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 2023 r.

Contents

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, 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, so called bake animations, 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 (IK) 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, 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 [10] to speed up the process of limb manipulation when creating a static animation.

There are multiple approaches that exist within the inverse kinematics domain, such as analytical methods 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 work is to gain a better understanding of the basic algorithms used in inverse kinematics and to 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 methods 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, conclusion will be presented.

2. Related Work and Theory

Inverse kinematics is applied in various fields, mainly robotics and computer graphics. However, it is applied to a few unique problems, such as the prediction of protein structure [6]. It has also found its uses in rehabilitation medicine due to its biomechanical modelling ability. 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 analytically [20], however, these solutions are often limited to solving two bone IK scenarios. More recently neural networks have also been used as a tool to solve IK problems [9]. This paper will go over the iterative and heuristic approaches which provide less complex solutions when increasing the kinematic chain length and degrees of freedom.

2.1 Kinematics

Kinematics is a branch of classical mechanics concerned with the geometrically possible motion of a body or system of bodies without consideration of the forces involved [5]. A kinematic chain is a tree like hierarchical structure of joint transforms which are connected to each other. Because the relationship between joints is hierarchical, a transformation which is applied to a given joint affects all of its descendant nodes. When manipulating a kinematic chain, transformations in the form of rotations and translations are applied to joint transforms in order to achieve a desired position of one or multiple transforms called the end effectors. The problem of kinematics can be subdivided into two approaches which each present their own problem and solution.

- The problem of *Forward Kinematics* which has to do with the identification of the final position of an end effector as a result of a set of transformations being applied to the kinematic chain to which the end effector belongs.
- The problem of *Inverse Kinematics* which pertains to the search for a configuration of joint transformations for a kinematic chain which allow the end effector to reach a predefined target position.

The forward kinematics (FK) problem can be said to have a guaranteed solution as long as the set of joint transformations used as an input are known. To solve the FK problem, the transformations are applied to the kinematic chain, and the transform of the end effector is taken as an output. On the contrary, when dealing with IK, the given problem may have no solutions, one solution, or many solutions.

No Solutions

An IK problem with no solutions can occur for several reasons. Most notably, and IK solution is impossible when the target is unreachable through the limitations of chain length. When the distance between the chain root and the target is larger than the sum of distances between each adjacent chain link (Fig. 2.1), it is impossible to achieve a solution, as the root of the chain is static and cannot be moved in order to allow the end effector to reach its target. Similarly, and unreachable case exists if the first bone segment is longer than the sum of the remaining bone segments (Fig. 2.2).

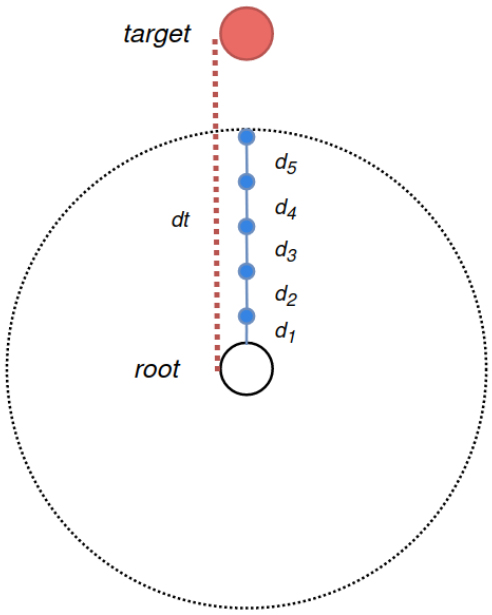


Figure 2.1: An IK problem with no solution where the target is unreachable because the sum of segment distances is less than the distance between the root and the target $\sum_{i=1}^{n} d_i < dt$

Another reason for the lack of solutions to an IK problem can be its over-constraining. Constraints are used to dictate the range of motion of each joint. For example, they might

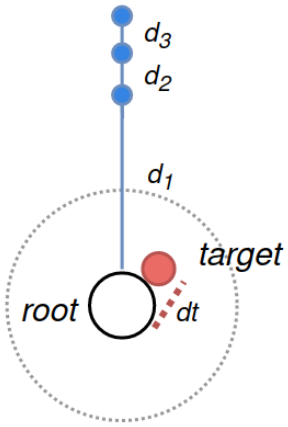


Figure 2.2: An IK problem with no solution where the target is unreachable because the length of the first segment is greater than the summed length of the rest of the kinematic chain $d_1 - \sum_{i=2}^{n} d_i > dt$. This creates a radius around the root where if the root's distance to the target is smaller than the radius, the target is unreachable.

limit the joint's degrees of freedom by reducing the dimensions in which it can perform its rotational and translational transformations. They are often modelled after joints which are present in the human body, such as hinge joints, ball joints, or pivot joints. A constraint can also limit the extent of a joint's ability to bend which is defined by the allowed angle of the joint's rotation in relation to the rotation of its parent node. If too many of such constraints are added to a kinematic chain, it may have blind spots for which it is unable to find a configuration of transformations which can bend the chain to reach the target (Fig. 2.3).

If a kinematic chain has more than one joint which is classified as an end effector with its own separate target, a problem with no solutions can be defined in a way that prevents both end effectors from reaching their target. If at least one end effector is unable to reach its target then the problem can be said to be unsolvable.

One or many solutions

When an IK problem is not limited by the cases mentioned above, it can have one, or many solutions depending on the case and constraints. More often than not the problem will have multiple solutions, and if the kinematic chain is unconstrained, the solution set for and

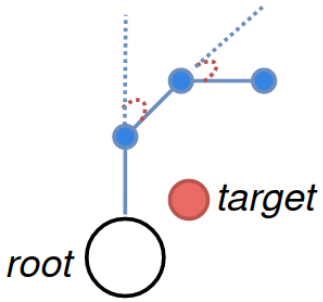


Figure 2.3: An IK problem with no solution where the rotational constraints placed on the joints prevent the end effector from being able to bend enough to reach the target

IK problem grows very large.

While the introduction of constraints can decrease the number of solutions, the simplest case to consider for an unconstrained kinematic chain is one where the chain must stretch to its full extent in order to reach the target. The sum of the chain's segment lengths d_i is equal to the distance from the root to the target dt:

$$\sum_{i=1}^{n} d_i = dt \tag{1}$$

2.2 Underlying principals

In order to describe the configuration of a given kinematic chain, a set of scalars $\theta_1, \ldots, \theta_n$ are defined for each of the chain's n joints transforms. A set of k joints $s_1, \ldots, s_k$ is a subset of the kinematic chain called the end effectors, and these end effector's positions are a function of the chain's configuration.

$$s_i = f_i(\theta) \tag{2}$$

The above equation presents the FK problem where the position of an end effector is calculated based on the given configuration of the chain. To define the IK problem, a configuration of scalars must be found based on the given final positions of the end effectors.

$$\theta = f^{-1}(s) \tag{3}$$

The problem lies in inverting f, as it is non-linear. Taking this into account, along with the fact that an IK problem may have no solutions, one solution, or many solutions, an

approximation approach can be taken to solve the problem efficiently. This can be achieved with iterative and heuristic algorithms.

2.3 IK Algorithms

2.3.1 Jacobian Inverse

This approach to the IK problem [4, 21, 3] offers a linear approximation through an iterative calculation which estimates a change to the given configuration of joint scalars necessary to reduce the distance between an end effector and its desired destination. This is done through the use of a matrix J called the Jacobian matrix which contains partial derivatives of the entire kinematic chain relative to the set of end effectors. J is a function of the chain configuration's joint scalars: θ:

$$J(\theta)_{ij} = \left(\frac{\partial s_i}{\partial \theta_j} \right)_{ij} \tag{4}$$

where n and k are the previously defined number of joints and number of end effectors in the chain respectively. Given the position p_j of the jth joint and its current rotational axis v_j each entry of the Jacobian matrix can be defined as

$$\frac{\partial s_i}{\partial \theta_j} = \mathrm{v}_j \times (s_i - \mathrm{p}_j) \tag{5}$$

Using the configuration of joint values θ, a column vector of end effector positions $\vec{s}$ and a column vector of target positions $\vec{t}$, the Jacobian can be computed as $J = J(\theta)$. With prime notation denoting a first derivative with respect to time, the forward dynamics equation can be presented as

$$\vec{s}' = J(\theta)\theta' \tag{6}$$

A change to the kinematic chain's configuration $\Delta\theta$ is then needed which when applied, updates the end effectors to minimize their distance to the targets. The resulting change in the end effector positions $\Delta\vec{s}$ due to the modification of the configuration is then estimated as

$$\Delta\vec{s} \approx J\Delta\theta \tag{7}$$

and the changes made to the joint scalars should be done in a way where $\vec{s}$ is as close to $\vec{e}$ which is a vector of desired changes in position for each of the chain's end effectors,

defined as $e_i = \mathrm{t}_i - s_i$. Due to this relation between $\vec{s}$ and $\vec{e}$, the FK equation can be rewritten:

$$\vec{e} = \Delta J(\theta) \tag{8}$$

and thus the IK problem for the Jacobian method is as follows:

$$\Delta\theta = J^{-1}\vec{e} \tag{9}$$

A solution to this Jacobian method of the IK problem is not always simple due to the possibility of a non-invertible matrix, or singularities which prevent changes of the chain's configurations which achieve the sought-after result in end effector positions. Certain methods have been brought forward to surmount these challenges.

