POLITECHNIKA BIAŁOSTOCKA

WYDZIAŁ INFORMATYKI

PRACA DYPLOMOWA INŻYNIERSKA

TEMAT: SKELETAL ANIMATION USING INVERSE KINEMATICS IN THE UNITY ENGINE

WYKONAWCA: ŁUKASZ BIAŁCZAK

.....

podpis

PROMOTOR: DR INŻ. ADAM BOROWICZ

BIAŁYSTOK 2022 r.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Formulation 2

2 Related Work **3**

2.1 IK Algorithms 3

3 Tools **4**

3.1 MakeHuman 4

3.2 Blender 4

3.3 Unity 6

3.3.1 Importing Animations 6

3.3.2 Animator Controller 6

3.3.3 Animation Rigging Package 8

4 Inverse Kinematics in the Unity Engine **9**

4.1 FABRIK implementation 9

4.2 Spider Movement 11

4.2.1 Project Setup 11

4.2.2 Scripts 12

4.3 Human Animation Sequence 15

4.3.1 Project Setup 15

4.3.2 Scripts 16

5 Experiments **19**

6 Conclusion **20**

1. Introduction

1.1 Motivation

Animation is the technique of displaying different positions of a character or object in rapid succession to create the illusion of movement. It is used in various forms of entertainment, such as movies and video games. In the latter, unlike in the former, the animation sequences are performed in real time and therefore impose additional constraints. Without the freedom to process a single frame for minutes or hours during the rendering of the scene, the animator must compromise on the quality and realism of the sequence in order to optimize for gameplay. One such optimization is the use of skeletal animation in which animation sequences are performed by manipulating a tree-like structure of interconnected bones, represented by transforms, to create the desired motion of the character. Furthermore, the interactive nature of video games makes it impossible for the artist to create predefined animation sequences for every possible situation that may occur in the game. For example, an animation of a hand pressing a button is only valid if the character to which the hand belongs is placed exactly in a predefined position. If the button changes its size or position, the baked animation must be altered, or it simply loses its realism. As a result, predefined animation sequences are often generic and do not allow the character or object to interact naturally with their surroundings. Game developers have come up with many methods to improve the realism of animation in games such as playing cutscenes for critical interactions between a character and the world. However, this paper will focus on the use of procedural animation and, more specifically, the application of inverse kinematics to skeletal animations in video games.

Inverse kinematics is a technique used in fields such as robotics and computer graphics to determine the joint angles of a kinematic chain that will result in a particular part of the chain, usually an end effector, reaching a specified position in 3D space. In computer graphics specifically, the technique is often used to animate the movement of characters and objects such that they interact with their surroundings in a more realistic manner. Inverse kinematics is even used in modeling and animation software such as Blender [3] to speed up the process of limb manipulation when creating a static animation.

There are multiple approaches and algorithms that exist within the inverse kinematics domain, such as analytical methods, gradient descent, and optimization techniques. The choice of approach varies depending on the complexity of the use case, the desired realism of the animation, and system limitations.

1.2 Problem Formulation

The aim of this dissertation is to gain a better understanding of the basic algorithms used in inverse kinematics, discover the built-in functionalities that the Unity engine offers for such implementation. The project implementation will apply these concepts to create pairs of animations which consist of baked and inverse kinematics variants. The use cases will expand the problem by introducing additional constraints which will be required to keep the consistency and realism of the animations. The variations will then be compared through the lens of realism and performance.

The author will begin by discussing the theory of the different approaches and algorithms used to solve the inverse kinematics problem, and the resulting choice of the algorithm to be used in the project implementation. The following sections will explain in depth the implementation of two use cases which demonstrate the purpose of inverse kinematics as a skeletal animation technique. Experiments will then be conducted to compare the inverse kinematics animations with their baked counterparts based on realism and performance. Finally, a summary and conclusion of important points will be presented to the reader.

2. Related Work

Inverse kinematics is applied in various fields such as robotics and computer graphics. Over time, many approaches have surfaced in order to solve the IK problem. There are multiple families of solutions [2] which suit different use cases. Among others, trigonometric solutions can be used to solve certain IK problems, however, these solutions are often limited to solving two bone IK scenarios.

2.1 IK Algorithms

3. Tools

Multiple tools were used in the process of creating the demo application for this paper including the Unity game engine, Blender as a modeling and animation software, and MakeHuman as a model creation tool. This chapter discusses the built-in functionalities which make the mentioned tools an effective choice.

