# POLITECHNIKA BIAŁOSTOCKA

## WYDZIAŁ INFORMATYKI

PRACA DYPLOMOWA INŻYNIERSKA

# TEMAT: SKELETAL ANIMATION USING INVERSE KINEMATICS IN THE UNITY ENGINE

WYKONAWCA: ŁUKASZ JAN BIAŁCZAK

.....................................

podpis

PROMOTOR: DR INŻ. ADAM BOROWICZ

BIAŁYSTOK 2023 r.

**Animacja szkieletowa z wykorzystaniem kinematyki odwrotnej w silniku Unity**

W klasycznej animacji szkieletowej kości reprezentowane są za pomocą pewnej struktury drzewiastej obiektów. Wykonywanie sekwencji animacji polega zwykle na aktualizacji parametrów tych obiektów (położenia i orientacji kości) w porządku od korzenia do liści. W przypadku kinematyki odwrotnej, jak sama nazwa wskazuje, porządek aktualizacji jest odwrotny tj. od danego liścia do określonego węzła nadrzędnego. Metody te pozwalają na proceduralne tworzenie sekwencji animacji umożliwiających wierniejsze odzwierciedlenie interakcji animowanego obiektu z otoczeniem bez konieczności ręcznego "wypalania" sekwencji animacji. W ramach tej pracy, autor zapoznał się z teorią i literaturą na temat animacji szkieletowej i kinematyki odwrotnej, a następnie zaimplementował aplikację demonstracyjną w silniku Unity. Aplikacja ma na celu porównanie animacji "wypalanych" z animacjami tworzonymi z wykorzystaniem kinematyki odwrotnej. Zostało to zrealizowane na przykładzie czworonoga, i sekwencji animacji człowieka. Wyżej wymienione techniki są porównane pod kątem efektu wizualnego oraz wydajnościowego.

# 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 sequences, so called baked animations, 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 [16] to speed up the process of limb manipulation when creating a baked 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, a 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 [9]. 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 [29], 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 [14]. 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    Skeletal Animation

Skeletal animation is a popular technique used to animate characters and objects in computer graphics. It uses meshes and armatures as the two components which make up a single model. The mesh is an outer shell of vertices which acts as the "skin" of the model, and it is the visible representation of a character. The armature is an interconnected hierarchical structure of transforms, often referred to as joints, which each have a degree of influence on the mesh's vertices. Therefore, the manipulation of the armature's joint positions results in the deformation of the model's mesh, similarly to how one might imagine the human body follows the shape of its skeleton. This technique allows for the creation of animations by defining key poses for the armature on a timeline, which are then interpolated for the remainder of the frames. This process of animation, while reducing the control over detail, is much quicker than the manual sculpting and deformation of a characters mesh. It also allows for procedural control of the animations, such as the application of inverse kinematics to the tree-like hierarchy of joints.

The position, rotation, and scale of each joint in a transformation hierarchy are often described using a transformation matrix which holds all three attributes, and can be multiplied with other matrices to combine their transformations. These matrices are usually de-

fined in local space, which means the transformations are relative to those of its parent node. In order to find the positioning of a joint in the scene, these attributes must be converted to world space. A joint's world space transformation can be calculated as

$$W = W_p M, \tag{1}$$

where $W_p$ i the parent joint's world space transformation matrix, $M$ is the current joint's local transformation matrix, and $W$ is the current joint's world transformation matrix which represents its actual position in the scene. In the case of the root node of a transformation hierarchy, both world and local transformation matrices are equal to each other, because its parent node is the scene itself, and so $W_p$ is an identity matrix.

The process of controlling the deformations of a mesh using a set of deformation primitives is called skinning. Many methods exist to define the deformation of a mesh in relation to its armature, however, the basic idea behind them can be demonstrated using the Linear blend skinning method described in part one of [12]. To describe the relation between joints and mesh vertices, we can define a set of $m$ joint transformation matrices $T_1, \ldots, T_m$, and a set of $n$ mesh vertex positions $\mathbf{v}_1, \ldots, \mathbf{v}_n$, where each vertex $\mathbf{v}_i$ has a set of weights $w_{i,1}, \ldots, w_{i,m}$. A weight $w_{i,j}$ represents the amount of influence of joint $j$ on mesh vertex $i$. The equation for a mesh vertex's deformed position $\mathbf{v}'_i$ is as follows:

$$\mathbf{v}'_i = \left( \sum_{j=1}^{m} w_{i,j} T_j \right) \mathbf{v}_i. \tag{2}$$

As seen in the above equation, the deformed position of a mesh vertex is dependent on the combined transformations of the joints which have a non-zero weight for the given mesh vertex. Therefore, weights can be adjusted so that certain joints affect only certain vertices, and to varying extents. Once these weights are calibrated in a way which allows the mesh to adequately adapt to the armature's pose, skeletal animation becomes a powerful tool for the control of a model in computer graphics.

## 2.2  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 [6]. 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 as shown in 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, as demonstrated in Fig. 2.2, an unreachable case exists if the first bone segment is longer than the sum of the remaining bone segments.

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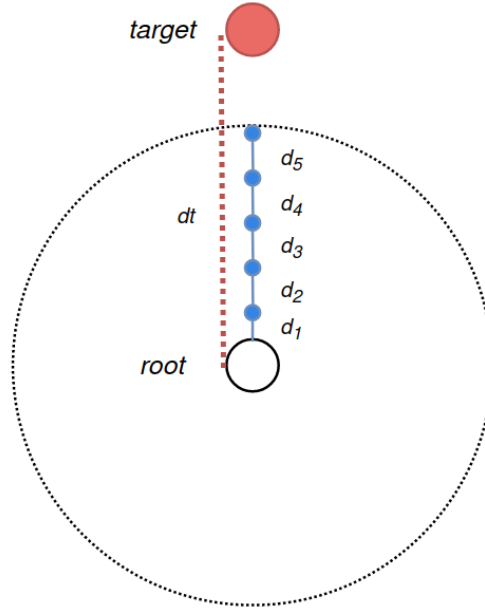
Figure 2.1: An IK problem with no solution where the target is unreachable because the sum of segment distances is less than the distance between the root and the target $\sum_{i=1}^{n} d_i < dt$
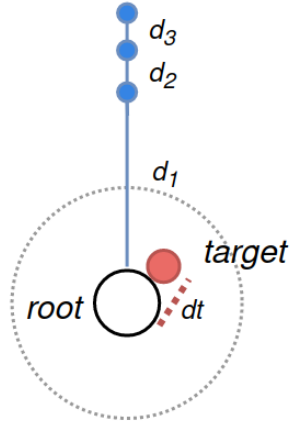


