals without legs can have gaits - however gait typically refers to biped or quadruped movement. Through evolution animals have evolved complex muscular movements that propel them forward as efficiently as possible. Thus it is important to first identify the gaits used by our animal, and then the muscle/skeletal movement of the body. It is also helpful to understand Primarily how character animation is accomplished in games. Characters in computer games are often displayed as a collection of connected polygons and vertices, animation of these polygon based characters in computer games are thus often realized and simplified by means of skeletal animation[5] which we will discuss later.

Gaits are typically defined by footfall; where the animal's feet land during a certain period of movement.This period is called a cycle. A single cycle is determined by the time it takes for all feet to move, the time of which the feet move is referred to as a beat. Gaits can also have several other properties. Gaits are either being symmetrical or asymmetrical: the left and right side either alternating movement time or moving at the same time. Gaits can have diagonal movement: the hind foot moves and then the alternate side front foot, and have other miscellaneous properties. Gait types for most animals fall into a couple of common names; the *walk*, *amble*, *trot*, *pace*, *canter*, and *gallop* [6].

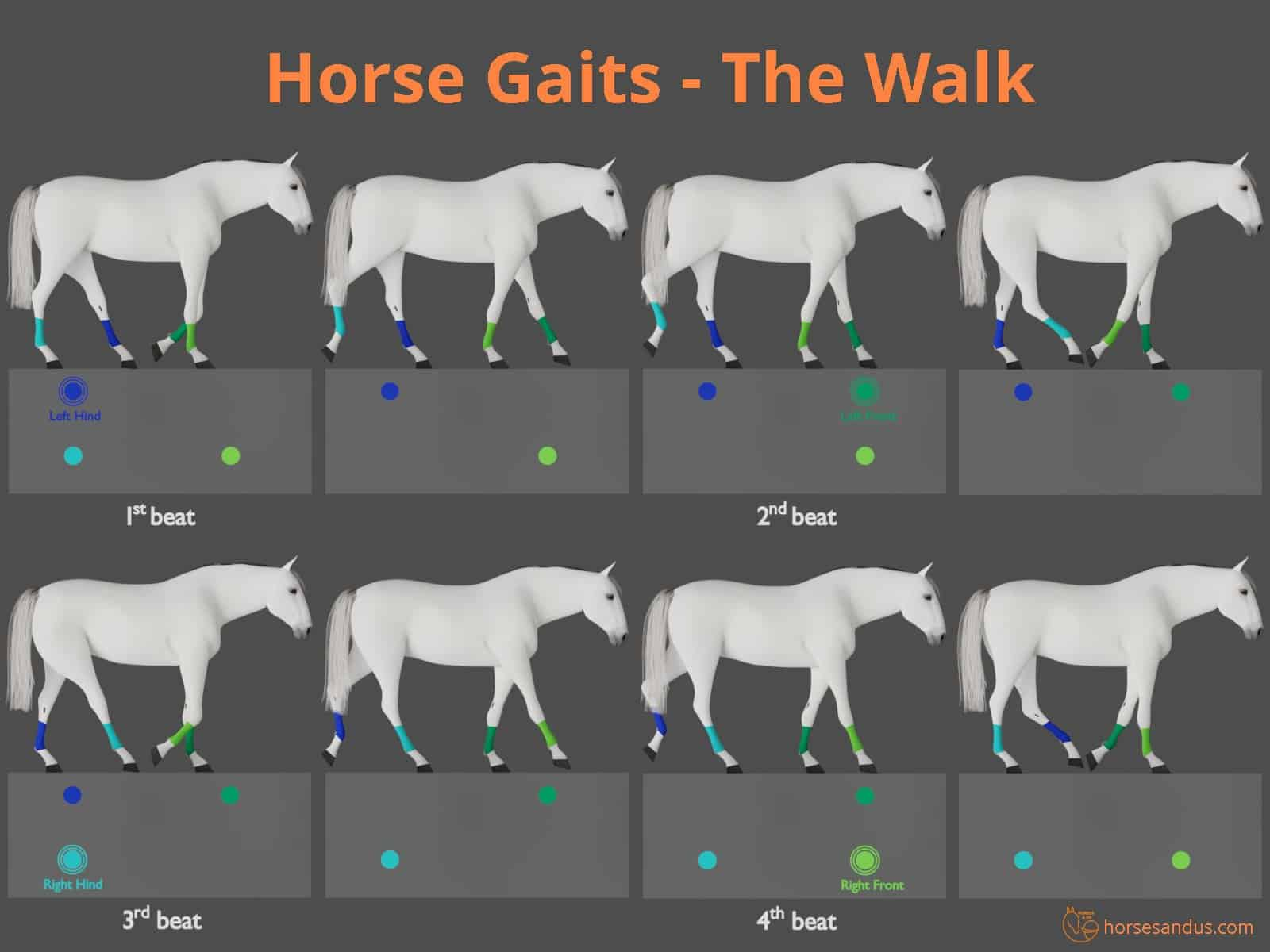


Figure 1: An example of a walk gait. The walk is a slow, symmetrical, four-beat gait. It is characterized by four distinct beats, where each hoof lands on the ground at a separate interval, independent of the other hooves[11]. Note that this is very similar to other ungulates; Llamas have the same walk.

All of these can be achieved by most mammals, but typically the weight position and diet of the animals determines which they use. Large herbivores have legs used only for running and thus use the amble more frequently for example, compared to carnivores with legs that are multi-functional to both move and attack. Ungulates use the majority of these gates, but the majority of data found for ungulate movement is with horses. There is data for others that can be synthesized into data too [7], but many gaits are shared between species and thus we can derive most ungulate gaits from horse data. For a Llama, their primary forms of movement are the walk, the pace, and the gallop. The Llama walk is a symmetrical lateral gait, where each side alternates moving. Two limbs alternate with three limbs in supporting body weight [6].