3.1 MakeHuman

MakeHuman is an open source tool for making 3D characters. It provides a convenient way of acquiring a human model which is customizable and can be exported in various formats in order to be used in other software programs. The key factors which make this tool suitable for use in the demo application is the options it provides regarding the complexity of the topology of the model's mesh, and the choice of skeleton rig alongside the fact that the exported model is already rigged and ready to be used in an animation software. One of the rig options is specifically designed to be then used in a game engine setting (Figure 3.1).

3.2 Blender

The tool of choice for modeling and animation used for the demo application is Blender. It is a free and open source tool offering a suite of functionalities including the creation of 3D models, rigging, and animation.

Blender offers the functionality of importing existing models in various formats including the *collada* format [1] which is the default export option in MakeHuman. The models, which are already rigged, can then be animated using the Blender animation pose editor. Models can also be created from scratch. Blender offers a 3D modelling tool to create a desired mesh. A custom rig can also be constructed and attached to the created model. Weights can be painted on the meshes vertices for each bone to define how tightly they are bound, and how much the position of each vertex depends on the given bone position.

Lastly, an animation sequence can be created for an existing mesh and rig using the character animation pose editor. The user can define poses for different points in time by creating key frames on a timeline, and blender interpolates the bone positions in between the

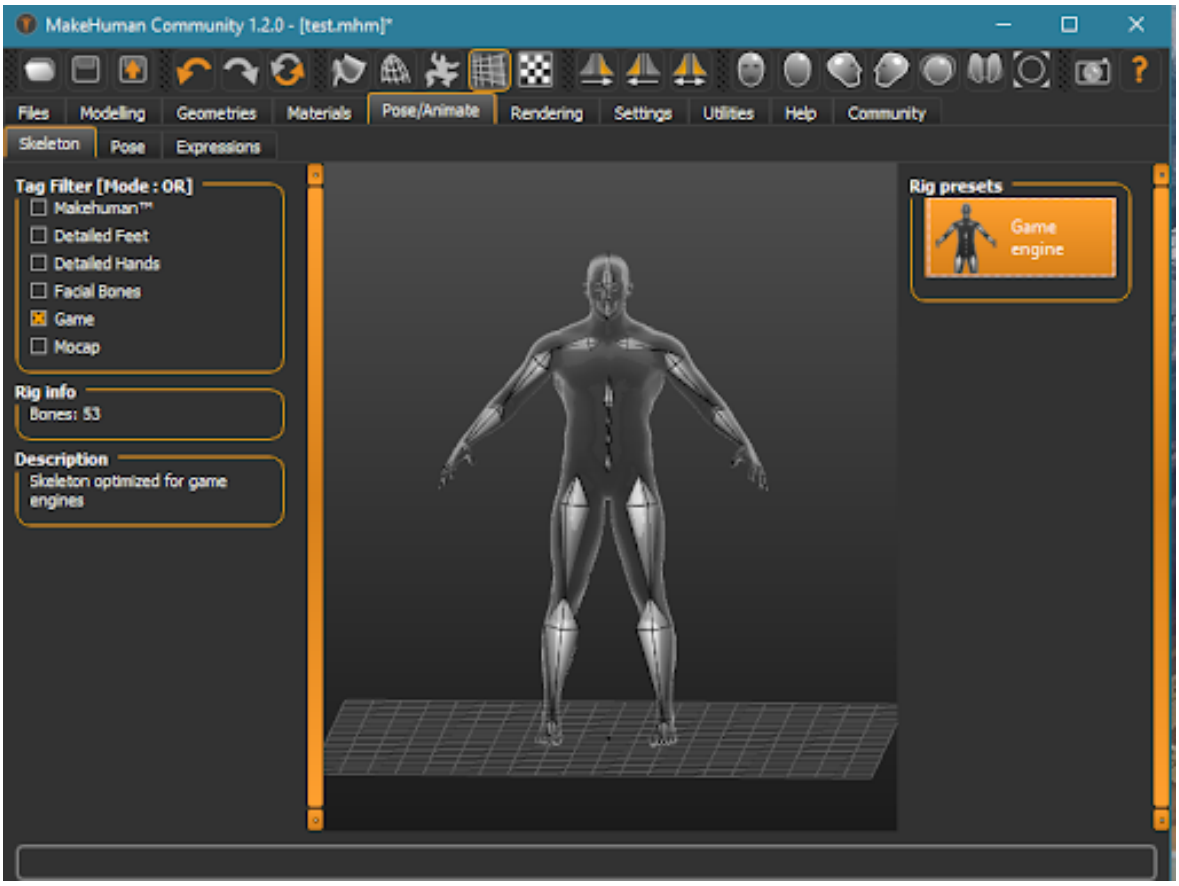

Figure 3.1: MakeHuman rig selection

key frames. This is used to create baked animations for characters and objects, as well as defining animations that are later blended with IK constraints. An animated model can be then exported in the *fbx* format to be used in other software programs. Unity also supports importing a model from a *.blend* file which is the extension of a blender project file.