Figure 2.2: An IK problem with no solution where the target is unreachable because the length of the first segment is greater than the summed length of the rest of the kinematic chain $d_1 - \sum_{i=2}^{n} d_i > dt$. This creates a radius around the root where if the root's distance to the target is smaller than the radius, the target is unreachable.
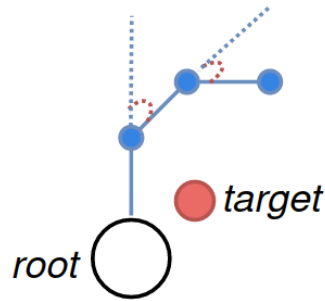
Figure 2.3: An IK problem with no solution where the rotational constraints placed on the joints prevent the end effector from being able to bend enough to reach the target

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 the problem will have multiple solutions, and if the kinematic chain is unconstrained, the solution set for and 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 $dt$:

$$\sum_{i=1}^{n} d_i = dt. \tag{3}$$

## 2.3   Underlying principals

In order to describe the configuration of a given kinematic chain, a set of scalars $\theta_1, \ldots, \theta_n$ called *joint angles* are defined for each of the chain's $n$ joints transforms. A set of $k$ joint positions $\mathbf{s}_1, \ldots, \mathbf{s}_k$ is defined for the end effectors, where these positions are a function of the chain's configuration. With the joint angles in the form of a column vector $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n)^T$, and the end effectors as the vector $\mathbf{s} = (\mathbf{s}_1, \ldots, \mathbf{s}_k)^T$, their relation can be expressed as:

$$\mathbf{s} = f(\boldsymbol{\theta}). \tag{4}$$

The above equation [3] 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:

$$\boldsymbol{\theta} = f^{-1}(\mathbf{s}). \tag{5}$$

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.

## 2.4   IK Algorithms

The methods discussed in this chapter are numerical, which means that they take an iterative approach to achieve a solution. The numerical methods can be divided into the *Jacobian*, *Newton*, and *Heuristic* categories. A comprehensive review of these methods is given by Aristidou and co-workers in [3]. For convenience, the overview of these methods will follow their work.

### 2.4.1 Jacobian methods

This approach to the IK problem [5, 32, 4] 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 angle scalars:

$$J(\boldsymbol{\theta})_{ij} = \left( \frac{\partial s_i}{\partial \theta_j} \right)_{ij} \tag{6}$$

as provided by [7], where $n$ and $k$ are the previously defined number of joints and number of end effectors in the chain respectively. Given the position $\mathbf{p}_j$ of the $j$th joint and its current rotational axis $\mathbf{v}_j$, each entry of the Jacobian matrix can be defined by the cross product

$$\frac{\partial s_i}{\partial \theta_j} = \mathbf{v}_j \times (\mathbf{s}_i - \mathbf{p}_j). \tag{7}$$

With prime notation denoting a first derivative with respect to time, the forward dynamics equation can be written as

$$\mathbf{s}' = J\boldsymbol{\theta}'. \tag{8}$$

A change to the kinematic chain's configuration $\Delta\boldsymbol{\theta}$ 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 $\Delta\mathbf{s}$ due to the modification of the configuration is then approximated as

$$\Delta\mathbf{s} \approx J\Delta\boldsymbol{\theta}, \tag{9}$$

and the changes made to the joint scalars should be done in a way where the change in end effector positions $\Delta\mathbf{s}$ minimizes the error $\mathbf{e}$, which is a vector of changes in position required for each of the chain's end effectors to reach their desired positions. Supposing that $\mathbf{t}_i$ is the target position for the end effector $i$, the errors can be defined as $\mathbf{e}_i = \mathbf{t}_i - \mathbf{s}_i$. By applying a left matrix multiplication by $J^{-1}$ to both sides of (9), the IK problem for the Jacobian method is rewritten as follows:

$$\Delta\theta \approx J^{-1}\mathbf{e}, \tag{10}$$

where $\Delta\mathbf{s}$ is substituted by e due to them being approximately equal to each other.

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* [11] can be used as a substitute:

$$\Delta\theta = J^\dagger \mathbf{e}. \tag{11}$$

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

$$J^\dagger = J^T(JJ^T)^{-1}. \tag{12}$$

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 $\alpha$:

$$\Delta\theta = \alpha J^T \mathbf{e}. \tag{13}$$

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

$$\alpha = \frac{\langle \mathbf{e}, JJ^T\vec{e}\rangle}{\langle JJ^T\mathbf{e}, JJ^T\vec{e}\rangle}, \tag{14}$$

where the angular brackets represent a dot product operation between the two values contained inside them. The value of the coefficient $\alpha$ 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 Singular Value Decomposition method [28] extends the pseudo-inverse method with orthogonal matrices, the Damped Least Squares method [30] helps avoid problems with singularities present in the pseudo-inverse method, the Pseudo-inverse Damped Least Squares method [15, 28] combines the two previously mentioned methods to improve their instability when nearing singularities, and the Selectively Damped Lease Squares method [8] builds on the previous method to converge in fewer iterations and optimize for multiple end effectors. A deep-learning approach to the Damped Lease Squares method was also proposed [31] with the aim of predicting the most optimal value for certain coefficients.

### 2.4.2 Newton Methods

The Newton methods approach IK as a minimization problem, and are based on a second order Taylor series expansion of $f(x + \sigma)$:

$$f(x + \sigma) \approx f(x) + [\nabla f(x)]^T \sigma + \frac{1}{2}\sigma^T \nabla^2 f(x)\sigma, \tag{15}$$

where $f(x)$ is the current joint configuration, $f(x + \sigma)$ is the desired joint configuration, and $\sigma$ is the required modification to $f(x)$ so that it satisfies $f(x + \sigma)$. Lastly, $\nabla^2 f(x)$ is the Hessian matrix, a square matrix of second-order partial derivatives as opposed to the first-order partial derivatives of the Jacobian matrix. Following the method described in [26], the search direction $p_k^N$ is

$$p_k^N = [\nabla^2 f(x)]^{-1} \nabla f(x), \tag{16}$$

where $\nabla f(x)$ is the gradient of the objective function. Assuming the notation of the end effector position as $e$ and goal position as $g$, the gradient can be defined as the Jacobian $\nabla f(x) = J(e - g)$.

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 them to

avoid oscillations. The methods also tend to converge in fewer iterations. 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.