The type of gait is determined primarily by speed and weight distribution. It has been observed that the movement of bipeds can be considered like an “inverse pendulum”. This can be extended to ungulates who can be considered like two inverted pendulums for the front and hind legs respectively [8]. While some animals evenly space their timing and weight distribution between their pairs of legs, ungulates achieve faster gaits through footfalls that are unevenly spaced in time, and by having an unequal mass distribution between the fore and hind quarters. The rotation of the leg joints can also be determined through a sinusoidal pattern using FDM (Fourier Decomposition Method) [7]. The weight distribution of the body will also follow this sinusoidal pattern, the back and front independently oscillating between a low and high point [8]. This rotational data can be calculated from motion data captured from real animals. This has been a frequent attempt at this problem in fact procedural animation has been generated from Motion Capture Data before for multiple animals [7,9].

## Approach

The above approach does not work perfectly with our objective however. In particular, the creation of animation in real time and responsive to gameplay. Motion capture data is a constant linear set of data that creates a single result; It does not account for all scenarios and does not account for varied player input. In video games there are several approaches to accomplish real-time animation. The method we discussed above that was rotation based can be referred to as *Kinematics Driven*. There’s *Dynamic Driven*, where joint rotations are applied a force/torque used to move the body forward [5]; and *Pose Driven*, where the skeleton’s bone rotations are calculated based on a desired “final pose” already pre-defined.

A pose-driven approach is best suited for our needs for a few reasons; For starters, foot fall in gaits can be calculated as an independent position unrelated to the rest of the body movement [9] which correlates well with the game world where the direction of the character changes (thus the next position of the foot). Plus, this works well with the traditional way 3d models are animated in games. Animating in video games (or any 3d software) is primarily done with a *skeletal system*. The surface involved is attached to an abstracted skeleton, where each joint has a certain amount of influence on parts of the model’s surface. Typically an animator with “hindsight” knows the point a limb is moving towards, meaning inverse kinematics is an effective way of calculating where an appendage should go. IK systems are frequently used to make characters reactive with an environment; this can also be used here. This function that calculates all rotations is called a *solver*. It would make sense to create a leg and body solver that gets the best rotation for the joints based on the food location.

It is worth briefly noting the first attempt at this started as a class project (VIZA 615); the objective was for the most part the same. This was completed within the unreal Game engine using the aforementioned approach, using built in Unreal systems. The results are shown [here](https://youtu.be/sq-XVrWwnP0). While this did capture full body movement for the character, gait transitions, and reactiveness to the environment, there were several issues; In particular

