CS354 Final: Procedural Animation Player Package

By: Anthony Hoang

Overview

For the final project of Computer Graphics, I wanted to explore inverse kinematics and its application in procedural animation in runtime environments such as video games. For this exploration, I will implement FABRIK from scratch on the Unity Game Engine version 2019.4.2f1 in C# that can be applied to any model exported into the game. Once this inverse kinematics system is implemented, I'll look into implementing a player package that will utilize this system to create procedural animations, specifically on the legs, tail, and arms. In order to create this player package, I will be using the Wolf model from Super Smash Bros Brawl with the original rig from the game. This player package will be used in a simple third person shooter game in which the player shoots at 10 red targets to get points.

Inverse Kinematics - FABRIK

Before going into any of the functionalities for the game or the player package, an inverse kinematic system must be implemented for the program to easily control limbs and bone chains through simple movements. There are many inverse kinematics algorithms to choose from, but I decided to choose Forward and Backward Reaching Inverse Kinematics (FABRIK) because of its simplicity, efficiency, and widespread use in games.

FABRIK is an iterative, convergence algorithm that looks at bone chains and limbs as joint positions on a line. The algorithm has two parts within a loop in this order: backwards reaching inverse kinematics and forward reaching inverse kinematics. In backward inverse kinematics, the program will have two vectors of joint positions representing bone chains: the original bone chain that's currently there and the working bone chain to be created. Immediately, the program will set the final joint position to the target position that the user can set in 3D space. Once this final position is set, we work backwards from this final joint to the root of the chain to calculate this new working chain. To get the joint for the new bone chain ($j1_{i+1}$), the program uses this formula from the original joint calculated in the previous iteration (j0) and the joint we're looking for in the original bone chain:

$$j1_{i+1} = j0_{i+1} + ((j1_i - j0_{i+1}).\text{normalized} * \text{bone.length})$$

In which j1 is the joint you want to look for, j0 is the joint that the program calculated in the previous iteration, and the bone refers to the bone that contains both of these joints.

Once all joints in this chain are calculated, the program must go to the second phase of the program in the loop which would be forward inverse kinematics. If you look at the chain created by the first phase, you will see that the first joint could be some distance away from the root, which would be unacceptable. To fix this, the program will essentially do the same exact thing as it did with the first joint except instead of going backwards from the edge of the chain to the root, it will go forward from the root to the edge of the chain. The target in this case would be the original position of the first joint and the initial condition would be to set the first joint of the chain to that original position. Going forward in the bone chain, we do the exact same thing.