Pseudo-Inverse

To address the problem of non-invertible Jacobian matrices, the Jacobian Pseudo-inverse $J^{\dagger}$, also called the *Moore-Penrose* inverse can be used as a substitute:

$$\Delta\theta = J^{\dagger}\vec{e} \tag{10}$$

The benefit of using the pseudo-inverse method is that the matrix $J^{\dagger}$ is defined for every case of the Jacobian matrix, whether it is invertible or not. The pseudo-inverse matrix is defined as

$$J^{\dagger} = J^T(JJ^T)^{-1} \tag{11}$$

Where the pseudo-inverse falls short is when approaching singularities. In these cases, the resulting values become unstable leading to oscillations.

Jacobian Transpose

Another way to overcome the case where J is a non-invertible matrix is to use the matrix transpose instead. An so, the inverse is substituted with the transpose, with a coefficient α:

$$\Delta\theta = \alpha J^T\vec{e}. \tag{12}$$

In order to minimize the distance of each $\vec{e}$ between the end effectors and their targets, the coefficient α is calculated with the goal of achieving the smallest value of $\vec{e}$ after each iteration. The equation which models this calculation is as follows

$$\alpha = \frac{\langle \vec{e}, JJ^T\vec{e} \rangle}{\langle JJ^T\vec{e}, JJ^T\vec{e} \rangle} \tag{13}$$

where the angular brackets represent a dot product operation between the two values contained inside them. The value of the coefficient α must be small enough to avoid oscillations, and due to this the process of converging end effectors on the targets is slow and requires many iterations to achieve a desired configuration.

Other Jacobian methods

A multitude of approaches to the IK problem using Jacobian methods have been suggested, some of which are variations or combinations of the presented methods. Among others, the Single Value Decomposition method extends the pseudo-inverse method with orthogonal matrices, the Damped Least Squares method helps avoid problems with singularities present in the pseudo-inverse method, the Pseudo-inverse Damped Lease Squares combines the two previously mentioned methods to improve their instability when nearing singularities, and the Selectively Damped Lease Squares method builds on the previous method to converge in fewer iterations and optimize for multiple end effectors.

2.3.2 Newton Methods

The Newton methods are based on a second order Taylor series expansion of the function $f(x)$ with $H_f(x)$ being the Hessian matrix:

$$f(x+\sigma) \approx f(x) + |\nabla f(x)|^T\sigma + \frac{1}{2}\sigma^T H_f(x)\sigma \tag{14}$$

The Newton methods have the advantage over the Jacobian inverse methods in the smoothness of the movement and the lack of singularity problems which allows the iterative approach to avoid oscillations. However, the complexity of the Hessian matrix drastically raises the computational costs for each iteration. A few methods have been brought forward which aim to approximate the Hessian matrix to reduce the complexity of the problem, however the computational costs remain relatively high.

2.3.3 Heuristic IK algorithms

Heuristic algorithms provide an approach which avoid the need for complex equations and heavy computations, and instead act as techniques to achieve an approximate solution iteratively in a much more simple way. These methods are very popular in the domain of computer graphics and animation, due to the problems not needing extremely accurate solutions but benefiting from a simple and fast approach in the real-time environment of video games.

Cyclic Coordinate Descent

The Cyclic Coordinate Descent (CCD) algorithm [8] is one of the simplest and most used IK solutions in the computer games industry. It also finds applications in robotics as well as the already mentioned protein structure prediction [6].

The algorithm acts on each joint transform separately, minimizing the distance between the end effector and the target by finding the appropriate angle by which it should rotate the given joint. The iteration begins at the end of the chain, excluding the end effector. The desired angle of rotation is calculated by finding lines, one of which passes through the current joint and the end effector, while the other passes through the current joint and the target. The angle between these two lines is the angle by which the current joint should be rotated in order to place the end effector on the line spanning between the current joint and the target, thus reducing the distance between the end effector and the target as much as possible (Fig. 2.4). The algorithm is repeated iteratively until the distance between the end effector and the target is smaller than some threshold.

While the CCD algorithm is very fast and simple to implement, it often produces unrealistic movements and unnatural poses when applied to a kinematic chain, partly due to the fact that the joints closer to the end effector have a high impact on the resolution of the problem, and the joints further away have a smaller impact. The unequal distribution of adjustments in the chain's configuration lead to chaotic animations. The algorithm also has a lesser affinity for solving IK problems with multiple end effectors, and to overcome this, a system with multiple end effectors must first be broken down into independent chains.

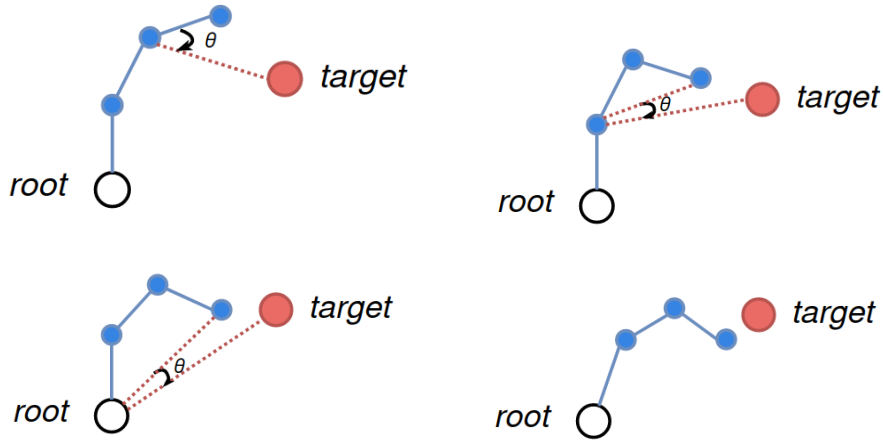


Figure 2.4: A single pass of the CCD algorithm

Forward and Backward Reaching Inverse Kinematics

The Forward and Backward Reaching Inverse Kinematics (FABRIK) algorithm [2] is another heuristic approach to the IK problem which gained popularity in computer graphics and video games.

Like the CCD algorithm, FABRIK iterates over the joints of a kinematic chain, adjusting them one at a time. However, this algorithm involves two passes per iteration where, as the name suggests, one pass is done starting from the end effector iterating towards the root, while the second pass is done in a reversed order as shown in Fig. 2.5. The algorithm starts by checking if the target is in reach before proceeding to iterate over the kinematic chain.

For a set of n joint positions $\mathrm{p}_1, \ldots, \mathrm{p}_n$ where p_1 is the root and p_n is the end effector, the forward pass begins by setting the position of the end effector to that of the target. The next joint p_{n-1} is then linearly interpolated on a line which passes through its current position and the position of the previously affected joint, in this case, the end effector. The interpolation is done to preserve the distance between the joints. Thus, the forward pass of the algorithm can be modeled by

$$\mathrm{p}_i = (1 - \lambda_i)\mathrm{p}_{i+1} + \lambda_i \mathrm{p}_i, \tag{15}$$

where λ_i is a coefficient which ensures that the linear interpolation preserves the distance between joint positions p_i and p_{i+1}. When the forward pass is finished, the root ends up being displaced from its original position, and in order to rectify this a backwards pass is initiated in a reverse order which starts by setting the root back to its original position. The

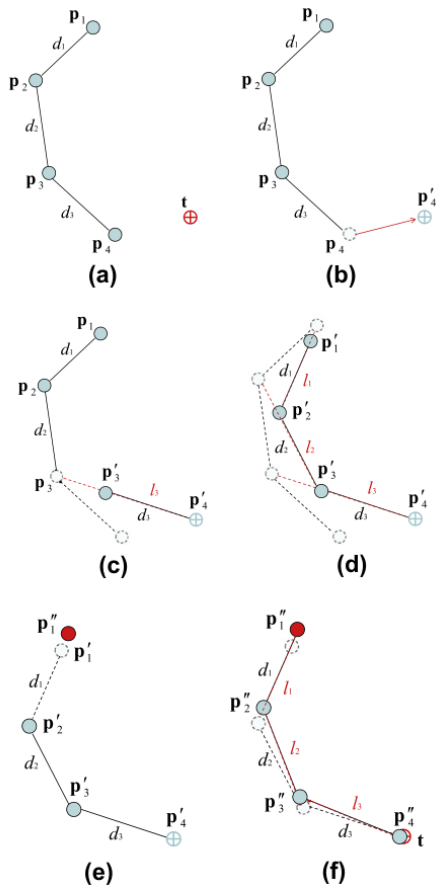


Figure 2.5: An iteration of the FABRIK algorithm where (a) through (d) are detailed steps of the forward pass while (e) and (f) show the solution of the backward pass. Source: [2]

iteration proceeds very similarly to the forwards pass, this time interpolating each joint p_i between its current position and the new position of p_{i-1}. The backwards pass is then

$$\mathrm{p}_i = (1 - \lambda_i)\mathrm{p}_{i-1} + \lambda_i \mathrm{p}_i. \tag{16}$$

The iterations are repeated until the distance between the end effector and the target are smaller than some threshold, at which point the problem can be considered solved.