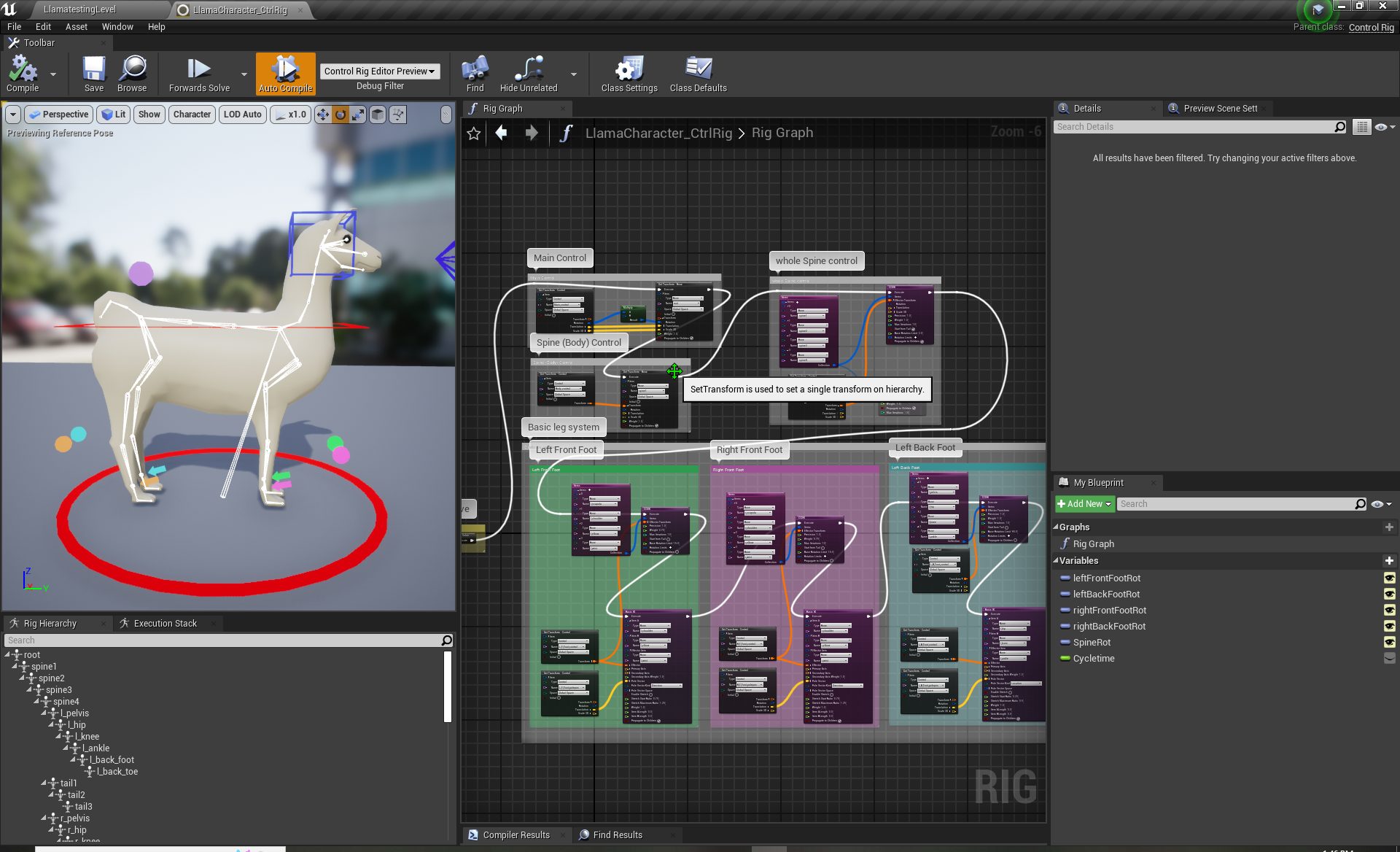
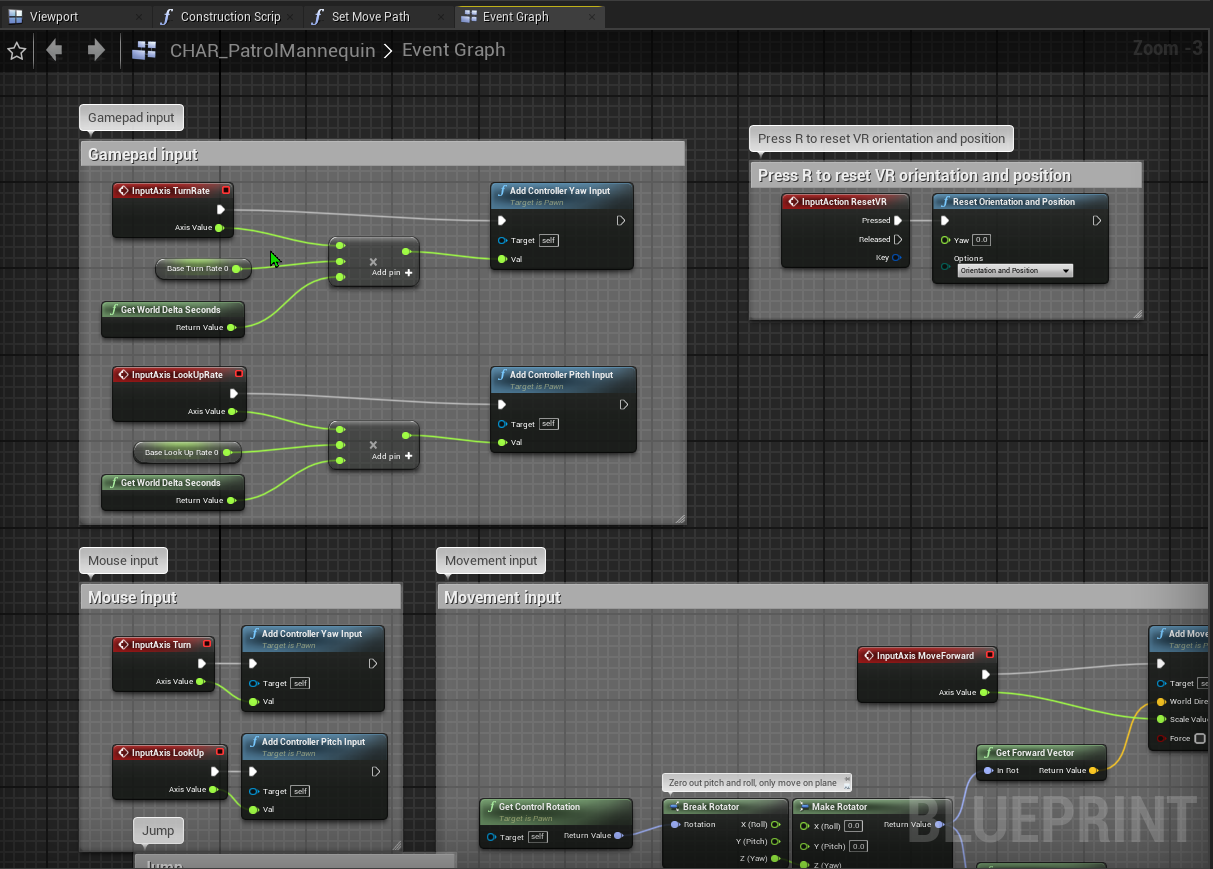
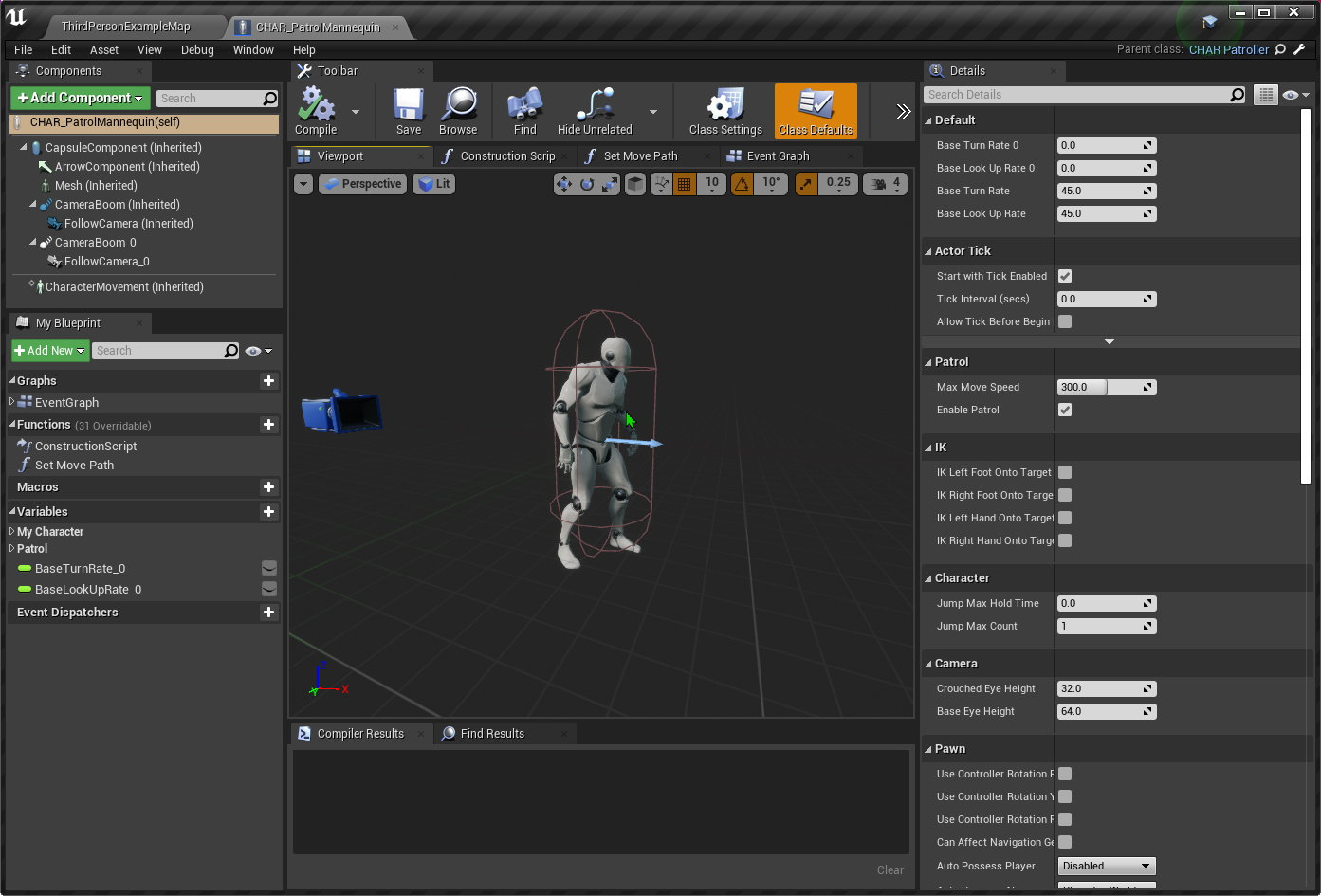