### 2.4.3 Heuristic IK algorithms

Heuristic algorithms 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 [13] 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 [9].

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. This process is illustrated in 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
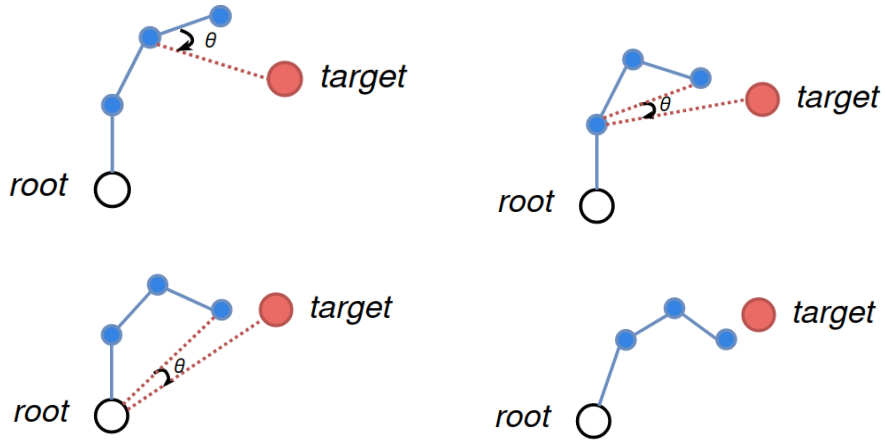
Figure 2.4: A single pass of the CCD algorithm

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.

*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 $\mathbf{p}_1, \ldots, \mathbf{p}_n$ where $\mathbf{p}_1$ is the root and $\mathbf{p}_n$ is the end effector, the forward pass begins by setting the position of the end effector to that of the target. The next joint $\mathbf{p}_{n-1}$ 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
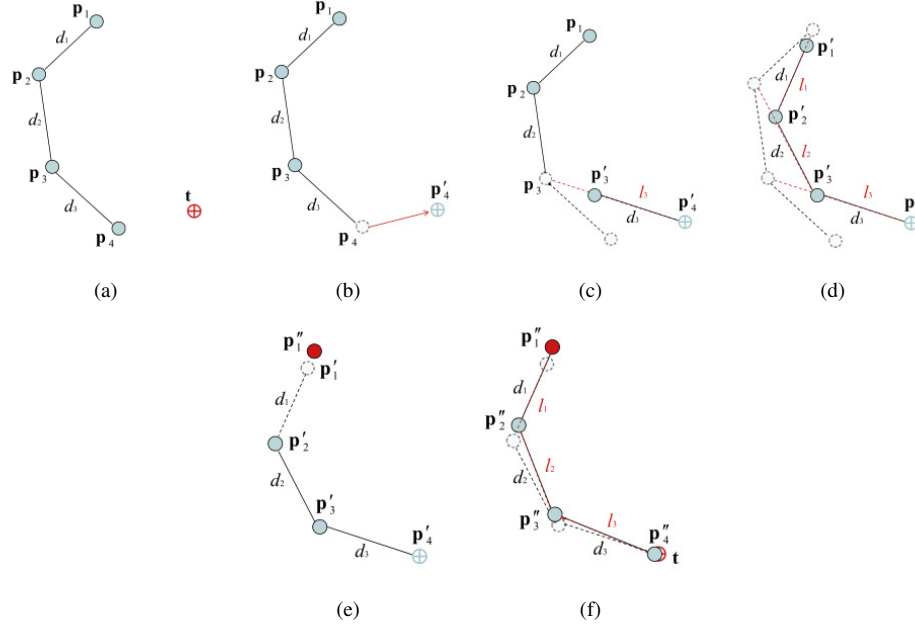
13

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]

the algorithm can be modeled by

$$\mathbf{p}_i = (1 - \lambda_i)\mathbf{p}_{i+1} + \lambda_i\mathbf{p}_i, \tag{17}$$

where $\lambda_i$ is a coefficient which ensures that the linear interpolation preserves the distance between joint positions $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$. 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 iteration proceeds very similarly to the forwards pass, this time interpolating each joint $\mathbf{p}_i$ between its current position and the new position of $\mathbf{p}_{i-1}$. The backwards pass is then

$$\mathbf{p}_i = (1 - \lambda_i)\mathbf{p}_{i-1} + \lambda_i\mathbf{p}_i. \tag{18}$$

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 [27].

# 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 Make-Human 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 are the options it provides regarding the complexity of the topology of the model's mesh, and the choice of skeleton rig. One of the rig presets, which is shown in Fig. 3.1, is specifically designed to be used for video games. Many additional cosmetic choices are also offered by the program, allowing for a full customization of the character. The model is automatically rigged and ready to be exported and used in an animation software.
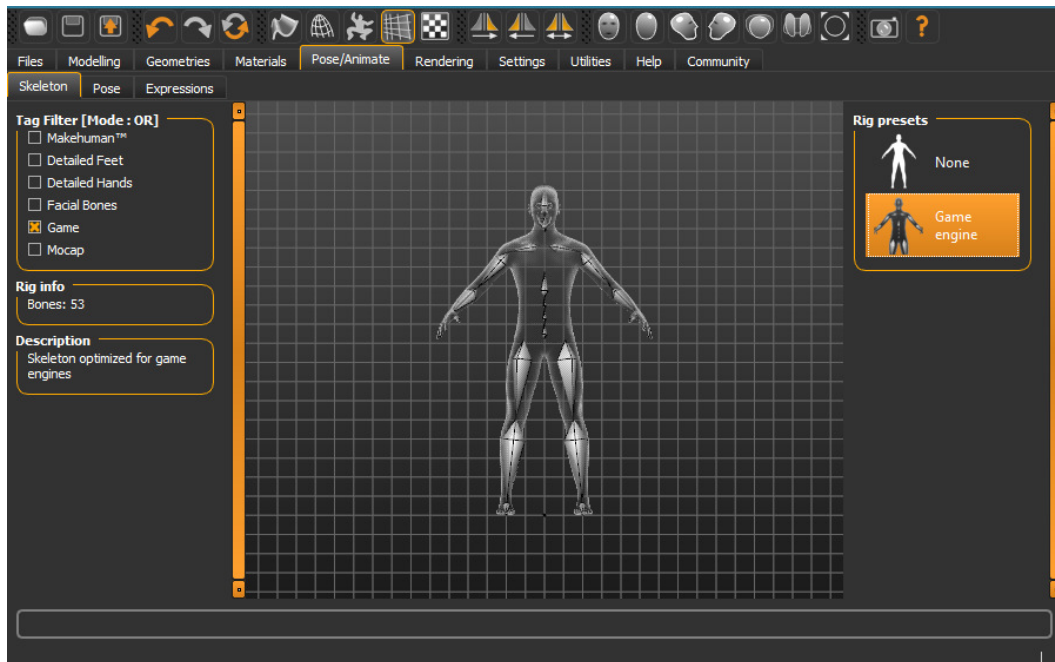