The FABRIK algorithm is very fast due to its calculations relying on adjusting positions along a line instead of calculating joint angles. This also majorly simplifies its implementation. It also provides a more natural configuration as a solution to the IK problems compared to the CCD algorithm as the degree to which each joint is adjusted during the iterations is not unevenly distributed along the chain. Additionally, the FABRIK algorithm is a good fit for computer graphics and animation because of its ability to handle multiple end effectors well and to be constrained in multiple ways.

For the purpose of this paper, the FABRIK algorithm was chosen as the algorithm to be implemented in the demo application. The choice helps avoid the problems with singularities which exist when using the Jacobian methods, the complex implementations and heavy calculations of the Newton methods, and instead provides a fast and lightweight algorithm which is simple to implement and has a better chance of producing more natural kinematic chain configurations than the CCD algorithm. The FABRIK algorithm also finds its implementations in many software programs, including the built-in Unity *Animation Rigging* package [19].

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 presets, which is shown in Fig. 3.1, is specifically designed to be used for video games.

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. 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 mesh's 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 key frames. This is used to create baked animations for characters and objects, as well

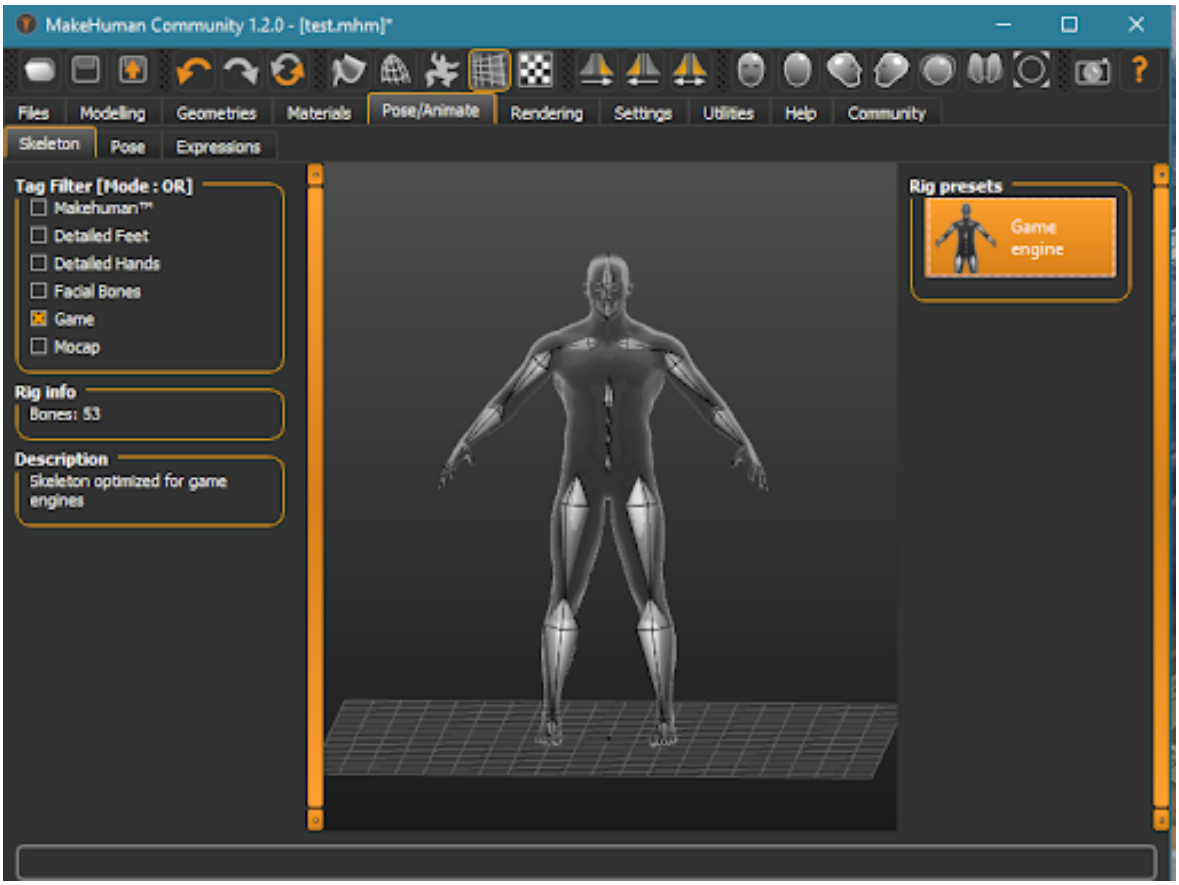


Figure 3.1: MakeHuman rig selection

as defining animations that are later blended with IK procedural animations. An animated model can be 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 [13]. 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 a 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 which control the flow between each clip in the form of a state machine (Figure 3.3). Animations must be added to an Animator controller in order to be used by a

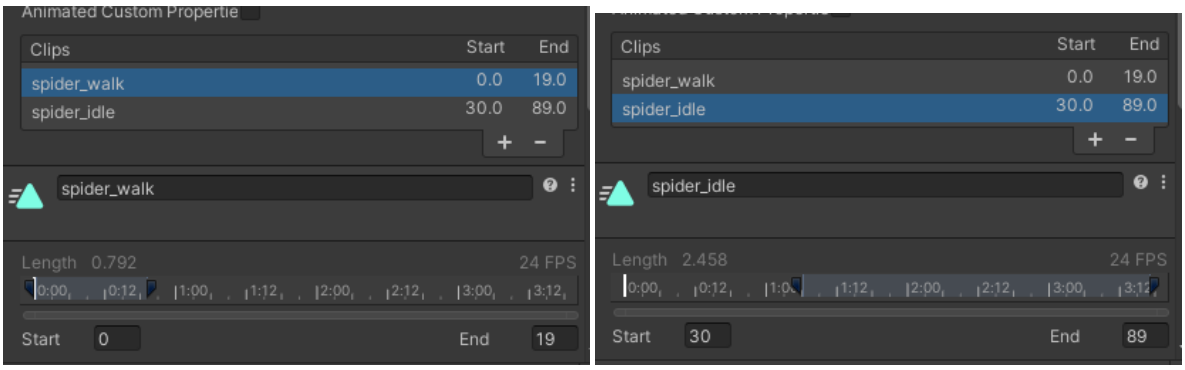


Figure 3.2: Animation clips extracted from a single animation during import

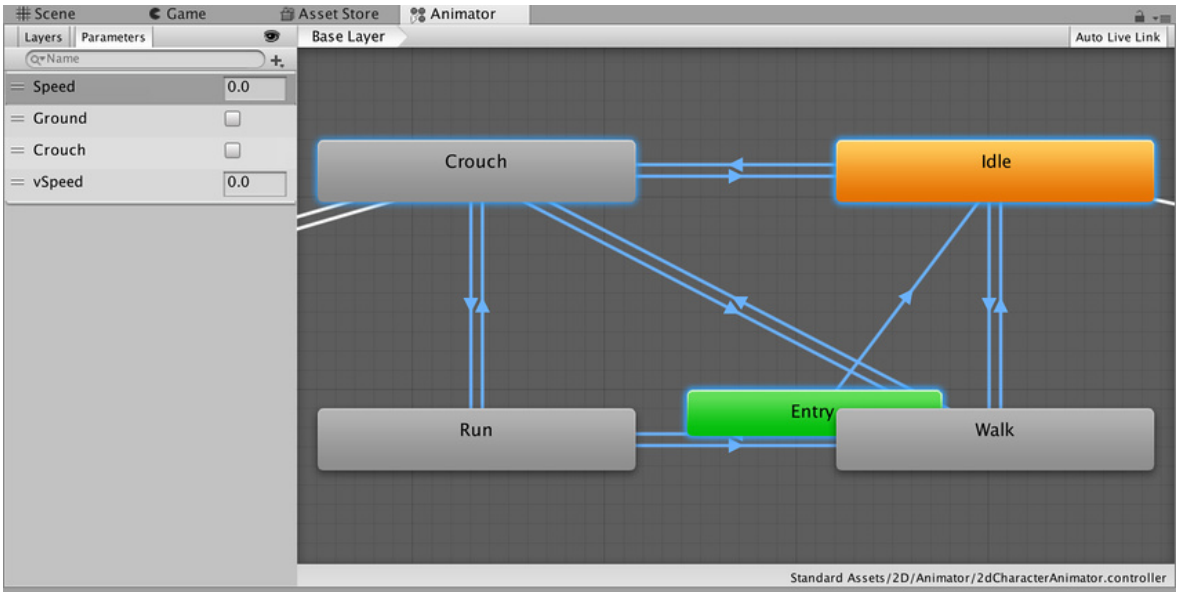


Figure 3.3: Animator controller animation state machine [12]

Unity *GameObject* [12]. 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 [17]. 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 [16]. Humanoid characters have a few additional functionalities. Namely, because of the criteria required to qualify as a humanoid model, these models are all similar in structure and as such, animations can be mapped

Figure 3.4: Predefined rig constraints available in Unity's Animation Rigging package [11]

from one humanoid model to another [15]. Additionally, these model's 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 [17, 15]. 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 which 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.

Before any transformations are applied to the joint transforms, their positions are copied. All operations and calculations are performed on the copied transforms and at the end of the full pass, the new positions are copied back to the kinematic chain.