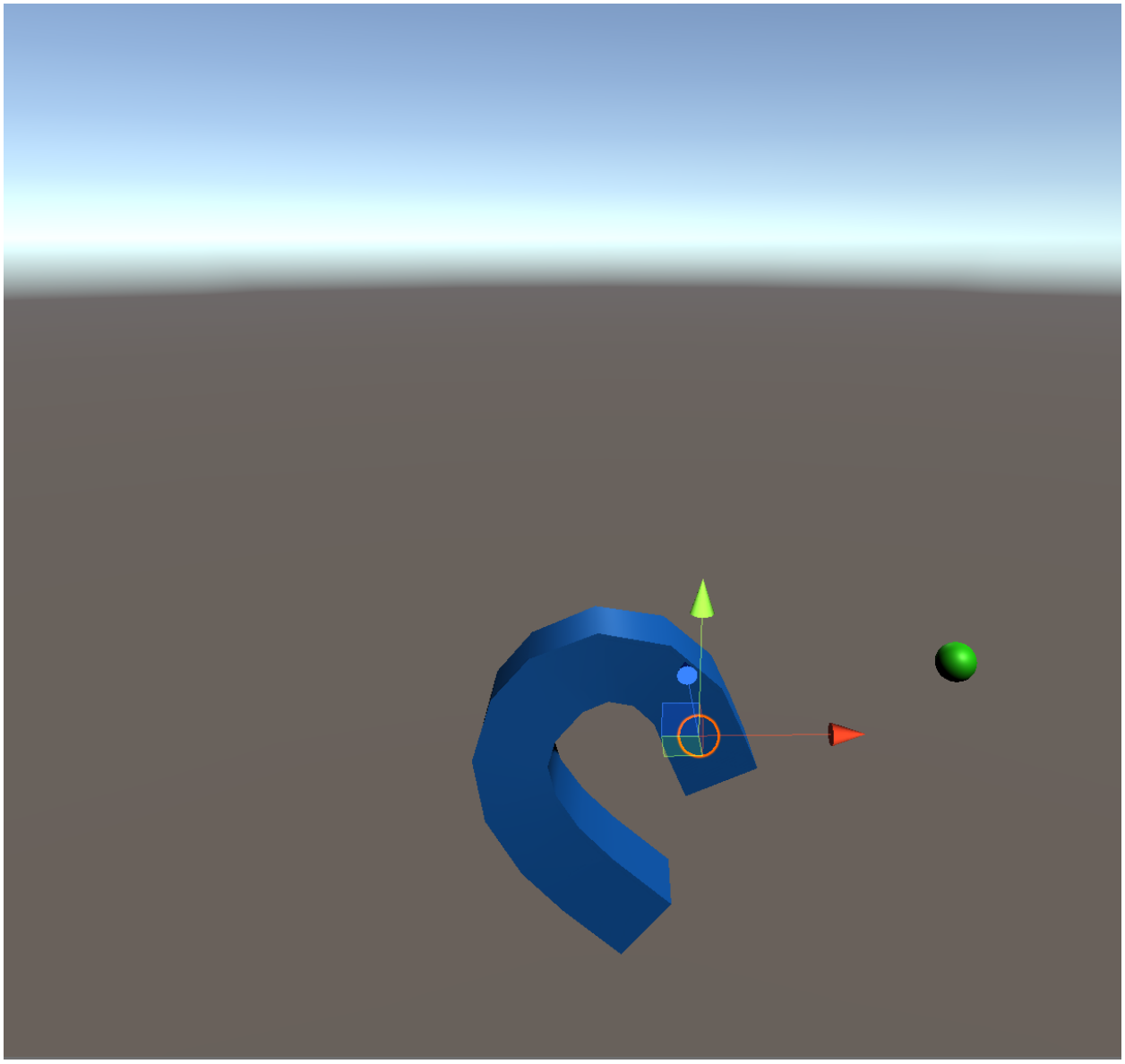
Once the forward inverse kinematic finishes, the program will get a chain whose end is much closer to the target than before. The program will keep running this loop until the following happens: the distance between the last joint and the target location is within a good threshold defined by the user or 10 iterations of the loop have passed within the algorithm. Because of FABRIK's relative efficiency, it would generally only take between 10-15 iterations to reach a suitable solution. The implementation can be found in IK_Manager.cs in the scripts folder.