Figure 2: Unreals Control Rig plugin as of 4.26. Note the Two large nodes for each foot; They are a CCDIK solver and an IK solver. Not ideal, does not sufficiently animate the character and is complicated.

# Implementation

For the implementation, the Unreal game engine was chosen. This engine has a well flushed out animation suite that allows for custom animation systems to be easily built. This is done through The Unreal’s Blueprint system. This scripting system uses nodes to create game logic; methods and properties are nodes that can be plugged into each other to create an execution sequence The user creates game logic within objects called actors, that reside in the 3D environment, called the level. This in particular is where the logic behind the gait is written. This is important because the player’s character is *a separate entity from the animation system*. In essence, the player controls an entity from which the gait system collects data. But the character movement is not dependent on the gait.



Figures 3 & 4: An example of Unreals Blueprint system. To the left is a scene interface which displays the object being created (the actor); the right showcases the node based scripting.

Unreal has unique Blueprints for animation systems; this is where a solver resides to calculate the skeletal movement of the Llama. Unreal has pre-build solvers; however, these solvers are insufficient to calculate proper leg movement. FABRIK and CCDIK solvers, while efficient and robust, can be taxing and more importantly do not respect the limited joint constraints/directions the joint can rotate (the knee cannot bend inwards and so on). Thus a custom solver will be programmed in C++, the language Unreal’s engine is built with.