3.3 Unity

The Unity game engine is the one tool which was non-negotiable as the paper is meant to specifically focus on the usage of inverse kinematics in said game engine. Nevertheless, the engine is a good selection for this use case due to its advanced 3D support, the built-in packages and functionalities which are geared towards the subject of this paper, and the overall popularity of the engine and large community built around it which results in a substantial amount of documentation and support.

3.3.1 Importing Animations

Unity enables users to import animated models from external sources using an *FBX* file or by importing project files from 3D modeling and animation software such as Blender, Autodesk Maya, Cinema4D, or Autodesk 3ds Max [6]. However, the modeling software must be installed on the user's machine in order to import a model from project files, as Unity uses the programs themselves to unpack the file.

When importing an animated model into Unity, the import settings allow the user to break the animation into multiple parts based on start and end times (Figure 3.2). This is convenient, as it allows multiple animations made in Blender to be placed on one timeline one after the other as a single animation, which can then be broken up in Unity. It also allows the user to extract multiple different variations of the same animation, such as importing the full animation as one whole chunk and also breaking it up into separate animations for different use cases.

3.3.2 Animator Controller

An Animator controller allows the user to maintain a set of animation clips and the associated transitions to control the flow between each clip in the form of a state machine

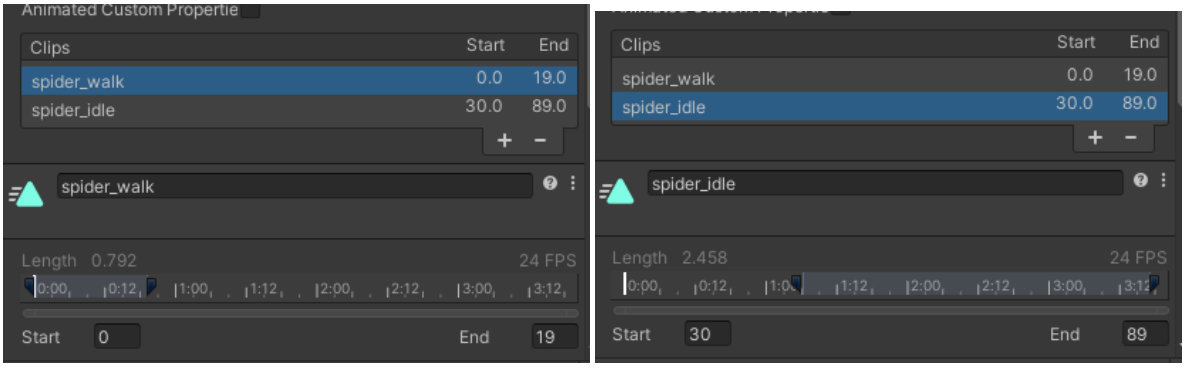

Figure 3.2: Animation clips extracted from a single animation during import

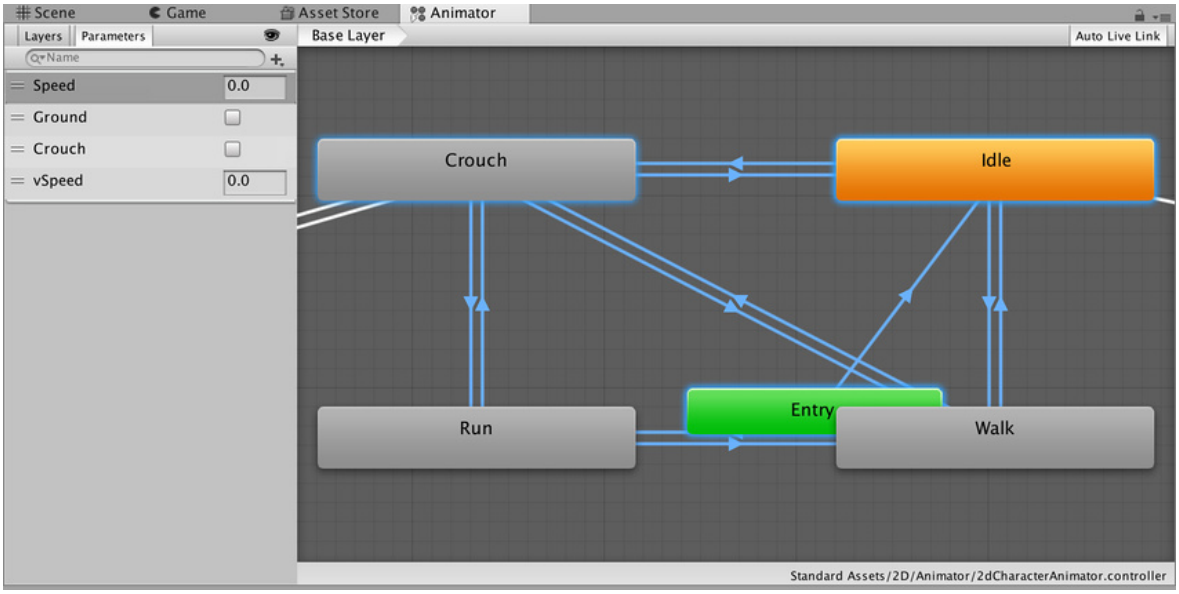

Figure 3.3: Animator controller animation state machine [5]

(Figure 3.3). Animations must be added to an Animator controller in order to be used by a Unity *GameObject* [5]. States can also be controlled procedurally from scripts by accessing an Animator component which must be attached to the Unity *GameObject* and must hold a reference to the given Animator controller.

The Animator controller also has built in IK functionality. This is only available for humanoid models which have a correctly configured avatar [10]. A humanoid avatar is available for models which adhere to a set of defined general guidelines. In essence, a model which falls into the humanoid category must have at least 15 bones which are organized in a way which resembles a human skeleton [9]. Humanoid characters have a few additional functionalities. Namely, because of the criteria required to qualify as a humanoid model, these