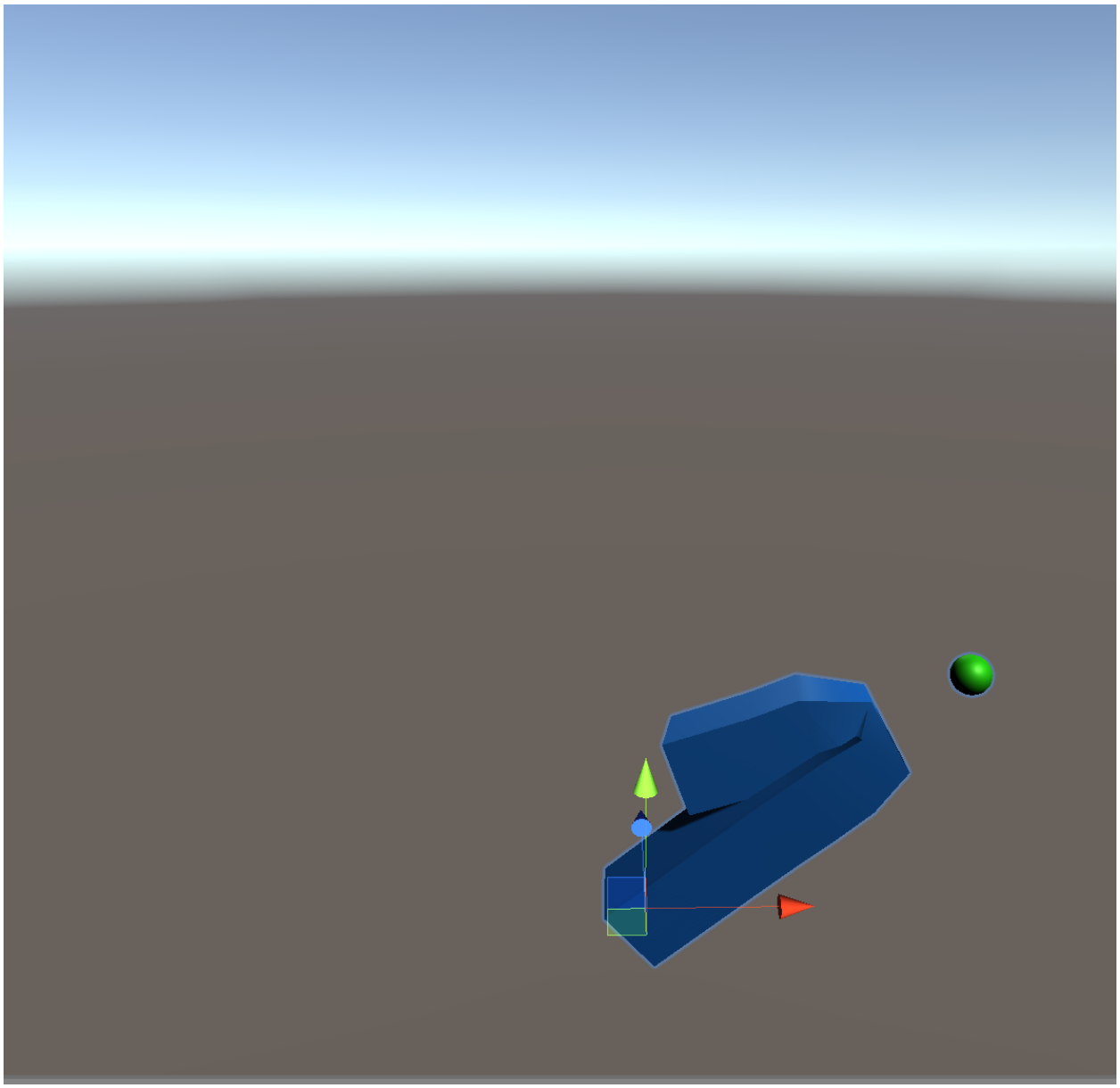
Once an optimal bone chain has been found, the program will use this bone chain to easily update all the bones in the chain, specifically only the rotation of the bones. This can easily be achieved by just running Unity's Quaternion.FromToRotation(zeroTangent, tgtTangent) within Bone.cs in UpdateBone(). The zero tangent is the bone tangent when the rotation is the zero vector when converted to euler angles. The tgt tangent is the bone tangent calculated by the new updated destination joint from the vector and the origin joint. However, this doesn't consider bone roll which was a constant problem within this project.

To consider bone roll, the program will assume that the bone's local roll will stay constant throughout the entire execution runtime. To do this, the program will keep track of the original local normal vector through the localOrigNormal variable and then the program will check the new normal against this localOrigNormal vector. Once we get the angle between these normals, the program can then rotate the bone that angle across the tangent axis to get the right roll. This can be seen in Bone.cs

With the official system established, there are two other variables to consider: the target which is indicated by a red sphere and the pole which is indicated by the green sphere. The target transform indicates the target that will be the target position of the IK bone chain in every updated frame. The pole indicates the direction in which all joints will bend to, if the pole is used at all. This was made to constrain the IK algorithm so that limbs will bend naturally. This can be seen in pictures below, with the first showing an arm that doesn't use the pole and the second

showing an arm using the pole. In both pictures, the red highlighted target end effector is in the same position.