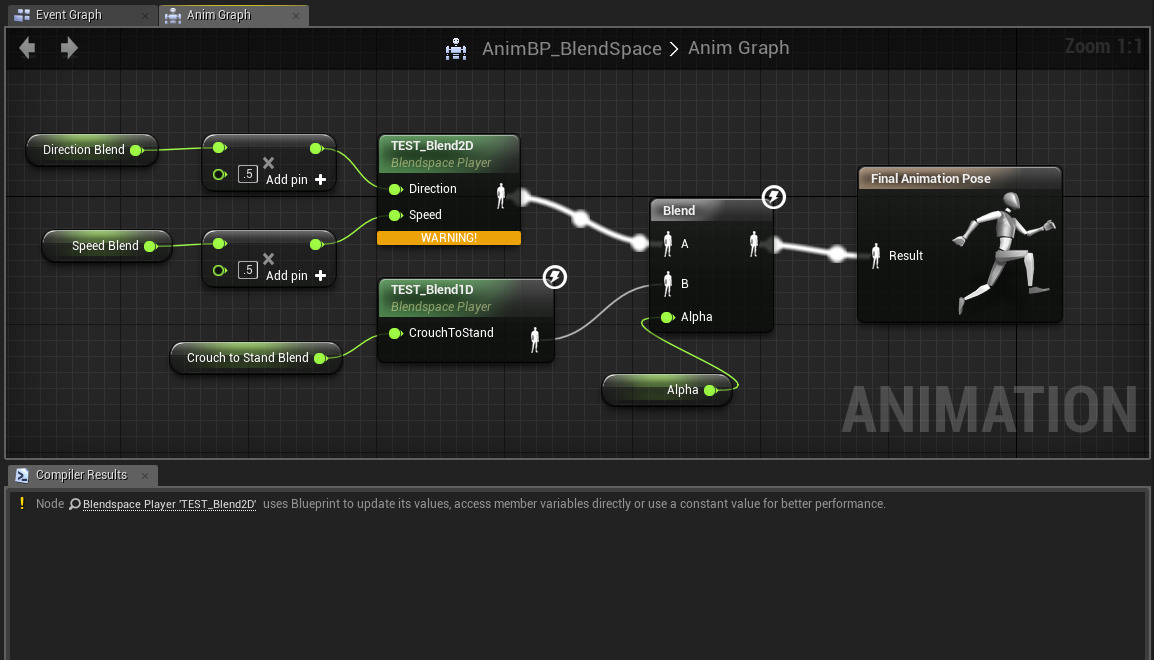


Figure 5: An example of an Animation Blueprint system. Note the Green middle nodes that are individual animation, being blended into a final output pose for the skeleton.

## Gait

To Start, a gait system is implemented. This was done within an actor component - an element of an actor that accomplishes a task. To make the gait, there were a couple of objectives; first the user must determine if the character is moving and thus is using a gait, what gait should be used, the current time of the gait, and its influence on the feet. In particular, what foot is moving and creating the foot’s cycle. The component’s tick continuously runs these operations:

| tickUpdate(deltatime){  CurrentVelocity = GetSpeed(deltatime);  UpdateCycle(deltatime);  For (foot in feet){  UpdateFootLocation();  CheckShouldFootMove();  }  } |
| --- |

The gait collects the forward vector of the character as well as the difference between the previous position last tick and the current tick, which determines the current velocity. Based on this, the cycle is updated: A cycle is a timer that updates anytime the llama moves. Its speed is consistent until the cycle completes, after which it will determine the current speed of the Llama and adjust the cycle’s speed and gait. It will then allow the cycle to run again (assuming the llama is moving). Once the cycle is updated, the individual feet are updated in a for loop. The feet also have their own cycle that is Instantiated at the correct time in the cycle with a constant speed - but is not dependent on the Llama’s velocity and so continuesIn order to constantly keep the Llama’s feet on the ground. If the Llama’s foot is not currently completing, a separate check determines the current beat in the cycle and what foot should move. The key here is determining the final location vs the current location - in this case, a simple lateral raycast to the ground in a position n mm in front of the Llama determines the next foot position. This position is also offset based on whether the llama is turning or not. This speed is factored into the gait. The *UpdateCycle* method compares the speed at the end of a cycle: at cycle end, check which gait’s minimum speed value *n* is below the current speed. If the speed is within the gait’s range, choose that gait.

| UpdateCycle(deltatime, gaitData){  if( isCycling ){  cyclePosition += currentSpeed \* deltatime;  if(cyclePosition >= totalCycleTime) {  isCycling = false;  cyclePosition = 0;  }  }  else{  For each( gait in gaitData ){  If (speed > gait.MinnimumSpeed){  set currentGait;  set isCycling;  }  }  } |
| --- |

## Skeleton

The Llama Model was constructed in Maya, the model takes on a semi-real look with an approximate realistic size. The Skeleton’s root starts at the front of the spine; a connecting joint where the neck and front legs all meet up. The spine is abstracted to 4 joints along the top, with 3 joints for a tail. The head has the appropriate amount of vertebrae (7), the head bones being bastracted to a head joint, a nose joint and a jaw joint. The legs were given more detail. The bones that make up the shoulder of the Llama are represented; the Scapula, Humerus, Pelvis, and Femur. The radius, MetaCampus, Tibia and Cannon are represented in the lower half of the leg. These bones are required for correct leg movement. In summary, typically 3d Model skeletons can be simplified into smaller bone numbers; but here the exact bones for the legs were needed.