Figure 3.4: Predefined rig constraints available in Unity's Animation Rigging package [4]

models are all similar in structure and as such, animation can be mapped from one humanoid model to another [8]. Additionally, these models have built in functionality which allows the developer to procedurally control the models IK weights and positions using the *SetIKPositionWeight, SetIKRotationWeight, SetIKPosition, SetIKRotation, SetLookAtPosition, bodyPosition, bodyRotation* functions [10, 8]. Because these methods aren't available to skeletal structures which do not fit the humanoid description, they are not applicable to the demo application created for this paper.

3.3.3 Animation Rigging Package

The Animation Rigging package is available in the Unity Package Manager, and it provides a much more general approach to the application of inverse kinematics to skeletal animations. After setting up a rig for a model with an Animator component, a mix of predefined constraints can be added to enhance the rig (Figure 3.4). The main constraint of interest for this paper is the *Chain IK Constraint* which implements the FABRIK algorithm. This constraint was useful in creating a proof of concept for the spider before creating the FABRIK script. It also provided a good basis for the public interface that such a script should have to function well in the Unity environment.

4. Inverse Kinematics in the Unity Engine

The demo application written for the purpose of this paper includes two separate use cases of skeletal animation using inverse kinematics in the Unity engine. The first example is that of a four legged spider which uses IK as a means to more naturally adjust its limbs to the terrain it moves around upon. The second example is the application of inverse kinematics to an animation sequence of a human character pressing multiple buttons in succession. The use of inverse kinematics allows the character to adjust its animation to hit all the buttons without the need for a baked animation targeted towards each button, as well as dynamically adjust the order of the buttons to be hit. Although there are two separate use cases demonstrated in this application, both use the same implementation of the FABRIK algorithm.

4.1 FABRIK implementation

This implementation of the FABRIK algorithm is based on the paper written by the author of the Unity engine FABRIK implementation [2]. For the purposes of this application, the basic algorithm is implemented without many additional constraints and limitation on the chain's degrees of freedom. The one constraint added on to this implementation is that of pole targets which will be further explained when discussing the code behind them.