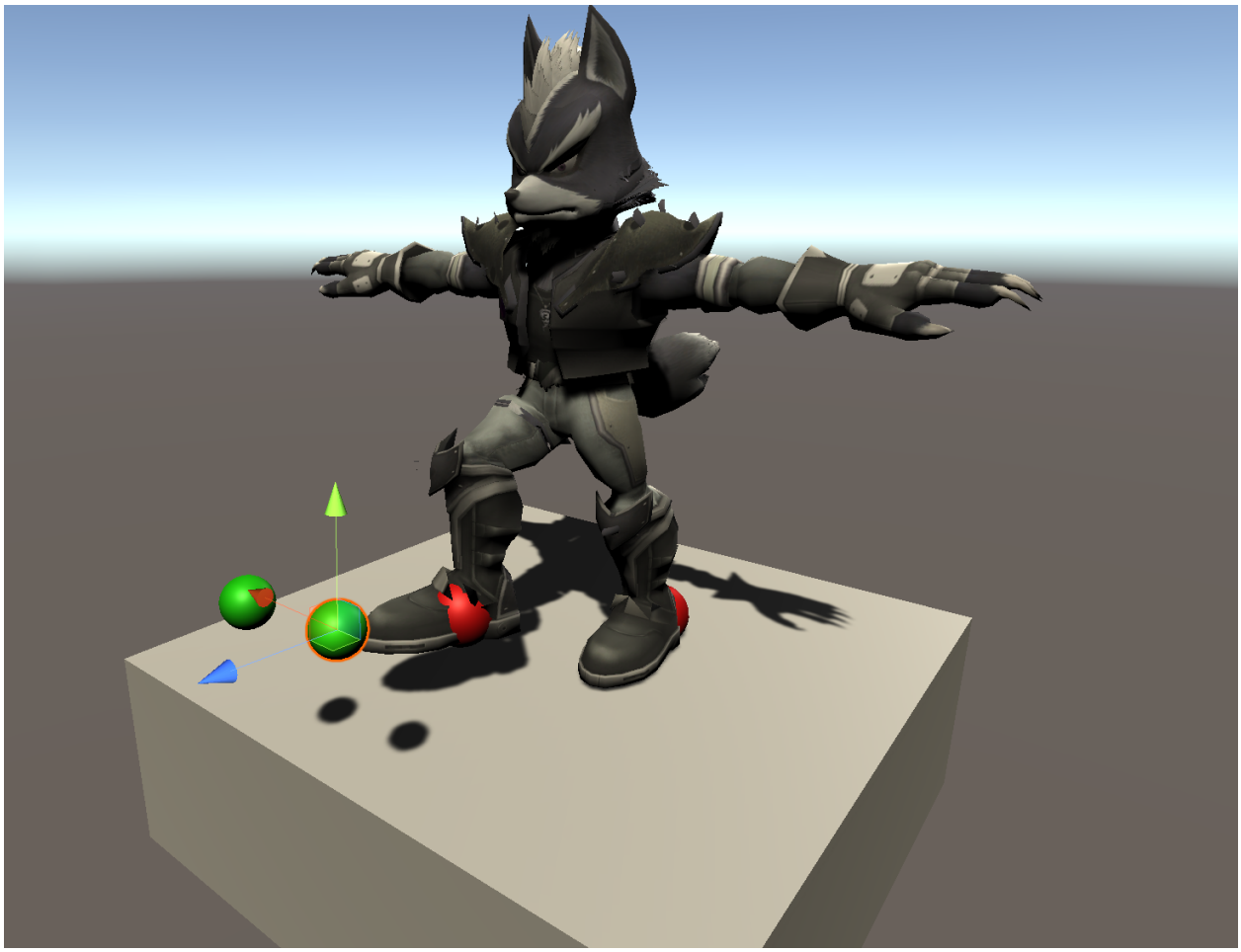


One of the more significant applications of this system is that this red target sphere can easily be moved by the algorithm to control the entire bone chain. It can easily be moved by simple linear interpolations, spherical interpolations or physics. Additionally, the sphere can keep track of collision information, leading to natural procedural animation that reacts to the terrain.

For efficiency purposes, if the target's distance from the root is bigger than the chain itself, the bones will just automatically align in a line from root to target.

FABRIK Application - Procedural Walking

The first and most useful application I wanted to look at was walking, specifically how to create walking that can move in any direction and react to different terrain. The walking itself was generated by creating specific paths for the end targets to go through. The path utilizes 1 LERPs and 2 SLERPs with 1 lerp representing the resting leg resting on the ground and the 2 slerps representing the leg that's in the air. There are 2 poles in front of the legs to ensure that the legs will always bend forward and not backwards. This configuration can be seen below.