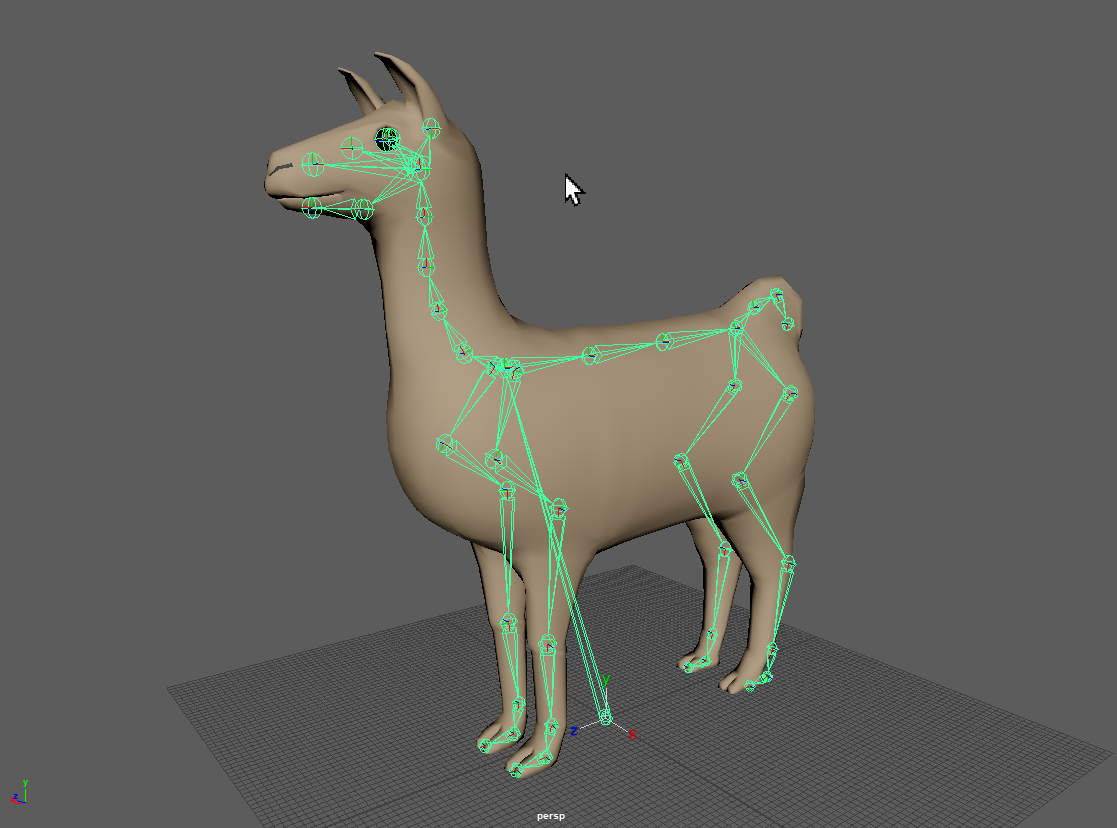


Figure 5: An image of the Llama Model in Maya displaying the skeleton

## Leg Solver

Animating the model starts with solving the leg; upon observation of Horse Gait and movement, a few conclusions were made. The leg of a horse can be viewed as an “upper” and “lower” part; the shoulder moves, and extends to do the majority of the rotation for the leg. The lower end of the leg then completes the extension, rotating the foot to clear the ground and creating a sort of two bone ik. Thus for the purposes of this project the solver worked in two parts; first it calculates a total rotation of the scapula/shoulder joints for extension, then completes the foot with a two bone IK solve. In particular, it forms a part process; 1) get two rotations for the top shoulder bones, then 2) get the final positions of the foot, and lastly 3) calculate the knee ik from the shoulder and foot pos.

| LegSolve(){  CalculateShoulderRotation();  CalculateFootLocation();  CalculateLowerLegIK();  } |
| --- |

To calculate the Scapula/Humerus rotations (shoulder), take the foot’s resting location and compare it to the target location. The farther the distance, the more rotation. If it’s forward, the shoulder extends (rotates forward); if negative the shoulder contracts and rotates negative. Note that the scapula and humerus have opposite rotations; as the scapula rotates forward, the humerus rotates negative & vice versa. This increases extension. The up movement of the location acts like forward rotation. For implementation, *i* is the initial foot position and *T* is the target foot position. The X and Z values are separate and multiplied by constants, after which are then normalized to a range to confine the rotation to a maximum Value *M* and minimum value 0.

| XValue = NormalizeToRange( (abs(Tx - ix) \* XConst), Mx)  ZValue = NormalizeToRange( (Clamp(Tz-iz) \* ZConst), Mz) |
| --- |

After this, the Final foot location can be determined; ideally the foot location should just be the target location, however we need to check to see if that target location is actually reachable with the full extent of the Llama leg. Thus the total length of the leg is calculated; Instead of simply taking the maximum length of the joints, the distance between the scapula and cubical (elbow) joint is used along with the rest of the leg. In essence, the size of all the bone vectors are added up except the humerus, and checked for length. Size is the function to get the magnitude of a vector.

| TotalLegLength =  Size(Scapula - Elbow) + Size(Elbow - Knee) + Size(Knee - Ankle) + Size(Knee - Foot)  If (TotalLength > Distance(Scapula, T) {  FinalFootPos = Normalize(LookAtVector(Scapula, T)) \* TotalLength  }  Else {  FinalFootPos = t  } |
| --- |

Finally a two bone ik can be solved for the elbow-knee joint. A backwards foot rotation can also be added based on the Zvalue (to mimic the foot pickup that Ungulates complete. This completed the leg movement. The Back legs can also be treated the same, albeit with much smaller Maximum M values to reduce rotations (the back leg also has minimal IK movement).

## Body Solver

The Body, as mentioned previously, is like two inverted pendulums; two points oscillating on top of the legs. By taking the average up value of the current foot position, we can get an average spine beginning and ending value. Then a FABRIK solver can resolve the in-between Spine Locations. This FABRIK Solver can also solve for head movement; while the head moves, it tries to keep its head at a constant value for less jarring movement. Below the ZLoc initial position of the spine joint. A constant is also added to adjust for unnecessary ground offset.

| FrontSpineLocation = (LFFootZLoc + RFFootZLoc / 2) - ZLoc - Const  BackSpineLocation = (LBFootZLoc + RBFootZLoc / 2) - ZLoc - Const |
| --- |