The script which implements the algorithm in this project takes in a few parameters required to set up the mechanism which are shown in Figure 4.1. First and foremost, the root and leaf nodes must be provided in order to define the kinematic chain which is to be manipulated. The next object which the script must have knowledge of is the target transform which the end effector will attempt to move to. The script must also have a tolerance parameter which dictates how close the end effector must be to the target for the position to be considered as solved. All the aforementioned parameters are required for the script to function. Optionally, a pole target object may be passed in if the use case requires it to function in a desired manner.

The first case which the algorithm must cover is if the distance from the root to the target object is greater than the sum of distances between each adjacent bone transform in the defined kinematic chain. In this case, the target is out of reach. Given that the bones in

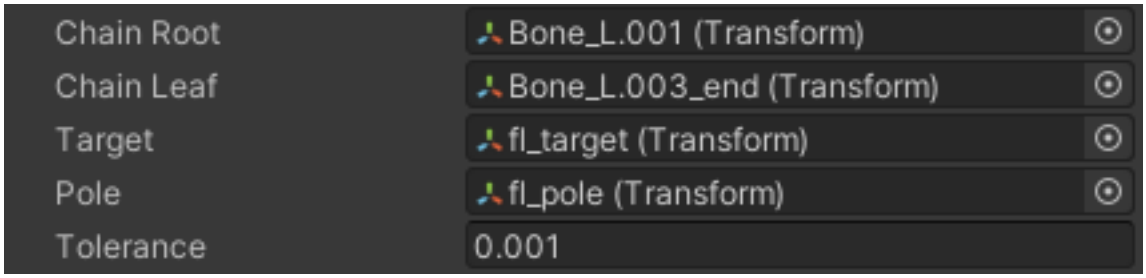

Figure 4.1: FABRIK script parameters

a skeleton are expected to keep a fixed length, the end effector will not be able to reach the target, and instead the kinematic chain straightens and extends in the direction of the target. There is no need to continue with the iterative portion of the algorithm.

A minor optimization in the implementation is the use of square magnitudes when comparing distances to avoid the calculation of square roots, thus reducing the computational costs.

The scenario where the target is within the reach of the kinematic chain utilizes an iterative forward and backward component after which the algorithm is named. Before each iteration, the positions of the joint transforms are copied. All operations and calculations are performed on these copied transforms and at the end of the full pass, the new positions are then copied back to the kinematic chain. It is also important to note that the modification of the transforms is done in Unity's *LateUpdate* function. When using a mix of inverse kinematics and baked animation, the object to which the IK script is attached will have Unity's built-in *Animator* component attached. If the custom IK script were to update joint transforms in the *Update* method, then their attributes may be overwritten by the *Animator* component.

The forward pass of the algorithm iterates through the chain starting from the end effector and ending at the root. At the start, the end effector's position is set to be equal to the position of the target. A straight line can then be imagined to exist between the end effector and the following node. This neighbor's position is then interpolated along the line so that the original distance between the two nodes is kept the same. The same operation is performed for each pair of neighboring nodes throughout the pass.

When the forward pass is complete, the root node is displaced from its original position. This is undesired, as the root's node position should not be affected by the algorithm.

To remedy this, the next step is to repeat the forwards pass, but this time in reverse. The root node's position is set equal to what it was at the beginning of the frame. The next node is then interpolated between its current position and the root to keep the initial bone length. As with the forward pass, this is repeated for each subsequent pair of nodes.

These two steps are repeated together until the end effector is within a threshold distance of the target. The FABRIK algorithm is a heuristic algorithm, and as such it does not lead to an exact result. Instead, it aims to approximate the correct solution and solves the problem in a less complex and more optimized way. Again, the square distances are used to avoid the calculation of square roots.

POLE TARGETS. FIND A SOURCE WHICH EXPLAINS THE METHOD USED IN MY PROGRAM. IF NOT THEN REFER TO THE YOUTUBE VIDEO LOLOL

4.2 Spider Movement

The first use case for inverse kinematics in the demo application is that of a four legged spider. The algorithm is used to adjust the creature's limbs to uneven terrain, leading to a much more natural and realistic movement. The IK version of the spider does not have an Animator component, and the whole of the animation and movement of the spider and its legs is done procedurally.

4.2.1 Project Setup

Each one of the spiders legs is treated as a separate kinematic chain. The spider prefab consists of a container which holds the spider object itself, set of empty objects to which the four IK scripts are attached for the purpose of easy yet separated access. The prefab also contains sets of ray casts and targets. The ray casts serve to scan the surface of the terrain under the spider, and mark the targets to which each leg should move.