The leg will only react to collision when the active leg starts to fall in the second slerp in the air. There are 2 specific cases: whether or not the active leg's end effector target hit something when the path is done.

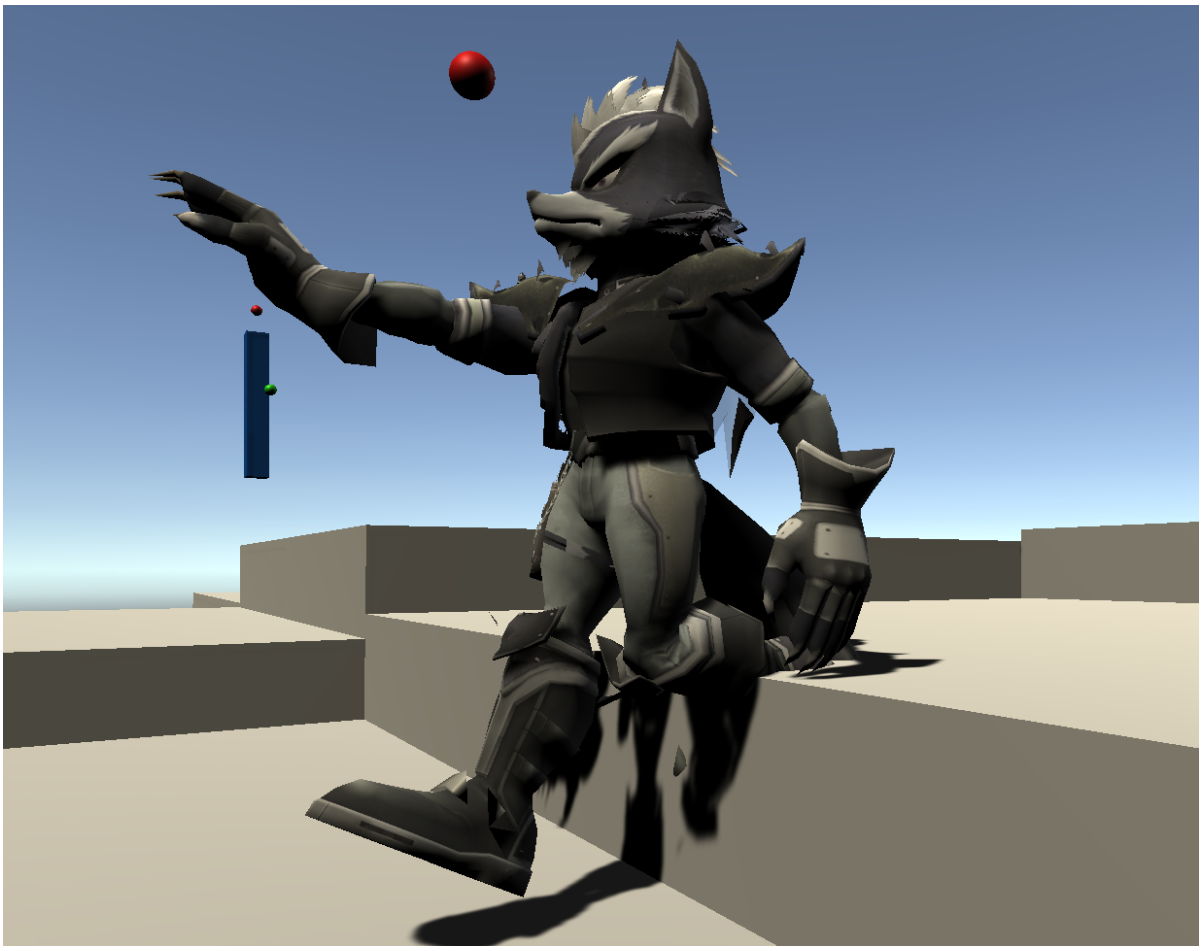
If the leg did hit ground either when or before the path ends, the active leg will switch and if the new resting leg is on ground higher than the previous resting ground, the entire

character will linear interpolate upward to align with that ground for a short time during the animation.

If the leg doesn't hit the ground, the entire body will move down with the resting leg trying to maintain absolute position and the active leg moving with the body. It will keep moving down until the active end effector touches ground. Once it touches ground, the character will switch legs and continue walking.