Figure 3.1: MakeHuman rig selection

## 3.2 Blender

The tool 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 in order to define how much the position of each vertex depends on the given bone position.
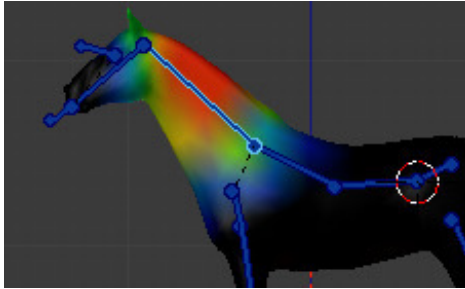


Figure 3.2: Weight painting in blender. Source: [24]

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 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 this work is meant to specifically focus on the usage of inverse kinematics in said game engine. Nevertheless, it is a good selection for this use case due to its advanced 3D support, the built-in packages and functionalities which help with implementing the techniques discussed in this work, and
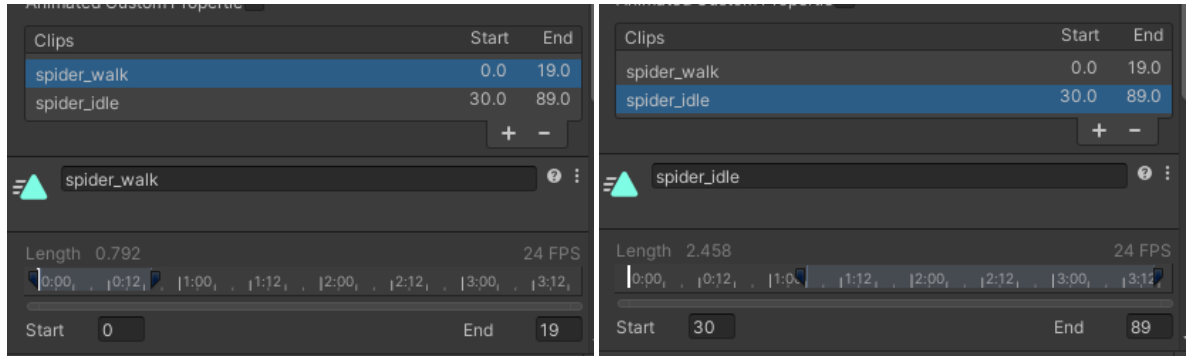
Figure 3.3: Animation clips extracted from a single animation during import

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 [18]. 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 as demonstrated in Fig. 3.3. 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 as shown in Fig. 3.4. Animations must be added to an *Animator controller* in order to be used by a Unity *GameObject* [17]. States can also be controlled procedurally from scripts
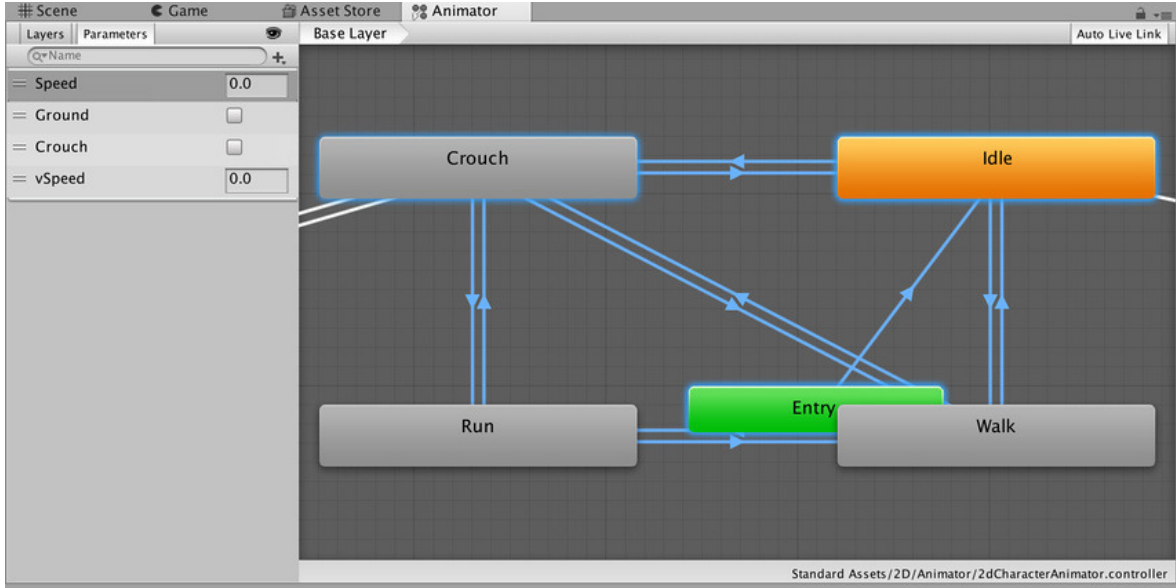
18

Figure 3.4: Animator controller animation state machine [17]

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 [22]. 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 [21]. 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 from one humanoid model to another [20]. 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 [22, 20]. Because these methods are not 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, which are briefly described in Table 3.1. 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.

| | |
|---|---|
| Blend Constraint | Allows the constrained object to blend between the position and rotation of two other objects. |
| Chain IK Constraint | Applies IK to a hierarchy of objects defined by a root and tip. Implements the FABRIK solver. |
| Damped Transform | Damps the position and rotation transform values from the source object to the constrained object in a hierarchy, increasing in time delay for each next object. |
| Multi-Aim Constraint | Rotates an object to face a chosen source object. Can have more than one source object with defined weighting. |
| Multi-Parent Constraint | Moves and rotates an object as if it is the child of another object in the hierarchy. Does not affect scale, and it can have more than one source object with defined weighting. |
| Multi-Position Constraint | Moves an object to follow a source object. Can have more than one source object with defined weighting. |
| Multi-Referential Constraint | Allows an object to act as the parent to multiple referenced objects. |
| Multi-Rotation Constraint | Rotates an object to match the rotation of its source object. Can have more than one source object with defined weighting. |
| Override Transform | Allows a constrained object's position and rotation to be overridden in world space, local space, or pivot space, by the displacement of the override transform, or by precise numerical values. |
| Twist Chain Constraint | Allows controlling world rotations on both ends of an object chain hierarchy. The two rotations are interpolated along the chain to create a smooth animated hierarchy. |
| Twist Correction | Applies a defined percentage of the rotation of a source object to chosen twist node objects. |
| Two Bone IK Constraint | Applies IK to a hierarchy composed of 2 bones which allows the use of a one target and one hint object for control. |

Table 3.1: An available set of predefined constraints from the Animation Rigging package which can be applied to a rig [27].