Ray casts are dispatched from above the spiders legs, and aim in the creatures local negative Y axis. This ensures that no matter what orientation the spider finds itself in, the rays are always pointing at the surface which it is standing on. Masks are applied to the rays, making sure that only terrain objects are taken into account, while the creature's body itself is not. The ray cast hit point positions are then applied to each leg's respective target object.

These targets serve as markers for the limbs end effectors.

4.2.2 Scripts

With the project set up in this manner, scripts must now be added on to make the scene functional. A script is required for the main movement of the spider, the ray cast logic, and the mechanism which controls the ik targets.

Ray casts

The ray cast objects contain a script component which dispatches the rays and sets the appropriate target positions. First, a mask must be established, which will then be passed into the ray cast operation. This is required so that only terrain is counted as a valid hit. The lack of such a mask may result in unexpected behavior, such as targets being set on the spider's body itself. The ray cast object is then created, shooting in the local negative Y axis direction. This ensures that no matter the orientation of the creature, the rays are sent towards the surface that the spider is standing on. The targets controlling the spider's legs will not be updating their positions to the ray cast hit points each frame, though their scripts must have knowledge of the hit positions at any given time. Given this, a separate set of objects are set to track the ray cast hit points each frame.

Target Logic

In order for the spider's movement to seem realistic, the targets controlling each leg must adhere to a set of rules pertaining to their movement. As mentioned in the ray cast section, the IK targets cannot simply be set to track the ray cast hit points. The following is an outline of the rules specified for the IK targets, which define if it should start moving towards its ray cast hit target:

- A target must be grounded to be eligible for a movement sequence.
- A target will only begin moving towards the ray cast hit point if the distance between them is above a specified threshold.
- A target is only allowed to begin moving towards the ray cast hit point if both legs on

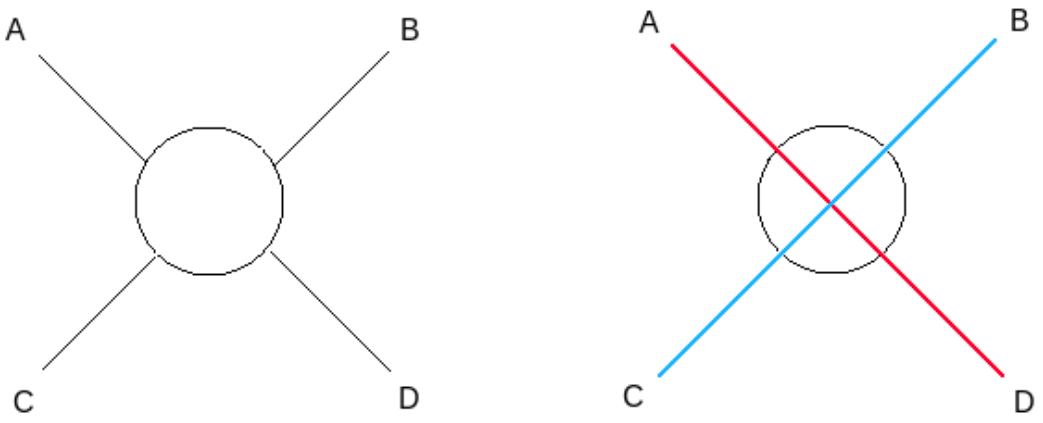

Figure 4.2: The diagonals referencing the spiders legs which are used when checking if a target is allowed to begin moving, where A and D are on an opposite diagonal to B and C

the opposite diagonal are grounded (See Figure 4.2).

When a target satisfies all of these conditions, it makes note of the ray cast hit's current position, which it will use for its upcoming movement sequence. Once it begins moving, the *Grounded* boolean is set to be false, so that the other targets know whether they can begin moving or not.

The movement sequence itself is done using Unity's *Vector3.MoveTowards* method, which takes in a current position vector, a vector to move to, and the maximum distance to move per frame, which can be used to control the movement speed. This method allows the target to interpolate its position every frame. The values fed into this method are simply the target's current position, the ray cast hit target's position, which was recorded right before the beginning of the movement sequence, and an arbitrary speed value, which is dependent on *Time.deltaTime* to avoid variations when the frame rate changes. The only caveat is that in the first half of the movement, the destination vector's height component is increased to achieve an arc-like movement. This produces the effect of lifting the spider's leg.

General Movement

The main movement script for the IK spider implementation brings the whole system together. It has three main objectives:

- Calculate the rotation of the spider so that it's local up and forward vectors can be set accordingly.
- React to input by moving the main body of the spider along with the ray casts.
- Regulate the height of the spiders body above the ground.