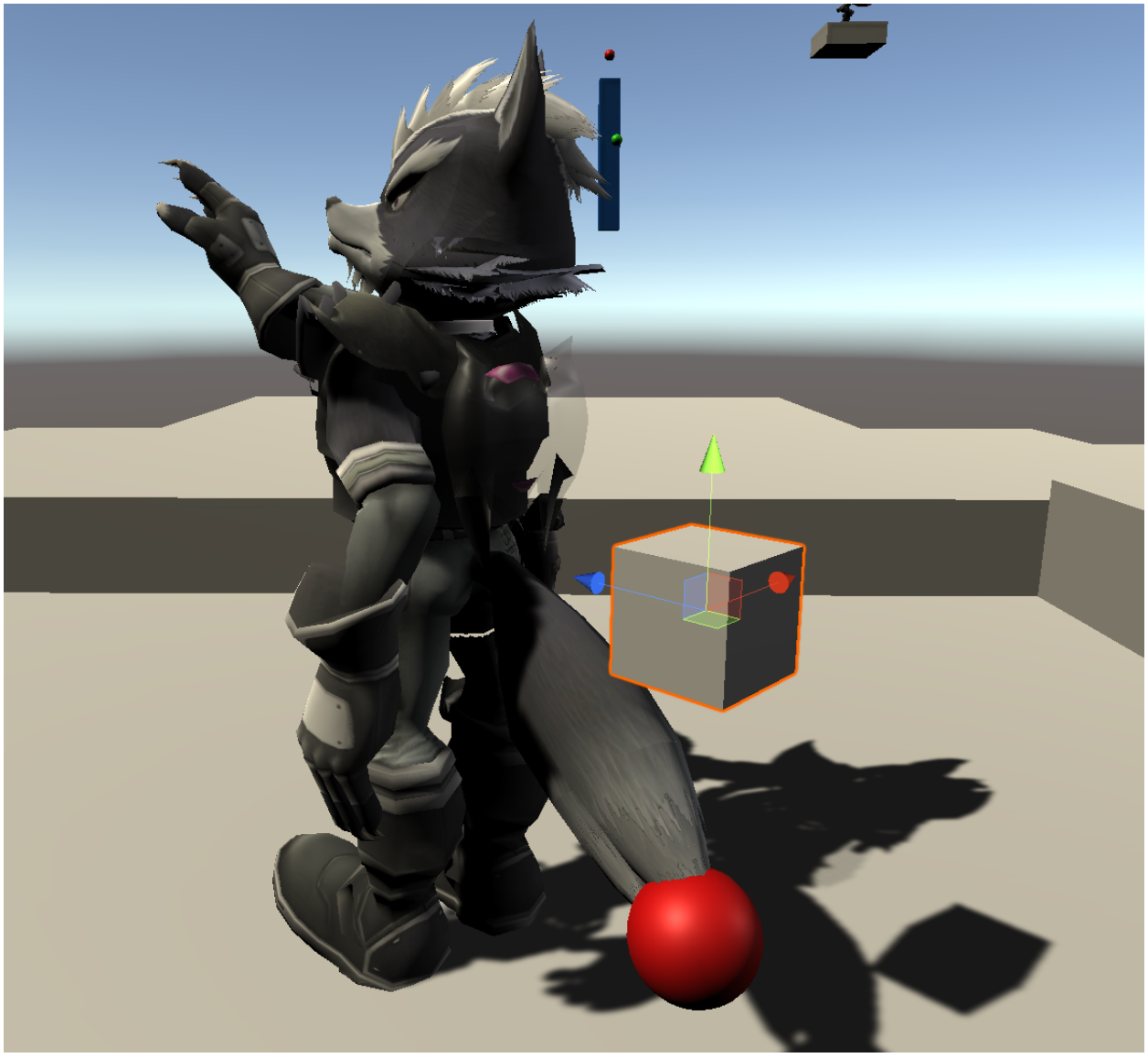
Additionally, walking must consider transitions to idle within a game. This is relatively easy to do because only one leg is active in the air at any given point. All the program has to do is have the leg end effector fall until it hits ground. This system creates pictures such as this where the legs will bend based on the terrain.