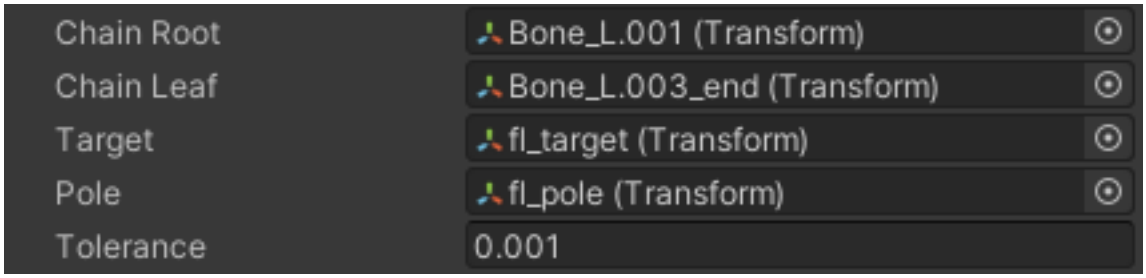


Figure 4.1: FABRIK script parameters

```
void LateUpdate()
{
    CopyTempPositions();
    ...
    ApplyTempPositions();
}
```

Listing 4.1: Joint transforms are copied at the start. Operations are performed on these copied joints, before applying them back to the kinematic chain

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 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.

```
void LateUpdate()
{
    ...
    float distRootToTargetSqr = (
        target.position - chainRoot.position
    ).sqrMagnitude;
    if (distRootToTargetSqr > totalDistance * totalDistance)
    {
        StretchToTarget();
    } else {
```

```
        ...
    }
    ...
}

void StretchToTarget()
{
    for (var i = 0; i < chainLen; i++)
    {
        float jointDistToTarget = (
            target.position - tempPositions[i]
        ).magnitude;
        var lambda = jointDistances[i] / jointDistToTarget;

        tempPositions[i + 1] =
            (1 - lambda) * tempPositions[i] + lambda * target.position;
    }
}
```

Listing 4.2: Target out of reach

If the target is within the reach of the kinematic chain, the iterative process of forward and backward passes, after which the algorithm is named, is executed. 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 as well. If the custom IK script updates 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 vector is then defined which spans between the end effector and the following joint. This neighbor's position is then interpolated along the vector 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.

```
void ForwardReachingPass()
```

```
{
    tempPositions[chainLen] = target.position;
    for (var i = chainLen - 1; i >= 0; i--)
    {
        var dist = (tempPositions[i + 1] - tempPositions[i]).magnitude;
        var lambda = jointDistances[i] / dist;
        tempPositions[i] =
            (1 - lambda) * tempPositions[i + 1]
            + lambda * tempPositions[i];
    }
}
```

Listing 4.3: FABRIK forward reaching 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.

```
void LateUpdate()
{
    ...
    else
    {
        var distEffectorToTargetSqr = (
            tempPositions[chainLen] - target.position
        ).sqrMagnitude;
        while (distEffectorToTargetSqr > tolerance * tolerance)
        {
            ForwardReachingPass();
```

```
            BackwardReachingPass();

            distEffectorToTargetSqr = (
                tempPositions[chainLen] - target.position
            ).sqrMagnitude;
        }
        ...
    }
    ...
}
```

Listing 4.4: Main iteration loop of FABRIK

While the basic FABRIK algorithm allows a kinematic chain adjust so that the end effector reaches a defined target, the lack of control over this process can lead to unnatural poses in certain use cases, defeating the purpose of the procedural animations which are designed to produce a more natural effect. In order to achieve a higher degree of control in one of the use cases described in the following section, pole target constraints were implemented as an optional supplement to the existing algorithm. This approach to the pole target algorithm is inspired by a video demonstrating this concept [7].

The pole algorithm acts on every joint in a given kinematic chain excluding the root and the end effector which already have defined target positions. For each of the iterated joints $p[i]$, calculations must take into account the positions of their preceding joint $p[i-1]$ and succeeding joint $p[i+1]$. A plane is constructed at the position $p[i-1]$, with a normal vector which points from $p[i-1]$ to $p[i+1]$. Both the pole and the currently manipulated joint are then projected onto the plane (Figure 4.2a) using Unity plane's *ClosestPointOnPlane* method. The *Vector3.SignedAngle* method then allows an angle to be found between both projections by passing the plane's normal vector as the angle axis (Figure 4.2b). Finally, the current joint's desired position is calculated by rotating the vector pointing from $p[i-1]$ to $p[i]$ around the plane's normal by the obtained angle, using the *Quaternion.AngleAxis* method. This can be imagined as lining up the joint so that its planar projection exists on the line between the origin of the plane and the pole target's planar projection (Figure 4.2c). The joint keeps its relation to its preceding and succeeding joints (the angle between vectors pointing from the joint to both surrounding joints is the same), but it is rotated in order to be

positioned as close to the pole target as possible.

```
void LateUpdate()
{
    ...
        while (distEffectorToTargetSqr > tolerance * tolerance)
        {
            ...
        }
        if (pole != null)
            BendToPole();
    }
    ...
}

void BendToPole()
{
    for (int i = 1; i < chainLen; ++i)
    {
        var plane = new Plane(
            tempPositions[i + 1] - tempPositions[i - 1],
            tempPositions[i - 1]
        );
        var projectedPole = plane.ClosestPointOnPlane((pole.position));
        var projectedBone = plane.ClosestPointOnPlane(
            tempPositions[i]
        );
        var angle = Vector3.SignedAngle(
            projectedBone - tempPositions[i - 1],
            projectedPole - tempPositions[i - 1],
            plane.normal
        );
        tempPositions[i] =
            Quaternion.AngleAxis(angle, plane.normal)
                * (tempPositions[i] - tempPositions[i - 1])
            + tempPositions[i - 1];
    }
}
```

Listing 4.5: Pole target constraint

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 is done procedurally.

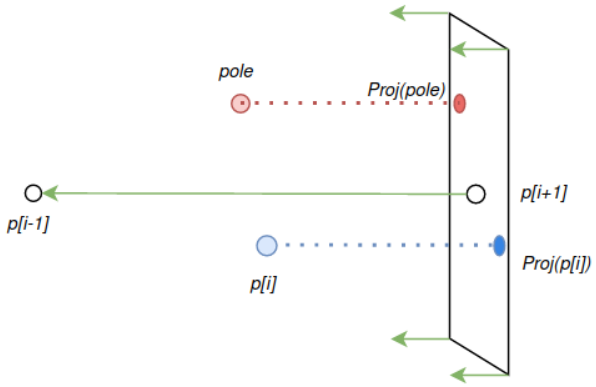
4.2.1 Project Setup

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, and a set of empty objects which the four IK scripts are attached to. The prefab also contains sets of raycasts and targets. The raycasts serve to scan the surface of the terrain under the spider, and mark the targets to which each leg should move.

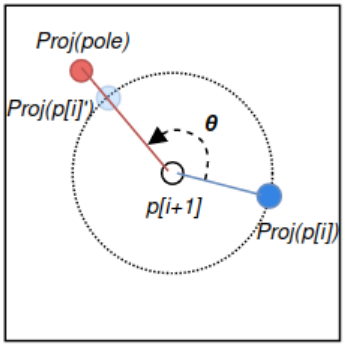
Raycasts are dispatched from above the spiders legs, and aim in the creatures local negative Y axis. This ensures that no matter what orientation the spider finds itself in, the rays are always pointing at the surface which it is standing on. Masks are applied to the rays, making sure that only terrain 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

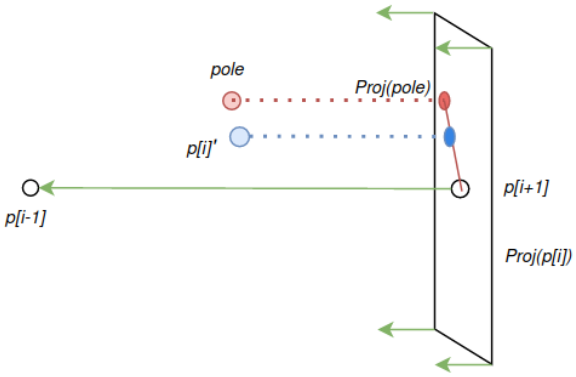
With the project set up in this manner, scripts must now be added on to make the scene functional. A script is required for the main movement of the spider, the raycast logic, and the mechanism which controls the ik targets.



(a) The initial setup where a plane is constructed with the vector from *p[i+1]* to *p[i-1]* as the normal. Both the current joint *p[i]* and the pole target are projected onto the plane



(b) A view of the plane and the angle between the projections which dictates the new position of the joint *p[i]*



(c) The final position of the joint as a result of the pole target mechanism

Figure 4.2: A diagram showing the concept behind the implemented pole target mechanism

Raycasts

The raycast 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 raycast 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 raycast 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 raycast 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 raycast 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 raycast section, the IK targets cannot simply be set to track the raycast hit points. The following is an outline of the rules specified for the IK targets, which define if it should start moving towards its raycast hit target:

- A target must be grounded to be eligible for a movement sequence.
- A target will only begin moving towards the raycast hit point if the distance between them is above a specified threshold.
- A target is only allowed to begin moving towards the raycast hit point if both legs on the opposite diagonal are grounded (See Figure 4.3).