The first objective - calculating the rotation of the spider - is what allows it to scale walls and walk upside down on the ceiling. The rotation is determined based on the limb positions at the beginning of each *Update* call. First, two vectors are constructed from the end effectors of both sets of diagonally opposed legs. The local up vector of the spider is then calculated by taking the cross product of these two vectors. This determines the orientation of the spider's main body, and it also affects the direction that the four legs' rays are cast. The local forward vector is then obtained from the cross product between the newfound up vector and the spider's right vector. This new vector is what will be used to determine the direction of movement when the spider receives input from the user.

The second objective - reacting according to input - is quite simple once the local directional axes are determined. When the script detects a non-zero value on either the *Horizontal* or *Vertical* axis, it reacts accordingly by moving the spider along with all the ray cast objects. The *Vertical* axis corresponds to the forward and backward movement of the spider. The script reacts by simply moving the spider's main body along the local forward axis mentioned previously. Additionally, the ray casting objects are offset either forwards or backward depending on the direction of the movement. This is done because the ray casts must be slightly ahead of the default leg positions so that the legs end up moving in a natural manner. The *Horizontal* axis is responsible for rotating the spider about its local up axis which allows the spider to turn around.

Finally, the spider's height off of the surface must be regulated each frame. The lack of a gravitational force acting on the body to keep it flush with the ground, and the unconstrained rotational capability of the creature, means that the distance between the spider and the surface it is walking on must be procedurally kept in check. This is done with yet another ray cast which originates from the center of the spider's body and points in the negative up axis direction. If the distance to the hit point exceeds an acceptable range, the body is

moved towards said range, again using the *Vector3.MoveTowards* method to linearly interpolate the spiders position and avoid excessive jerkiness which occurs with a frequent variation in height.

4.3 Human Animation Sequence

The second example created to demonstrate the use of inverse kinematics for skeletal animation in Unity is that of a human animation sequence which consists of pressing multiple buttons in succession. While this may seem like a simple animation to create in a tool like Blender, such an animation lacks the adaptability needed for a game which has many differing button pressing scenarios. The integration of IK into this kind of animation sequence adds the ability to adjust to multiple amounts of buttons on a panel, different configurations of button positions, various sequences of buttons to press, and keep a consistent hand placement on the buttons without the need to force the character to stand in one defined position. The example was not made solely through the use of IK, and instead it uses a mix of baked animations for the part of the animation which doesn't require variation, and IK for the part of the animation which is subject to change.

4.3.1 Project Setup

A human model is exported from the MakeHuman software, imported into Blender for animation, and then again exported and imported into the Unity engine. The scene is set up with a group of buttons which are positioned on a wall. The buttons themselves each have an *EmptyObject* which will provide the transform needed in order to aim the hand at the given button. The human character is posted in front of this set up. An Animator component is attached to the human model in order to make use of certain animations which were extracted from the full button press animation created in Blender when importing the model into Unity. Namely, the portions of the animation which consist of raising the hand to the default pressing position and lowering it back down, are useful in the IK version of this animation sequence. This is because these two animations are not dependent on the amount of buttons on the wall, nor the position of the character relative to the buttons.

4.3.2 Scripts

The whole logic around this animation sequence is done using one script. It must take in a few public objects as parameters in order to work correctly (Figure ref). Firstly, the hand's IK target must be passed in to enable the procedural control of the model's position. The script must also have a reference to the *Rig* object to which the FABRIK script component is attached to. This is needed in order to enable and disable the IK depending on the phase of the animation sequence. When raising and lowering the hand, the baked animations are used and the IK should be disabled. The next parameter is a list of transforms for each button to which the IK target will move to in order to achieve the effect of pressing a button. A couple of private variables are also declared to make the script functional. These variables hold information such as references to the animator and FABRIK script components, an idle variable which prevents the character from starting a new animation sequence before finishing one that was previously initiated, a value which determines the speed of the hand's movement, and a list of integers which decide the sequence of buttons to be pressed. One last variable holds the *origin* position which is set to the hand's position at the moment between the raising of the hand and the press of a button. This transform is important because it is required in order for the IK target to know where to return to after it is done pressing a given button.