To make the walking animation look better, the character's chest will lean towards which direction the character is moving towards. This was easily done by editing the euler angles of the chest bone of the character to align with the direction of movement.



FABRIK Application - Tail Physics

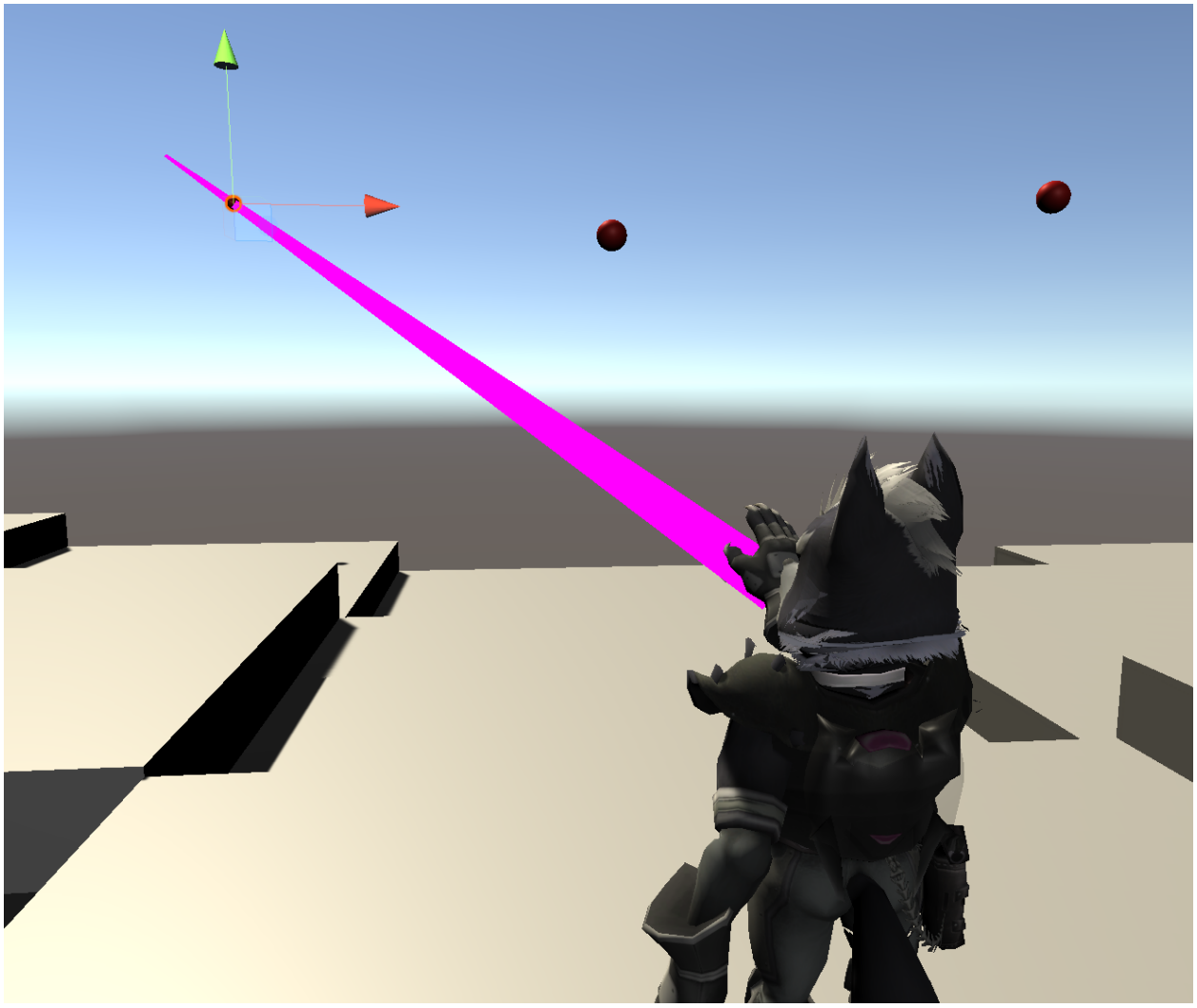
In order to add basic physics to the tail, I simply attached the end effector target for the tailbone chain onto a configuration joint which is a spring joint but with more settings for configuration. The red end effector is separate from the character model which moves with user input, but the white cube spring base is a child of the model, meaning it will move with it. This configuration can be seen below and it is hidden in the game:



This would make sure that when the character moves, the end effector will move as if it's on a spring instead of static constant movement.