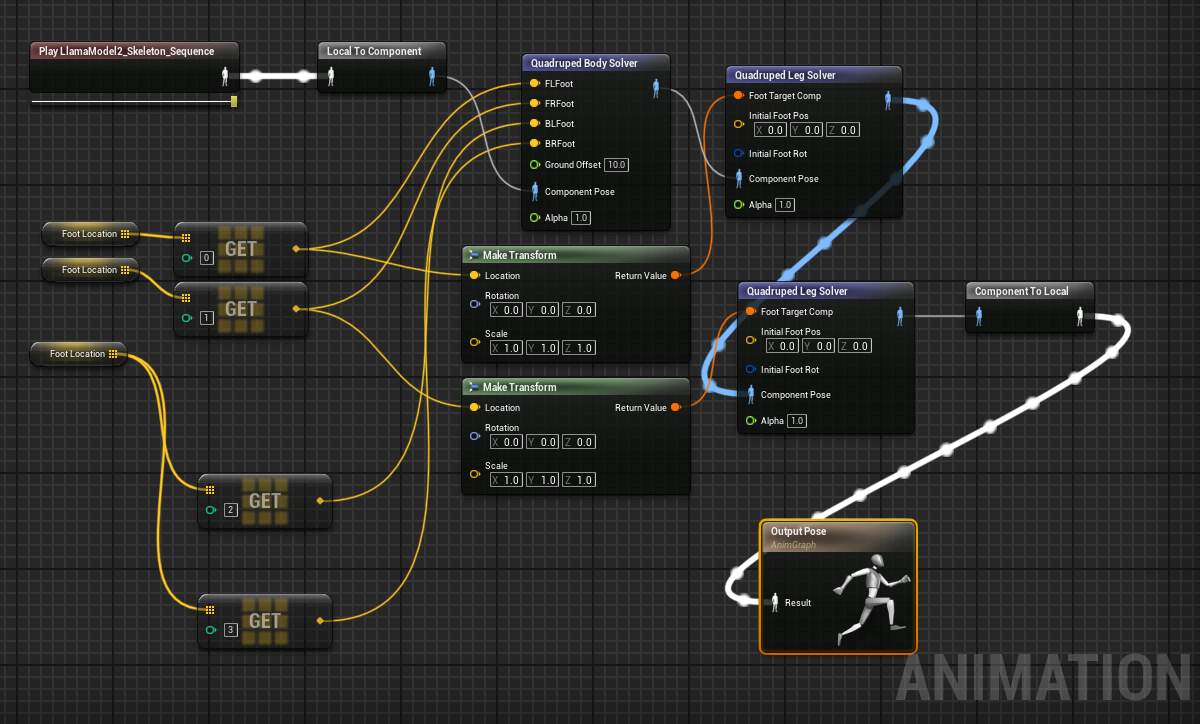


Figure 6: The final Animation System. The purple nodes are the ones implemented

# Conclusion

The gait system allowed for the legs to continuously find footing on an uneven terrain and properly place the body above. By having the feet as always oscillating as long as a gait was active, or when the character was moving, the model stayed “grounded” when moving around. The system in place allows for the user to choose their style of game input entirely independent of the animation system, in line with the objective to keep the musculoskeletal simulation separate from the gameplay. The system properly rotates shoulder and leg joints, and transitions gait appropriately. The system continuously updates the animation regardless of the user input as expected.

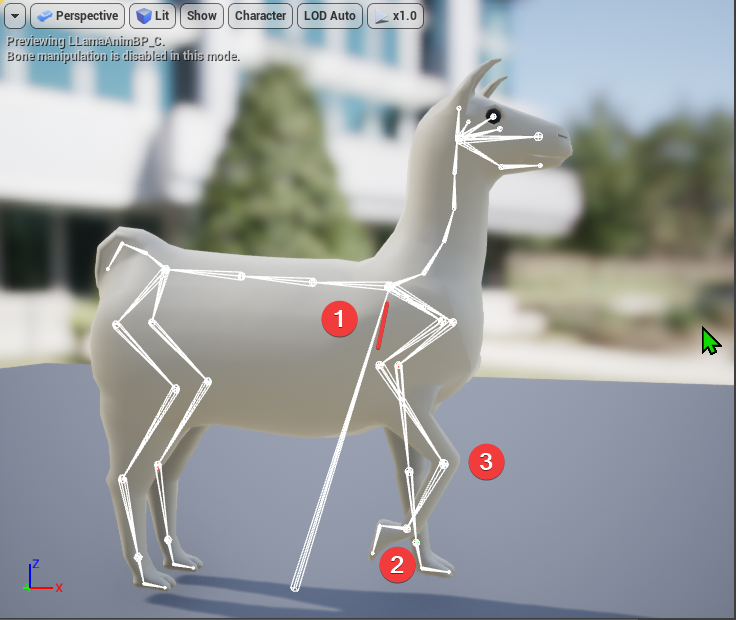


Figure 6: Displaying the Leg solver. The order of operations is noted: the Scapula/Humerus is calculated, the foot is put at the target, and then a normal ik is solved for the knee.