# 4. Inverse Kinematics in the Unity Engine

The demo application written for the purpose of this work 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 [2]. For the purposes of this application, the basic algorithm is implemented without many additional constraints 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 Fig. 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 know 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.

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
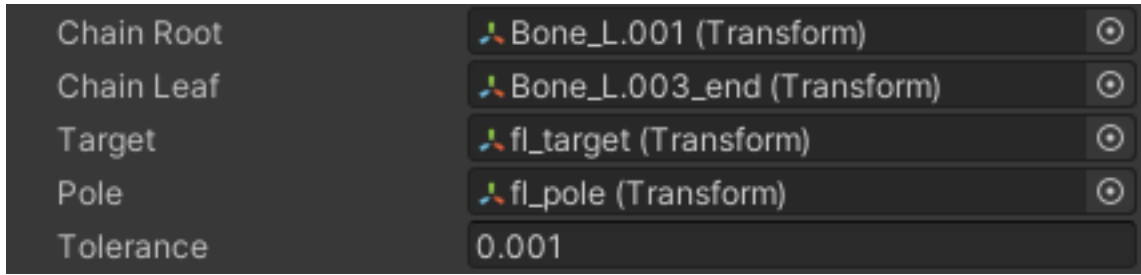
Figure 4.1: FABRIK script parameters

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, which can be seen later in Listing 4.3, is the use of square magnitudes when comparing distances to avoid the calculation of square roots, thus reducing the computational costs.

```
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 4.1: Target out of reach. *Fabrik.cs*

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

23

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 4.2: FABRIK forward reaching pass. *Fabrik.cs*

When the forward pass is complete, the root node is displaced from its original position. This is undesired, as the root node's 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 order. 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 cyclically until the end effector is within a threshold distance of the target. The FABRIK algorithm is heuristic, and as such it does not give an exact result. Instead, it aims to approximate the correct solution and solves the problem in a less complex way.

```csharp
void LateUpdate ()
{
    if (!enableIk)
        return;
    CopyTempPositions ();
    float distRootToTargetSqr = (
        target.position - chainRoot.position
    ).sqrMagnitude;
    if (distRootToTargetSqr > totalDistance * totalDistance)
    {
        StretchToTarget ();
    }
    else
    {
        var distEffectorToTargetSqr = (
            tempPositions [chainLen] - target.position
        ).sqrMagnitude;
        while (distEffectorToTargetSqr > tolerance * tolerance)
        {
            ForwardReachingPass ();
            BackwardReachingPass ();

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

        if (pole != null)
            BendToPole ();
    }
    ApplyTempPositions ();
}
```

Listing 4.3: Full execution of the FABRIK algorithm in each frame. *Fabrik.cs*

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 [10].

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 (Fig. 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 (Fig. 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 as demonstrated in Fig. 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.

## 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.

```
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];
    }
}
```
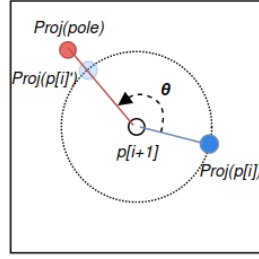
Listing 4.4: Pole target constraint. *Fabrik.cs*
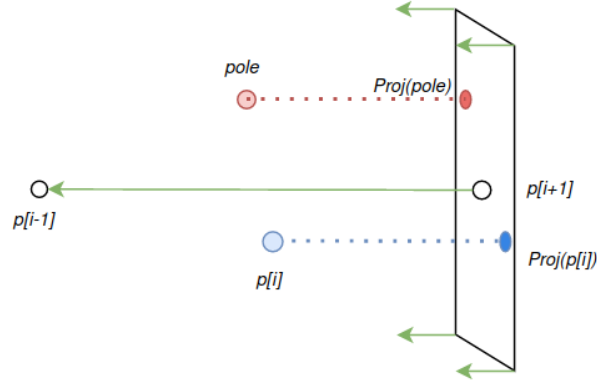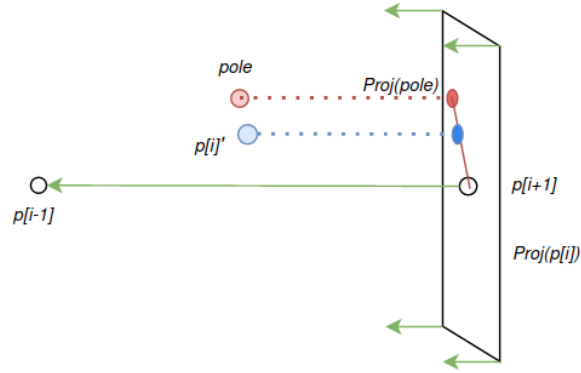
Figure 4.2: A diagram showing the concept behind the implemented pole target mechanism. (a) shows the initial setup where a plane is constructed with the vector from *p[i+1]* to *p[i-1]* as the normal. Both the current joint *p[i]* and the pole target are projected onto the plane. (b) is a view of the plane, showing the angle between the projections which dictates the new position of the joint *p[i]'*. (c) demonstrates the final position of the joint as a result of the pole target mechanism.

Figure 4.3: One raycast for each leg is pointed at the spider's local negative Y axis to scan the terrain.

## 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 as shown in 4.3. 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.

*Raycasts*

The raycast objects contain a script component which dispatches the rays and sets the appropriate hit target positions. First, a mask must be established, which will then be passed

29

```
void Update()
{
    if (Physics.Raycast(transform.position, -transform.up, out var hit,
                        100.0f, terrainLayerMask))
        HitTarget.position = hit.point;
}
```

Listing 4.5: Ray casts scanning for terrain. *RayLogic.cs*

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 hit 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 hit targets 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 Fig. 4.4).

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
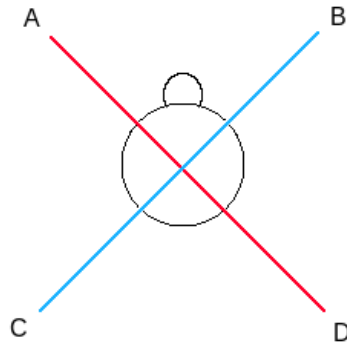
30

Figure 4.4: 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

*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 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.

## 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.

31

```
if (Grounded) // eligiblity checks
{
    _distToTarget =
        Vector3.Distance(transform.position, RayTarget.position);
    if (_distToTarget > DistanceThreshold && CanStep())
    {
        Grounded = false;
        _moveTo = RayTarget.position;
        _totalDist = _distToTarget;
    }
}
else // movement sequence
{
    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: The script's *Update* method which contains eligibility checks and execution logic for the leg target's movement sequence. *TargetLogic.cs*

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. *IKSpiderMovement.cs*

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.