When a target satisfies all of these conditions, it makes note of the raycast 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 raycast hit target's position, which was recorded right before the

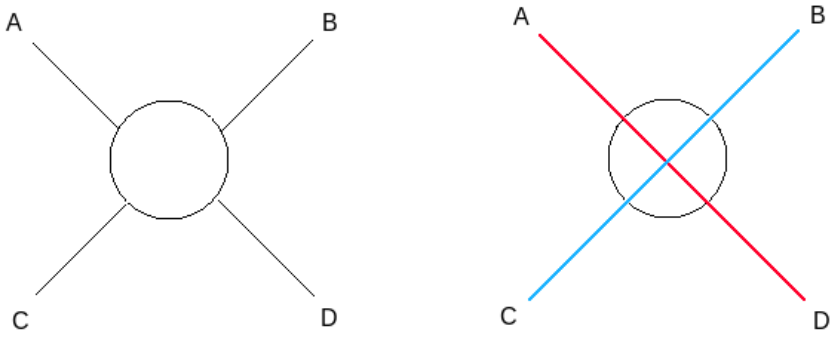


Figure 4.3: The diagonals referencing the spiders legs which are used when checking if a target is allowed to begin moving, where A and D are on an opposite diagonal to B and C

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 with an offset to achieve an arc-like movement. This produces the effect of lifting the spider's leg.

```
float distRemaining = Vector3.Distance(transform.position, _moveTo);
float heightOffset =
    distRemaining > _totalDist / 2.0f ? heightRise : 0.0f;
transform.position = Vector3.MoveTowards(
    transform.position,
    new Vector3(_moveTo.x, _moveTo.y + heightOffset, _moveTo.z),
    Time.deltaTime / 0.1f
);
if (Vector3.Distance(transform.position, _moveTo) < 0.001f)
    Grounded = true;
```

Listing 4.6: Movement logic for the legs' IK targets

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 raycasts.
- 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 in which 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.

```
Quaternion calculateRotation()
{
    Vector3 v1 = frLeg.position - blLeg.position;
    Vector3 v2 = flLeg.position - brLeg.position;

    _up = Vector3.Cross(v2, v1);
    _forwards = Vector3.Cross(transform.right, _up);
    return Quaternion.LookRotation( _forwards, _up) ;
}
```

Listing 4.7: Calculating the spider's rotation

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 raycast 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 raycasting objects are offset either forwards or backward depending on the direction of the movement. This is done because the raycasts 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.

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 raycast 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 raycasting objects are offset either forwards or backward depending on the direction of the movement. This is done because the raycasts 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.

```
if (Mathf.Abs(yVal) > 0.9f)
    rays.position = transform.position
        + _forwards.normalized * yVal * 0.3f
        + _up.normalized * 1.095f;
```

Listing 4.8: The raycasts which scan the terrain for leg placement positions are offset in the direction in which the spider is walking. The *yVal* variable is the value of the *Vertical* axis input

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 raycast 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 keeps 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 should be adaptable.

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. 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 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. To be able to set the origin to the appropriate position, a reference to the hand transform is taken in as the last parameter of the script.

The main functionality of the script revolves around Unity's coroutines. Coroutines enable a method to pause its execution and then continue where it left off on the next frame [14]. 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 created 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, then 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. 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 and the execution of another animation should not be blocked.

5. Experiments

The main purpose of this chapter of this paper is to compare animations generated using IK with baked animations. The comparison is broken up into two categories - visual and performance.

5.1 Baked animations

In order to conduct the experiments comprised of visual and performance comparisons, a second set of animations were created for each of the examples which are shown in the demo application. Below is an explanation of the process of creating this second set of animations.

Spider

The baked animation version of the spider was animated in Blender. The model has both idle and walking animations which, in Blender, are placed on a single timeline, one after the other (Figure 5.1). Once the model is imported into Unity, the animations can be broken up into their separate cases, as shown earlier in the "tools" chapter of this paper (Figure 3.2).

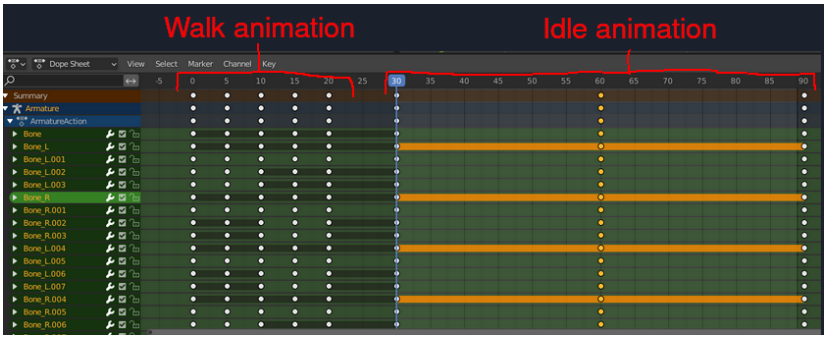


Figure 5.1: Two animations on a single timeline

Once the animations have been imported, a new animation controller is created for the new spider. The animations are added, this time with the use of a blend tree (Figure 5.2) to, as the name suggests, blend between the animation smoothly. A third animation state is added to the blend tree which is the equivalent of the walking animation, but the animation speed is set to a negative value. This plays the animation in reverse and is used when the

spider is walking backwards.

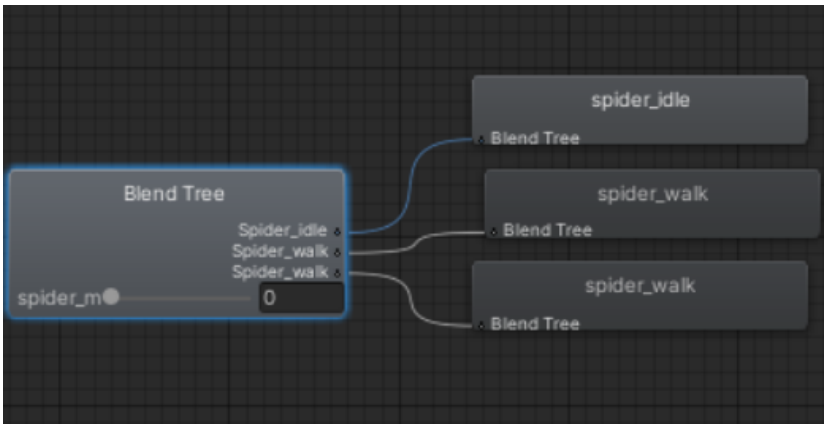


Figure 5.2: Blend tree containing animations for the spider

A variable named *spider_movement* is created in order to control the animations that are to be played in a given situation, and the manner in which the transitions should be blended. The thresholds for each animation can be defined in the blend tree's configuration in the animator (Figure 5.3).

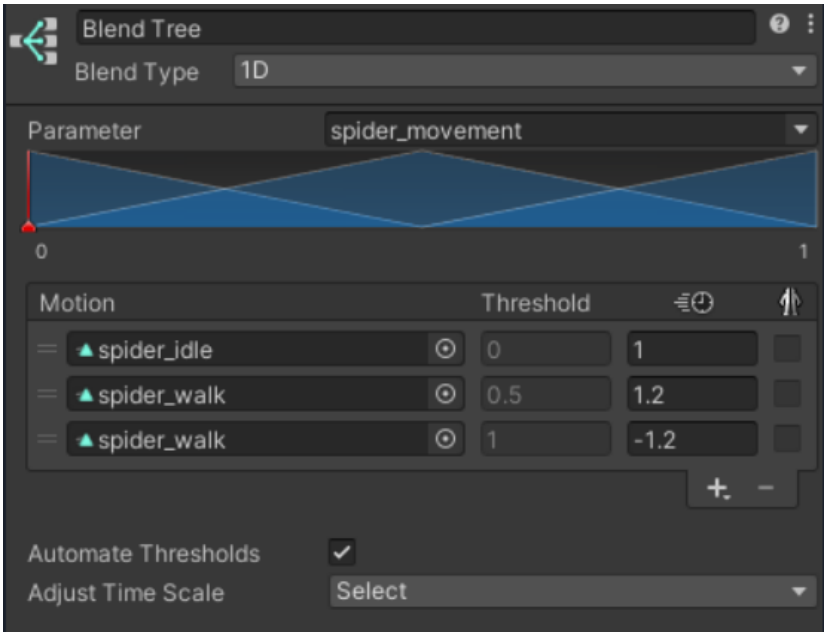


Figure 5.3: Configuration of the spider's blend tree which is dependent on the *spider_movement* variable

In the spiders movement script, this variable can then be set in reaction to certain inputs so that the proper animation is activated for each given situation. To achieve the transition blending, the *spider_movement* variable should not be set outright to the value which corre-

sponds to the next animation state. Instead, the *_animator.SetFloat* method is used, where the *_animator* is a reference to the spider's Animator component. This method allows the value of *spider_movement* to be interpolated from the value of the current animation state to another desired value.

```
if (verticalAxis == 0f && horizontalAxis == 0f)
    _animator.SetFloat("spider_movement", 0f, 0.05f, Time.deltaTime);
```

Listing 5.1: Transitioning to the spider's idle animation using the *SetFloat* method

This version of the spider has its movement based on the spider from the game Minecraft, which means that its rotation on the x and z axes is locked. The spider can rotate about the y-axis when turning to face a different direction, but it will not adjust to variations in the surface which it walks upon. Additionally, when the spider encounters a wall, it begins moving vertically instead of horizontally until it scales the entire obstacle.