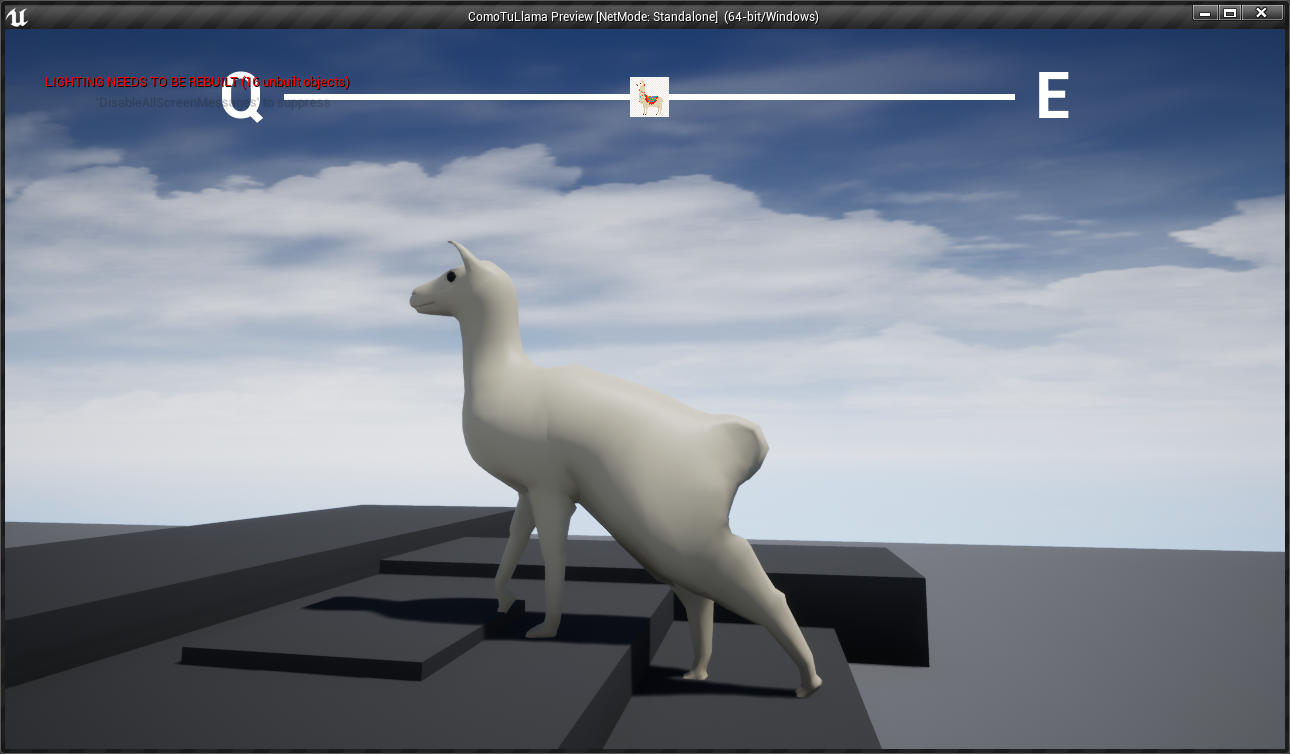


Figure 7 & 8: demonstration of the leg solvers on complicated surfaces.

While the system works overall, there are several areas of improvement. The system can still apply additive animation to create more detailed character movements (such as the movement of the head or added side to side spine rotations). While the system works, it is still prone to errors when the raycasts fail to find a place on the floor. Other issues related to the feet rotation on the ground can also be improved. The spine, while overall creating an effective rotation, was very problematic due to the order or operations. Skeletal animation depends on a skeleton hierarchy where bones are children of another; in this model the root bone was the front spine, the chest area. Properly animating the bones in the correct order proved to be a challenge along with the limited documentation for Unreal’s animation system in c++. Also, an improved character model and texture would help visualize the movement better.

The results serve to show a difference from traditional animation means of using pre-fabbed animation imported, and relying on a combination of these animations and additive ik solvers. How this system is implemented, it can still utilize this previous method; but currently serves as a demonstration of procedural animation for quadrupeds. By using a gait system that tracks the speed input of the player and cycles feet location and gait transitions, more natural animations can occur during runtime. Hopefully with added AI sensing systems and improvement to the spine oscillation, This character can move completely by itself and match the quality of hand mode animations.

# References

1. Sauvé, L., Renaud, L., Kaufman, D., & Marquis, J. S. (2007). Distinguishing between games and simulations: A systematic review. Educational Technology & Society, 10 (3), 247-256.
2. Frome, J. (2019, January 1). Interactive Works and Gameplay Emotions. Games and Culture, 14(7/8), 856–874.
3. Animation Bootcamp: An Indie Approach to Procedural Animation. (n.d.).   
   Retrieved July 20, 2022, from <https://www.gdcvault.com/play/1020049/Animation-Bootcamp-An-Indie-Approach>
4. AI Summit: Making the Believable Horses of “Red Dead Redemption II.” (n.d.).   
   Retrieved July 22, 2022, from <https://www.gdcvault.com/play/1027113/AI-Summit-Making-the-Believable>
5. Larsson, J. (2015). Performance of Physics-Driven Procedural Animation of Character Locomotion: For Bipedal and Quadrupedal Gait.
6. Gaits. (n.d.). Retrieved July 22, 2022, from <http://vanat.cvm.umn.edu/gaits/index.html>
7. Mielke, F., Van Ginneken, C., & Aerts, P. (2020). Quantifying intralimb coordination of terrestrial ungulates with Fourier Coefficient Affine Superimposition. Zoological Journal of the Linnean Society, 189(3), 1067-1083.
8. Griffin, T. M., Main, R. P., & Farley, C. T. (2004). Biomechanics of quadrupedal walking: how do four-legged animals achieve inverted pendulum-like movements?. Journal of Experimental Biology, 207(20), 3545-3558.
9. Zhou, J. (2013). Perception based gait generation for quadrupedal characters (Doctoral dissertation).
10. Edwards, L. F. (1939). A Critical Analysis of the Movements of the Shoulder-Joint. The Ohio Journal of Science, 39, 83-93.
11. Maria. (2020, December 10). The 4 Basic Horse Gaits Explained [Diagrams & Animations]. Horses and Us. https://www.horsesandus.com/the-4-basic-horse-gaits-explained/