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 uncon-

```
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. *IKSpiderMovement.cs*

strained 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 script's functionality relies on Unity's coroutines [19]. They enable a method to

```
private IEnumerator HandAnimation(string animation)
{
    _animator.Play(animation, 0, 0f);
    yield return new WaitForSeconds(0.2f);
    yield return new WaitUntil(
        () =>
            _animator.GetCurrentAnimatorStateInfo(0).normalizedTime
            >= 1.0f
    );
}
```

Listing 4.9: A coroutine which plays a designated animation and relinquishes control to the main loop once the animation is finished. *IKPressButton.cs*

pause its execution and then continue where it left off on the next frame. 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, presented in Listing 4.9, 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, shown in Listing 4.10. 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 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 (Listing 4.11. 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.

```
private IEnumerator PressButton(Vector3 buttonPos)
{
    yield return StartCoroutine(
        MoveToTarget(buttonPos, MOVEMENT_DURATION)
    );
    yield return StartCoroutine(
        MoveToTarget(origin.position, MOVEMENT_DURATION)
    );
}
private IEnumerator MoveToTarget(Vector3 destination, float duration)
{
    float time = 0;
    var startPos = target.position;
    while (time < duration)
    {
        target.position = Vector3.Lerp(
            startPos,
            destination,
            time / duration
        );
        time += Time.deltaTime;
        yield return null;
    }
}
```

Listing 4.10: The coroutine *PressButton* executes two *MoveToTarget* coroutines, moving the end effector's target to the specified button, and then back to the origin position. *IKPressButton.cs*

```csharp
private void Update()
{
    if (idle && Input.GetButtonDown("P_KEY"))
        StartCoroutine(PressButtons());
}
private IEnumerator PressButtons()
{
    idle = false;
    yield return StartCoroutine(HandAnimation("hand_lift"));
    _ik_component.enableIk = true;
    for (var i = 0; i < buttonSequence.Length; i++)
        yield return StartCoroutine(
            PressButton(buttons[buttonSequence[i]].position)
        );
    _ik_component.enableIk = false;
    yield return StartCoroutine(HandAnimation("hand_lower"));
    idle = true;
}
```

Listing 4.11: Reading input, and executing the main coroutine *PressButtons* which plays the full animation sequence. *IKPressButton.cs*

# 5. Experiments

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

## 5.1 Baked animations

In order to conduct the experiments, a second set of animations were prepared 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, as shown in Fig. 5.1. Once the model is imported into Unity, the animations can be broken up into their separate cases, as shown earlier in Chapter 3.3.1 (Fig. 3.3).
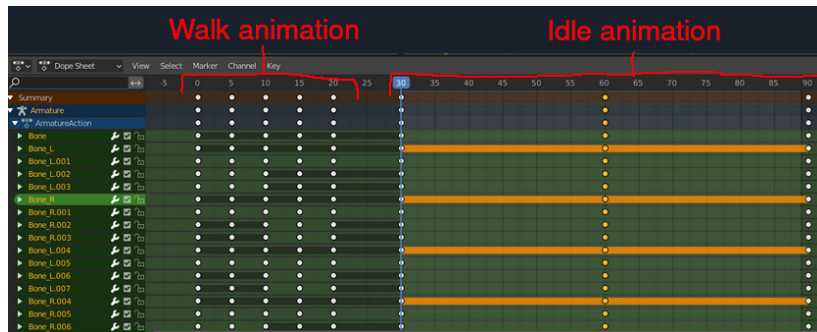


Figure 5.1: Two animations on a single timeline

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 (Fig. 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.
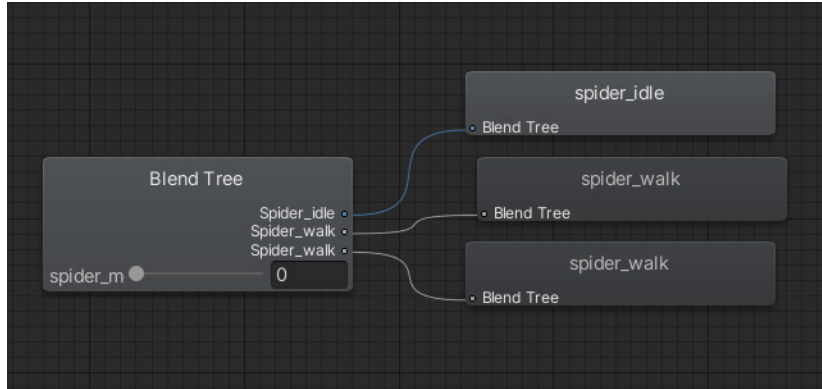
Figure 5.2: Blend tree containing animations for the spider

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 (Fig. 5.3).
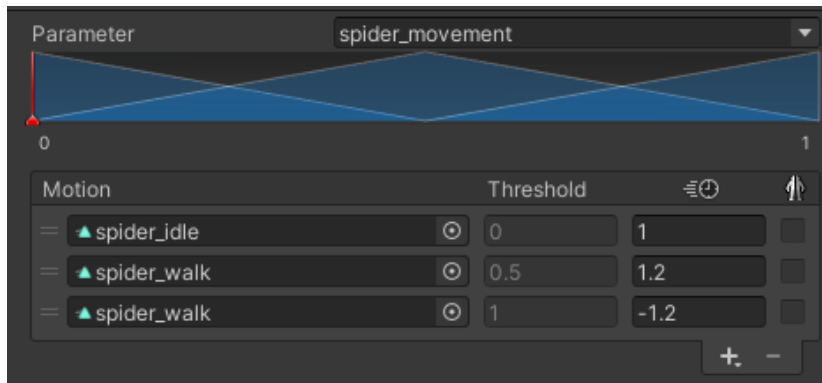


Figure 5.3: Configuration of the spider's blend tree which is dependent on the *spider_movement* variable

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 corresponds 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.

This version of the spider has its movement inspired by the spider from the game