Human

The baked version of the human character, which performs an animation sequence consisting of pressing buttons, is animated in Blender. The animation plays one full sequence of pressing a single button. Chunks of the same animation are used in the IK version of this animation sequence, described in the previous chapter, but the baked animation uses the full animation while the IK version uses only the beginning and the end. However, the animation is still broken up into three parts when imported into Unity: the raising of the hand, the button pressing motion, and the lowering of the hand (Figure 5.4). This is done because when the character is pressing multiple buttons in a row, the hand should not be lowered to its starting position after every press.

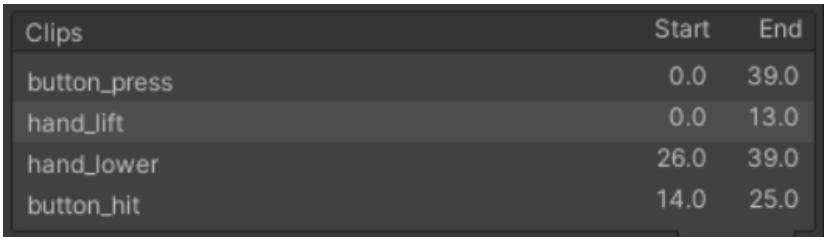


Figure 5.4: The full button press animation is broken up into 3 separate clips

Another animation controller is created for this version of the human. Unlike the spider example, the animation states do not have to be blended as they are all clips which combine to create the full button press animation, and the transitions are seamless as they are. Because of this, the animation states for the three animation clips are not part of a blend tree, and instead are just "floating" states with no defined transitions. The logic for which animation should be played at what time is defined in the main script attached to the character.

The script is a simplified version of the one which controls the IK version of this character's animation. It has no need for the public parameters present in it's IK version because there is no IK rig, IK target, or button transforms which it needs to control. Due to this, there is no list containing the sequence of buttons to be pressed. It is instead replaced by an integer value which dictates the number of button presses to execute in one animation sequence in order to convey the idea that the character is pressing multiple buttons in a sequence.

As with the IK version of this script, the logic is based on a set of coroutines which control the flow of the animation sequence. However, only the *HandAnimation* coroutine is used as a building block for the sequence because the whole action is now constructed using baked animations. When the script receives an input and the animation is not already playing, the sequence begins by setting the *idle* variable to false in order to prevent the sequence from being repeated while it is still in progress. All required animation clips are then set off one after the other, starting with the animation to lift the hand. This is then followed by the animation clip which is responsible for hitting the button, and it is repeated in a loop for the number of times defined by the button press count parameter. Finally, the animation clip for lowering the hand is executed, and the *idle* boolean is set to true before terminating the sequence.

5.2 Visual Comparison

The goal of using inverse kinematics to procedurally animate a model in video games is to achieve a more natural interaction between a character and the environment. The realism of the interaction is conveyed through how it looks, and how the animation is a more realistic representation of how such an action might look in real life. Therefore, the visual integrity

of the IK animation as it adapts to a plethora of scenarios is what sets it apart from a classic approach through baked animations.

Spider

The walking animation, or any basic movement animation, is the core of what most characters will be displaying a majority of the time. That being the case, a natural movement animation will contribute the most to the overall look and feel of the character. This difference can be clearly noticed by comparing the two version of the spider implemented in the demo application created for the purpose of this paper.

Starting with the idle position, both models look quite natural standing on a flat surface, as seen in Figure 5.5.

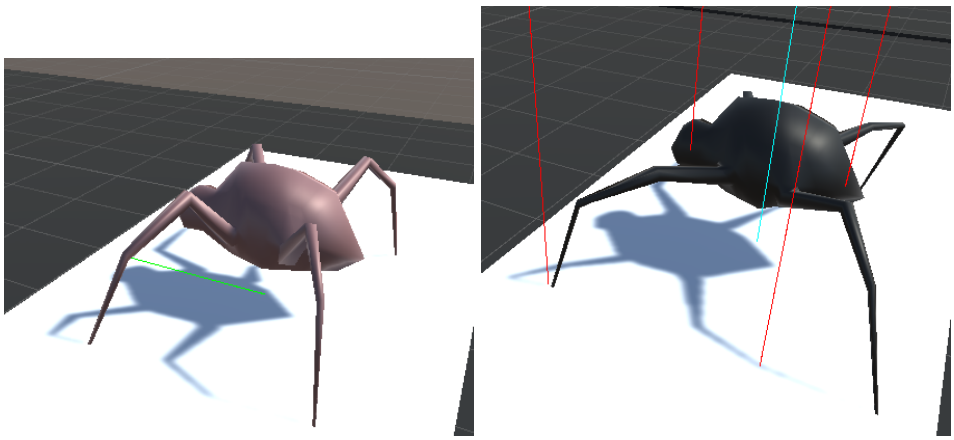


Figure 5.5: A comparison of both models standing on a flat surface

However, this changes when the surface is uneven. The IK acting on the legs of the spider on the right in Figure 5.6 is able to use its ray casts to match the surface exactly, while the baked version of the spider (left) doesn't have this mechanism, and its legs can be seen floating in the air. The same effect is observed when the spider is moving. The baked version's legs float when it is moving over uneven or slanted surfaces, which makes it seem as if the spider was swimming through the air.

Another difference between the two spiders is how they climb walls. Whereas the IK version is able to treat it the same as any other surface, the baked version keeps does not adjust its rotation, and instead climbs the wall head on (Figure 5.7). This kind of rotation could have been implemented, however, the transition between floor and wall would be very

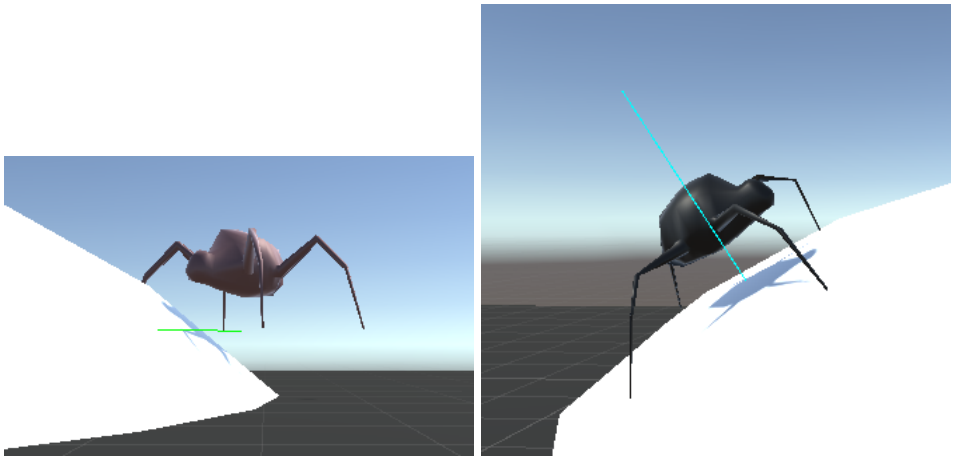


Figure 5.6: A comparison of both models standing on an uneven surface

unnatural either way and so an implementation inspired by the spiders in the popular game Minecraft was used instead.

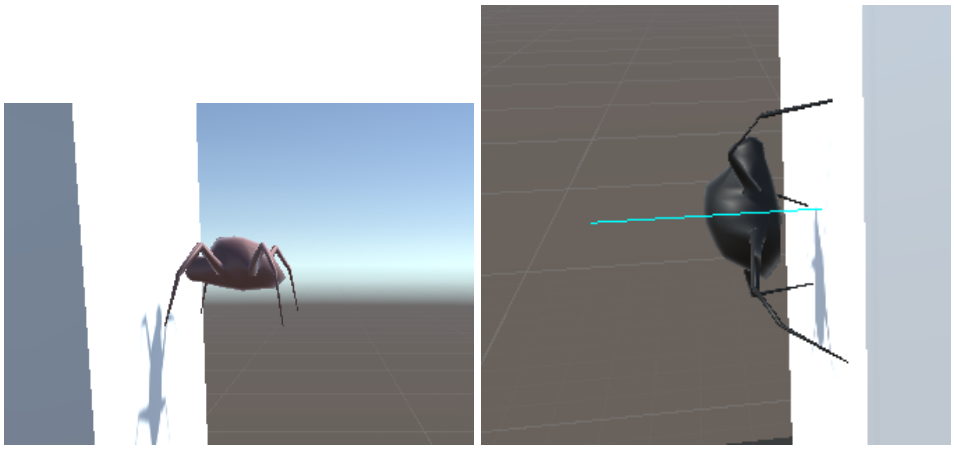


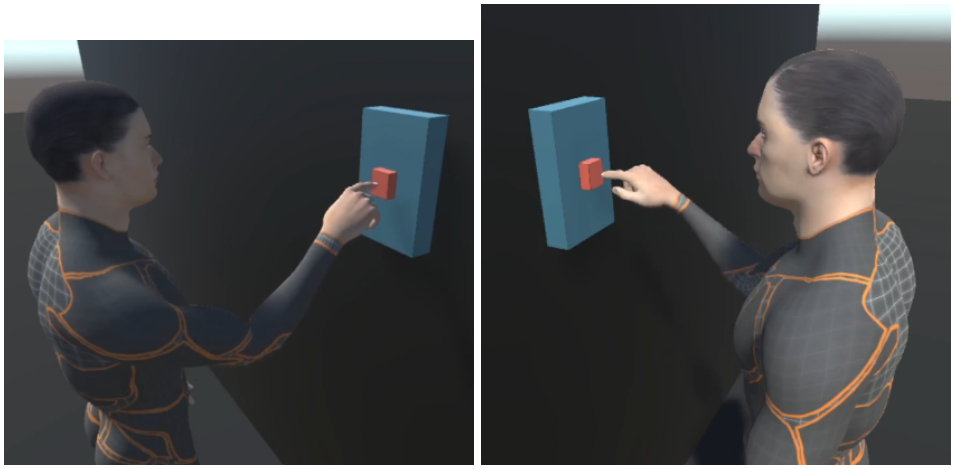
Figure 5.7: A comparison of both models scaling a wall

One important aspect to note when comparing the realism of both animations is how the legs move relative to the surface on which they stand. In the case of the procedural animation, the IK targets controlling the spider's legs remain fixed in one place until a leg movement is initiated. As a result, no matter the speed at which the spider is moving, the give the impression that they are placed on the floor and push away realistically. On the contrary, the baked animation is played at a certain speed, and it is very hard to match the animation speed to the spider's movement speed so that the legs seem to stay in a fixed position on the ground while the spider is moving. This breaks the illusion of the legs pushing the spider forwards, as they instead glide on the surface out of sync.