The main functionality of the script revolves around Unity's coroutines. Coroutines enable a method to pause it's execution and then continue where it left off on the next frame [7]. This allows the method to spread a task over multiple frames. It is useful when chaining together a series of events which must wait for a previous event to finish its task before beginning its own. The fundamental part of a coroutine is its *yield* statement which is what pauses the methods execution. A yield may be used to wait for a certain amount of time, to wait for a certain boolean expression to evaluate as true, or it can initiate a nested coroutine and wait for it to end before continuing its own execution. The yield can also simply return a null value which pauses the execution of the method until the next frame, which can be useful to spread out a loop to execute each iteration in a separate frame. In this human animation sequence demo, nested coroutines are used to enhance readability and maintainability, allowing for easy reconfiguration of the main loop.

The first coroutine which is used as a basic building block is the *HandAnimation* coroutine which takes in an animation name as a string, executes the animation, and allows the calling function to continue its execution only after the animation is done. This is possible by checking the animators state info for the *normalizedTime* attribute. This floating point value starts at 0 and grows as the animation plays through, with the end of the animation mapping to a value of 1. The coroutine can therefore yield a boolean expression checking if the *normalizedTime* value is equal or greater than one.

The second building block is the *MoveToTarget* coroutine. This method takes in a destination vector and a duration. Its purpose is to move the IK target to a provided destination over a given amount of time. This is implemented by keeping track of the time elapsed since the beginning of the coroutine, and then linearly interpolating (lerping) the position of the target in a while loop which checks to see if the time elapsed hasn't exceeded the desired duration of the movement. A *yield return null* in the while loop ensures that each of its iterations are separated in a way where only one iteration is executed per frame. A set of two of these coroutines are combined in the *PressButton* coroutine which combines the movement of the hand from its origin position to the button and the movement back to the origin position.

When the script receives a certain input from the player, and an animation is not already being played - which is monitored through the *idle* boolean, the *PressButtons* coroutine begins execution. This coroutine acts as the main event loop for the entire animation sequence. The *idle* boolean is then set to false in order to block the initiation of a subsequent animation sequence before the current one is finished. The *HandAnimation* coroutine is executed to play the baked animation of the hand, raising it to its origin position. The origin position is then saved in the *origin* variable which will later be used during the IK movements. Now that the baked animation part of the animation sequence is complete, the IK component of the rig must be enabled to use its functionality, and have the hand track the IK target. The *PressButton* coroutine is then executed multiple times in a loop for every value in the *buttonSequence* array. This can ideally be integrated with player input in a real game scenario, where the *PressButton* coroutine is played in reaction to sequence entered by the player. After the pressing sequence is complete, the IK component must be disabled before playing the animation responsible for lowering the character's hand back to its default position. Lastly,

the *idle* boolean can be set back to true as the current animation sequence is complete the initiating of subsequent animations should not be blocked.

5. Experiments

6. Conclusion

Bibliography

[1] Adobe. Collada file type. https://www.adobe.com/creativecloud/file-types/image/vector/collada-file.html, 2022. Accessed: 2022-12-27.

[2] Andreas Aristidou and Joan Lasenby. Fabrik: A fast, iterative solver for the inverse kinematics problem. *Graphical Models*, 73:243–260, 9 2011.

[3] Blender 3.4 Manual. Inverse kinematics introduction. https://docs.blender.org/manual/en/latest/animation/armatures/posing/bone_constraints/inverse_kinematics/introduction.html, 2022. Accessed: 2022-12-26.

[4] Unity User Manual. Animation rigging - constraint components. https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/ConstraintComponents.html, 2020. Accessed: 2022-12-28.

[5] Unity User Manual. Animation controller. https://docs.unity3d.com/Manual/class-AnimatorController.html, 2022. Accessed: 2022-12-28.

[6] Unity User Manual. Animation from external sources. https://docs.unity3d.com/Manual/AnimationsImport.html, 2022. Accessed: 2022-12-28.

[7] Unity User Manual. Coroutines. https://docs.unity3d.com/Manual/Coroutines.html, 2022. Accessed: 2022-12-29.

[8] Unity User Manual. Humanoid avatars. https://docs.unity3d.com/Manual/AvatarCreationandSetup.html, 2022. Accessed: 2022-12-28.

[9] Unity User Manual. Importing a model with humanoid animations. https://docs.unity3d.com/Manual/ConfiguringtheAvatar.html, 2022. Accessed: 2022-12-28.

[10] Unity User Manual. Inverse kinematics. https://docs.unity3d.com/Manual/InverseKinematics.html, 2022. Accessed: 2022-12-28.