FABRIK Application - Aiming Animations

Aiming was easy to implement. Essentially, the program will move the end effector of the aiming based on the mouse's position. It calculates the world position by converting screen space to world space using Unity's base functions to get a ray and get a position some distance along that ray. Once the position is set, the shooting arm will simply point towards that position at all times.

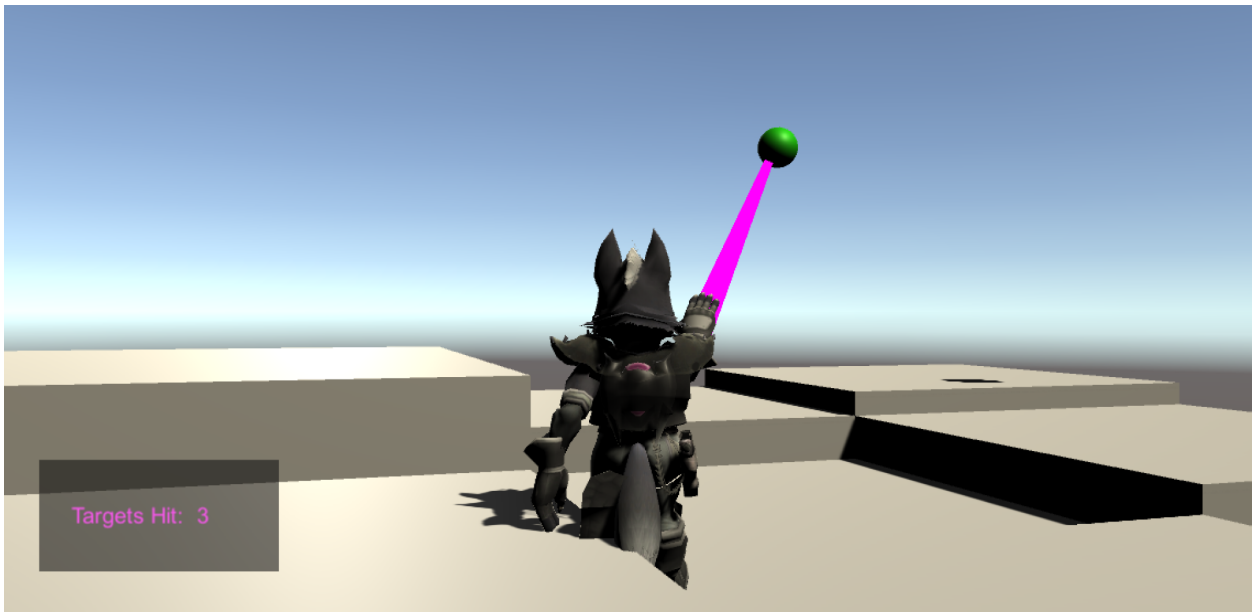


To make this animation look better, I used the LookAt function in Unity on the head so that the character is always looking at the target at all times.

Game Functionality

In addition to implementing the FABRIK inverse kinematic system and applying it in different ways, I added simple game functionality that uses raycasts for collisions and the line renderer to render lasers. The player can move through basic WASD keys, can aim using the mouse, can shoot using left click, and can move the camera by moving the mouse to the sides of the screen. Shooting shoots a raycast which is based off of the "gun" in the character's hands which acts as the src and the target end effector discussed in aiming as the destination point. This would form a ray that will return the first hit object in the ray through Physics.Raycast(). Once the programs get the first hit object, the program will act accordingly.

The line renderer will adjust depending on this first hit object, and if the object is of class target, it will activate the target if the target isn't activated using hitTarget() in target.cs and, if the target was activated, increment the number of points the player has. Pictures like this are the result:



Potential Bugs

- Physic velocity issues where characters will drift uncontrollably (Very rare)
- Characters can phase through the floor or walls with no collision (Rare)
- When pausing in the middle of walk animations, bone roll could potentially be off
- Aiming with the mouse is not precise and can miss.