Human

In a game where pressing buttons is not a core mechanic, the accuracy of such an animation sequence may not be important, and a generic animation can be used to convey the idea. However, in the event that the sequence of the buttons is important, such as a player needing to provide a specific input to hack a code to open a door, the realism of the game can be elevated through the use of inverse kinematics. The adaptability of a character's action can be seen in this comparison between the baked and IK versions of the button press animation.

When the character is only required to press a single button, while also having a proper position and orientation relative to the button, the baked animation looks quite natural, and looks no different from the IK version (Figure 5.8)

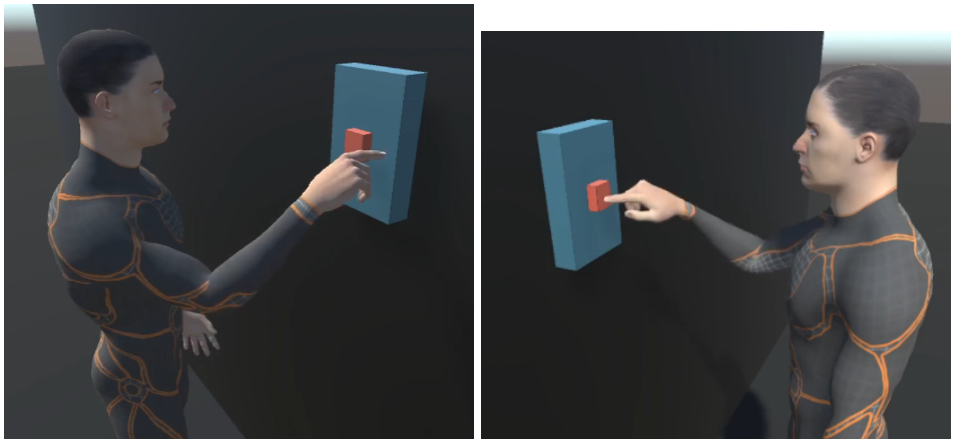


(a) Baked animation (b) IK animation

Figure 5.8: A comparison of both models pressing a single button

The difference starts to be noticeable when the character is offset or rotated from the default position. While the IK target leads the hand correctly to the button (Figure 5.9b), the baked animation has no knowledge of the characters surroundings, and misses its mark (Figure 5.9a). Due to this, for realism to be preserved when using a baked animation, either the character must be reoriented as part of the action, or a cutscene should be played for the duration of the action.

The next benefit of the procedural animation is its ability to adapt to a panel with multiple buttons. The character which is using the baked animation is only able to aim its hand at one specific point, consequently missing all the buttons (Figure 5.10). In the



(a) Baked animation (b) IK animation

Figure 5.9: A comparison of both models pressing a single button after offsetting the position and rotation of the characters

case of a sequence of buttons being pressed, the character repeats the press in a single spot which doesn't realistically convey the pressing of a sequence of buttons. This can be slightly improved by constructing the animation to aim at different buttons on the panel in a generic order. The downside of this approach is that it falls apart if there is a change in the amount of buttons, or their positions and configuration.

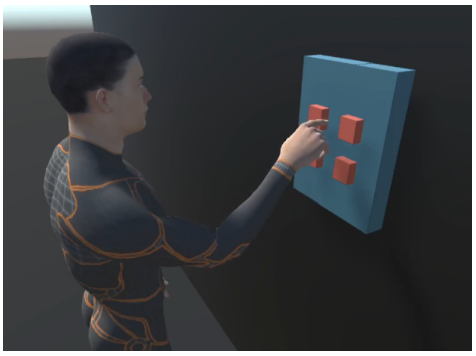


Figure 5.10: The baked model executing the button press animation on a panel of buttons

Comparing this to the IK animation, as seen in Figure 5.11, the IK target leads the hand to the appropriate position for each separate button. This allows for a specific sequence of buttons - taking the door hacking example from before this would be a player's input - to be pressed in a way which visually conveys the exact action which was commanded, in a responsive manner.

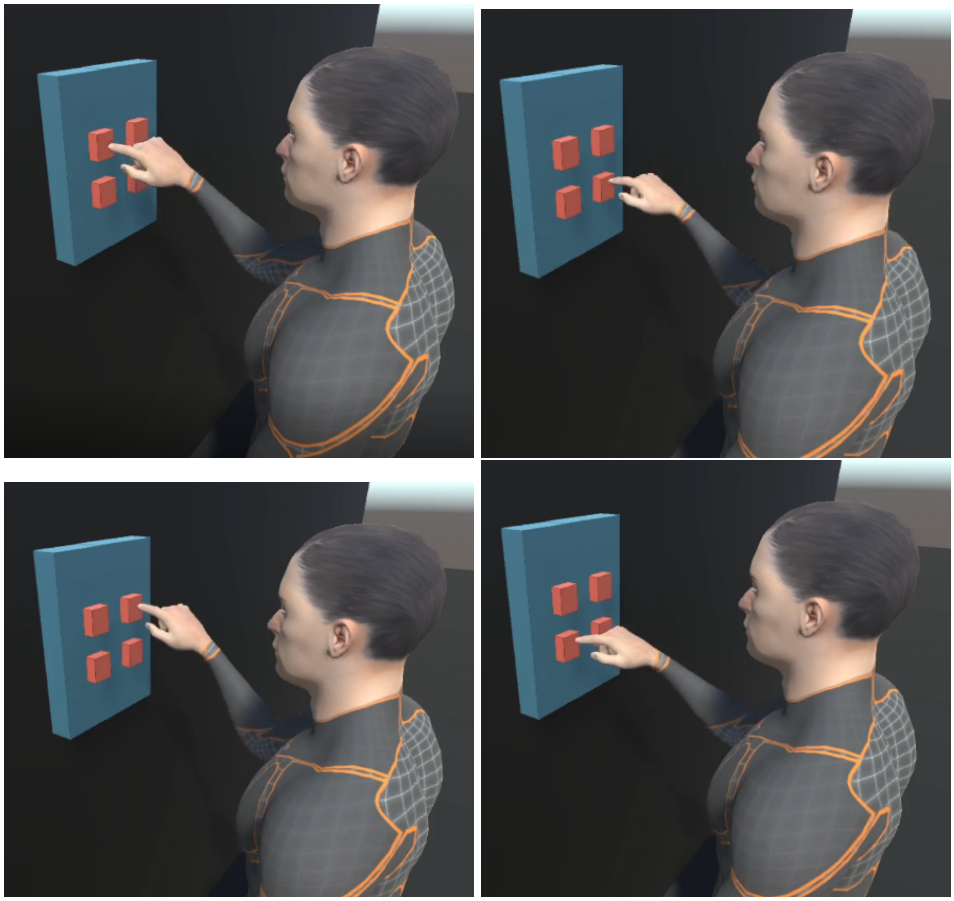


Figure 5.11: The IK model pressing each of the four buttons on a panel

5.3 Performance Comparison

One thing to consider when implementing a task procedurally is the effect it may have on the performance of the application. Resorting to the use of code and calculations instead of executing a predefined sequence introduces the potential of increasing the CPU usage and decreasing performance. An analysis was conducted on the examples created in the demo application using the Unity Profiler [18]. Each experiment was carried out by spawning 100 objects of each type and comparing the CPU usages both while they were idle and while they were moving. Some usage categories were discarded as they were not relevant to the analysis (Figure 5.12).