```csharp
void SpiderMove()
{
    // Check grounded and gravity values
    (...)
    float verticalAxis = Input.GetAxis("Vertical");
    float horizontalAxis = Input.GetAxis("Horizontal");
    if (verticalAxis == 0f && horizontalAxis == 0f)
        _animator.SetFloat("spider_movement", 0.0f, 0.05f, Time.deltaTime);
    else
    {
        var speed = verticalAxis * spiderSpeed;
        //Wall climbing
        if (Physics.Raycast(transform.position, transform.forward,
                            out var hit, 1.1f, layerMask))
            spiderVelocity.y = speed * Time.deltaTime * 100.0f;
        else
            spiderVelocity.y = 0 < spiderVelocity.y ? 0 : spiderVelocity.y;
        transform.Rotate(
            0, horizontalAxis * rotateSpeed * Time.deltaTime, 0f
        );
        _controller.Move(speed * Time.deltaTime * transform.forward);
        _animator.SetFloat("spider_movement",
            verticalAxis >= 0.0f ? 0.5f : 1.0f, 0.1f, Time.deltaTime);
    }
    //Applying Gravity
    if (spiderVelocity.y < 0.0f)
        spiderVelocity.y += gravity * Time.deltaTime;
    _controller.Move(spiderVelocity * Time.deltaTime);
}
```

Listing 5.1: Moving the baked version of the spider, and playing the appropriate animations by setting the *spider_movement* variable using the *_animator.SetFloat()* method. *BakedSpiderMovement.cs*

Minecraft [25], 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 (Fig. 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.

| Clips | Start | End |
| --- | --- | --- |
| button_press | 0.0 | 39.0 |
| hand_lift | 0.0 | 13.0 |
| hand_lower | 26.0 | 39.0 |
| button_hit | 14.0 | 25.0 |

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

43

```
IEnumerator PressButtons()
{
    idle = false;
    yield return StartCoroutine(HandAnimation("hand_lift"));


    for (int i = 0; i < BUTTON_PRESS_COUNT; i++)
        yield return StartCoroutine(HandAnimation("button_hit"));