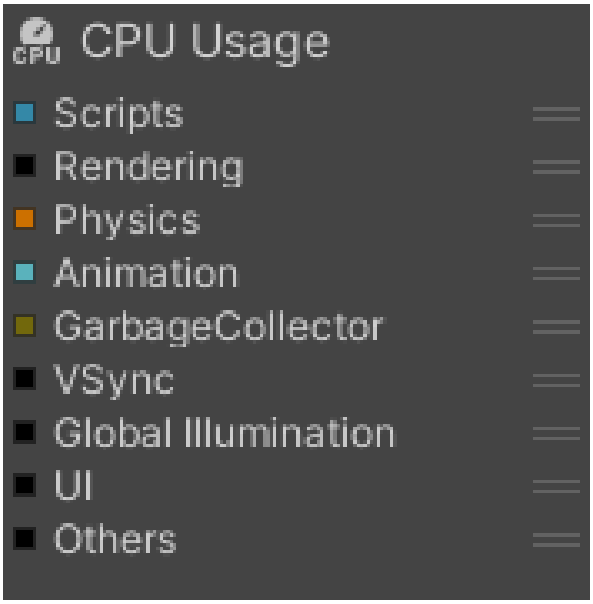


Figure 5.12: Usage categories selected when analyzing CPU usage

Spider

Starting with the baked version of the spider, a hundred duplicates spawned simultaneously produced the result seen in Figure 5.13. Similarly to the IK version presented further down, most of the CPU usage comes from the "scripts" usage category. A clear distinction can be made between when the spider was moving and when it was not. During the profiling, the state of the spider started out as idle, and then alternated between moving and idle at approximately each quarter of the graph presented. In the idle state, the CPU usage for the selected categories remains steadily below 10 milliseconds. However, when the spider is moving the usage rises with an initial spike which is almost double that of the idle state, and then lowering to around 10 milliseconds. The increase in CPU usage, as seen from the profiler, is mostly in the "scripts" category, which means that the animations themselves are not influencing the performance of the application too much.

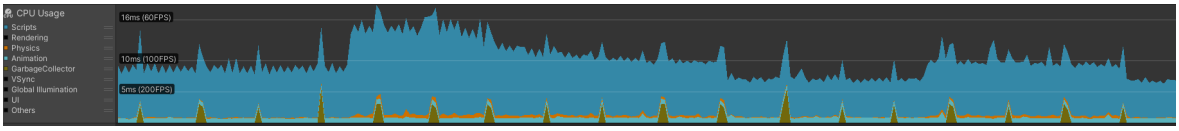


Figure 5.13: Profiling of a hundred instances of the baked animation spider for CPU usage

The IK version of the spider has a much more steady average regardless of its moving

or idle state. In Figure 5.14, the graph rises minimally at the halfway mark which is where the spiders were transitioned to a moving state. The lack of distinct change between states can be attributed to the fact that the IK calculations done on the kinematic chains and their respective targets are done regardless if the spider is moving or not. The same can be said for the raycasts and body rotation calculations which are executed no matter the state. The slight increase is most likely due to the calculations required to determine the forward movement vector when the spider receives input.

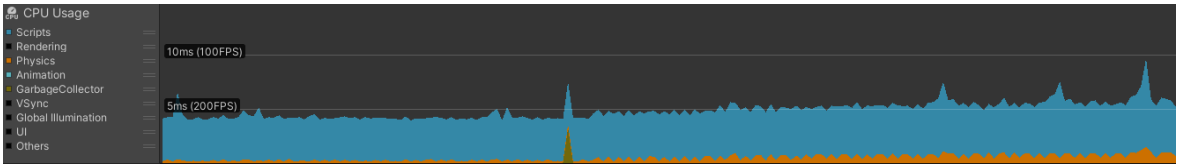


Figure 5.14: Profiling of a hundred instances of the procedurally animated spider for CPU usage

The average CPU usage for the IK version of the spider is around half that of its baked counterpart. This is most likely due to the baked animation spider requiring more checks and calculations which pertain to the physics of its movement. The IK version does not need many physics checks as the simple ray cast system keeps the spider stuck to the surface, and combined with the IK mechanism it moves the spider along without the need for collision checking. However, this is relevant for the implementation presented in this demo application, and may not be the case if a given use case required the spider to be able to register collisions and be affected by gravity.

Human

The purpose of the human character in the demo application is to demonstrate a specific animation sequence executed through the means of baked and procedural animations. Because there are no collisions, movement, or other checks, the baked animation character's script contains almost no major calculations. Since baked animations do not have a big influence on the CPU usage as seen earlier in the case of the spider, the performance of the application does not vary in relation to when the button pressing sequence is executed. Figure 5.15 displays the results of profiling one hundred instances of the baked animation human characters executing the animation sequence simultaneously, and no spike in CPU

usage, which hovers around 1ms, can be observed

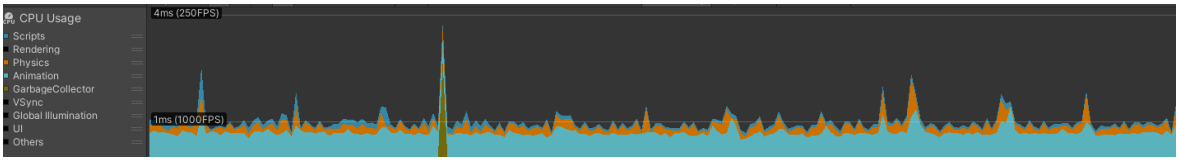


Figure 5.15: Profiling of a hundred instances of the human character with baked animations for CPU usage

In contrast, the procedurally animated human does require a few calculations during the animations sequence. While it does use baked animations for the beginning and end of the sequence, the middle section switches to using inverse kinematics, and additionally the IK targets are interpolated between the hand's origin position and the button target positions. These calculations can be seen in Figure 5.16 where during the execution of the animation there is a noticeable spike in the "scripts" usage category. Other than that, the rest of the graph looks very similar to the baked animation variant, sitting at around 1 milliseconds of CPU usage.

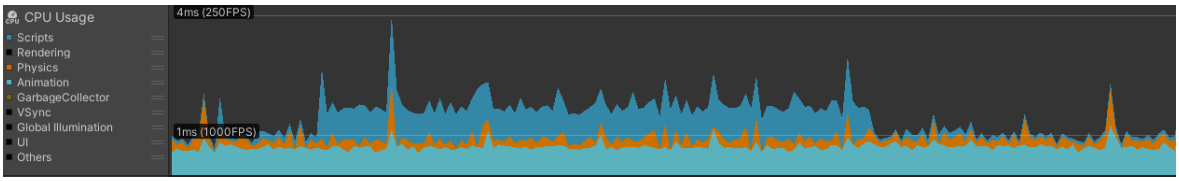


Figure 5.16: Profiling of a hundred instances of the procedurally animated human for CPU usage

6. Conclusion

The aim of this paper was to gain a better understanding of the basic algorithms used in inverse kinematics, to discover the built-in functionalities that the Unity engine offers for such implementation, and compare the IK procedural animations to a classic baked animation approach through visual and performance based metrics. The aim was achieved.

In the second chapter, the theory behind existing IK algorithms was analyzed. As a result of this analysis, the heuristic FABRIK algorithm was chosen for its speed and simple implementation which distinguishes itself from other approaches by avoiding problems such as instabilities and complex calculations.

In the third chapter, an overview was made of technologies used in the implementation of the demo application. Among others, the features provided by the Unity game engine which enable and facilitate the use of IK were outlined.

The third and fourth chapters served to display the implementation details of both the IK solutions and their baked animation counterparts, as well as compare them in the visual and performance categories. In terms of visual realism and natural motion, the IK animations proved to be superior in terms of precise interaction with the surrounding environment, and the adaptability of the characters to various situations. The human animation sequence demonstrated that the procedural animations using inverse kinematics result in a higher CPU usage compared to baked animations. However, the spider movement animation countered this by showing that in certain cases, the procedural approach to animation and movement can eliminate the need for various calculations such as physics checks, and can overall result in a lower net CPU usage compared to a character which has baked animations.

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, 9 2011.

[3] J. Baillieul. Kinematic programming alternatives for redundant manipulators. In *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, volume 2, pages 722–728, 1985.

[4] A. Balestrino, G. De Maria, and L. Sciavicco. Robust control of robotic manipulators. *IFAC Proceedings Volumes*, 17(2), 1984. 9th IFAC World Congress: A Bridge Between Control Science and Technology, Budapest, Hungary, 2-6 July 1984.

[5] Encyclopedia Britannica. Kinematics. https://www.britannica.com/science/kinematics. Accessed: 2023-01-07.

[6] Adrian Canutescu and Roland Dunbrack. Cyclic coordinate descent: A robotics algorithm for protein loop closure. *Protein science : a publication of the Protein Society*, 12:963–72, 06 2003.

[7] DitzelGames. C# inverse kinematics in unity. https://www.youtube.com/watch?v=qqOAzn05fvk, 2019. Accessed: 2022-10-13.

[8] Ben Kenwright. Inverse kinematics – cyclic coordinate descent (ccd). *Journal of Graphics Tools*, 16(4):177–217, 2012.

[9] Jiaoyang Lu, Ting Zou, and Xianta Jiang. A neural network based approach to inverse kinematics problem for general six-axis robots. *Sensors*, 22(22), 2022.

[10] 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.

[11] 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.

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

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

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

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

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

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

[18] Unity User Manual. Profiler overview. https://docs.unity3d.com/Manual/Profiler.html, 2022. Accessed: 2023-01-04.

[19] Unity Packages. Animation rigging - chainikconstraint. https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/constraints/ChainIKConstraint.html, 2020. Accessed: 2023-01-02.

[20] Firas Raheem, Ahmed Sadiq, and Noor F. Abbas. Robot arm free cartesian space analysis for heuristic path planning enhancement. *International Journal of Mechanical & Mechatronics Engineering*, 19:29–42, 02 2019.

[21] W. A. Wolovich and H. Elliott. A computational technique for inverse kinematics. In *The 23rd IEEE Conference on Decision and Control*, 1984.

List of Figures