    yield return StartCoroutine(HandAnimation("hand_lower"));
    idle = true;
}
```

Listing 5.2: The main coroutine which plays the full animation sequence for the baked version of the human character. The *HandAnimation* coroutine is analogous to the one used for the IK character in Listing 4.9. *PressButton.cs*


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.

44

## 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.

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 work.

Starting with the idle position, both models look quite natural standing on a flat surface, as seen in Fig. 5.5a. However, this changes when the surface is uneven. The IK acting on the legs of the spider on the right in Fig. 5.5b 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 (Fig. 5.5c). This kind of rotation could have been implemented, however, the transition between floor and wall would be very unnatural either way and so an implementation inspired by the spiders in the game Minecraft was used instead.

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,
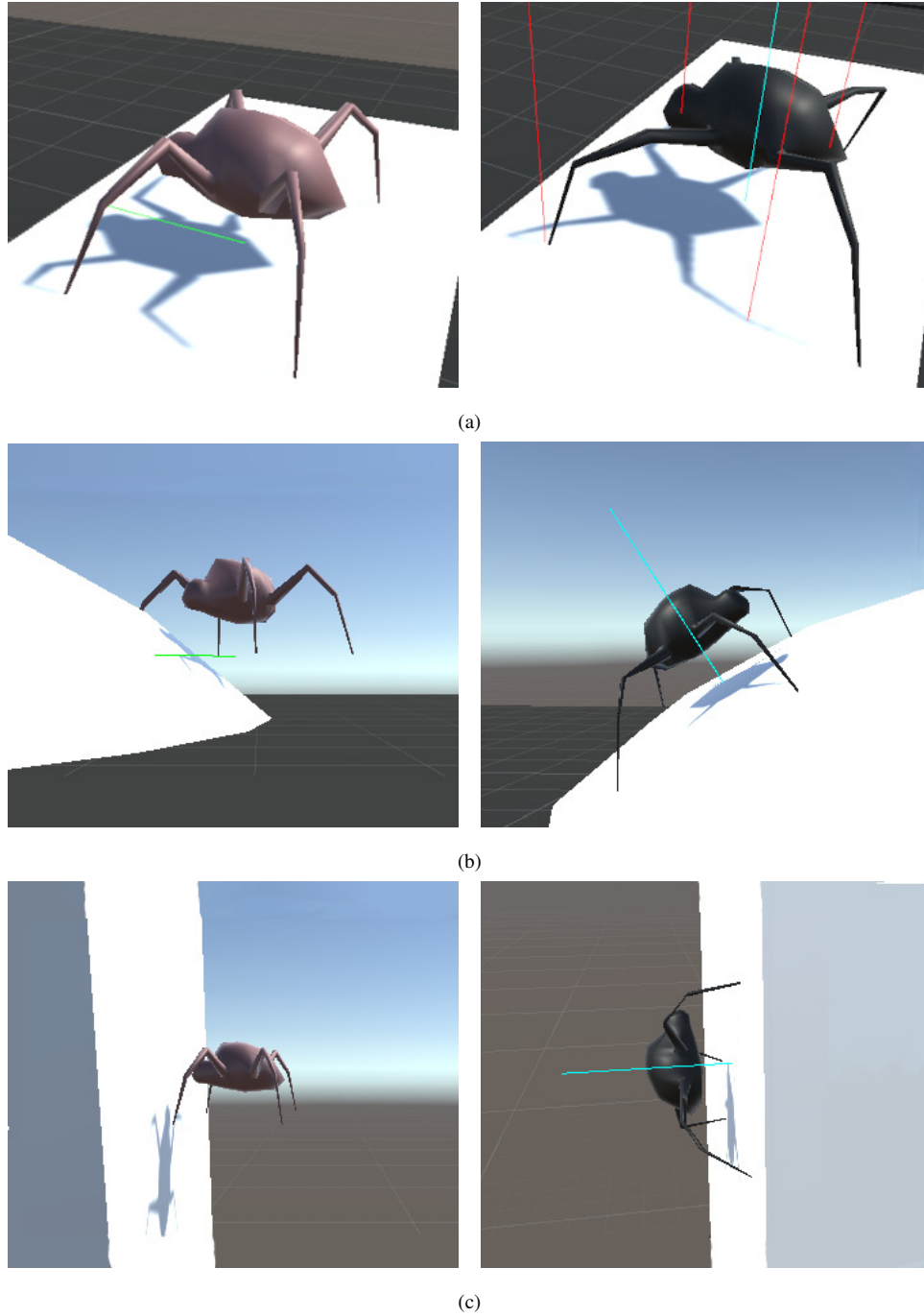
Figure 5.5: A comparison between the baked animation model (left) and IK model (right) of the spider.

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.

In certain games, the accuracy of animation sequences creates a greater immersion for

the player. In the event that the sequence of pressed 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 actions 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 as shown in Fig. 5.6a.

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, the baked animation has no knowledge of the characters surroundings, and misses its mark (Fig. 5.6b). 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. In the 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.

Comparing this to the IK animation, as seen in Fig. 5.7, 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.

47
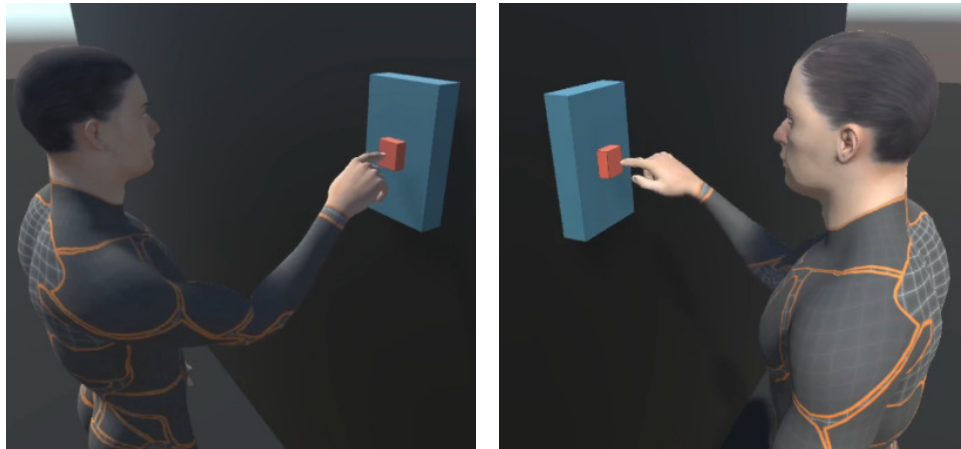
Figure 5.6: A comparison between the baked animation model (left) and IK model (right) of the human character. (a) shows both models executing the button press animation while being positioned and oriented correctly. (b) shows the models being offset and rotated from the default positioning.

Figure 5.7: 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 [23]. 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.

Starting with the baked version of the spider, a hundred duplicates spawned simultaneously produced the result seen in Fig. 5.8a. Similarly to the IK version presented further down, most of the CPU usage comes from the "scripts" usage category. A clear distinction
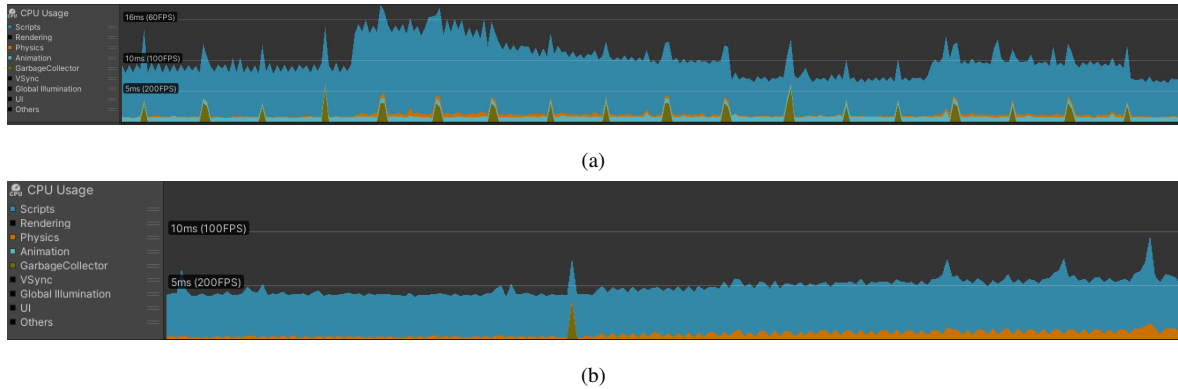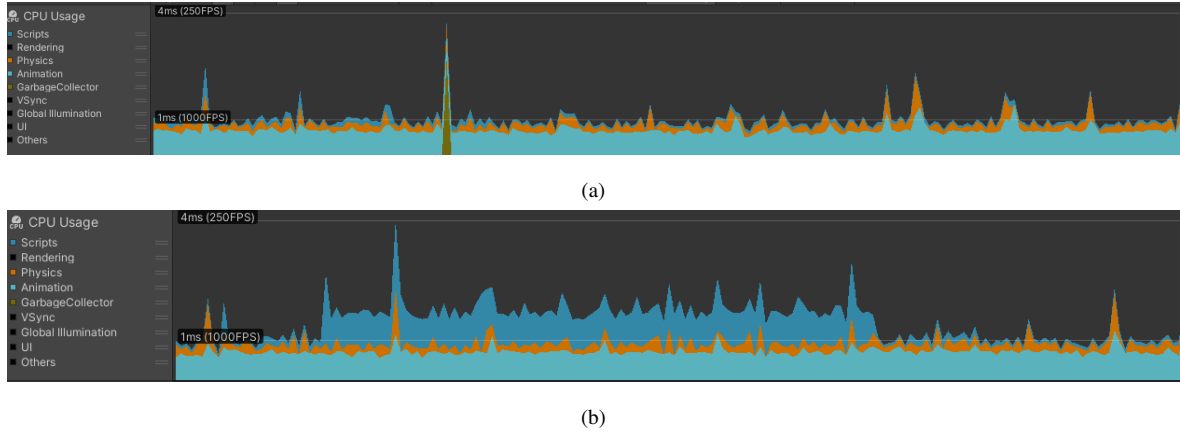
Figure 5.8: A comparison between the CPU usage profiling of one hundred instances of the spider baked animations (a) and procedural IK animations (b).

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.

The IK version of the spider has a much more steady average regardless of its moving or idle state. In Fig. 5.8b, 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.

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

(a)



(b)

Figure 5.9: A comparison between the CPU usage profiling of one hundred instances of the human character with baked animations (a) and procedural IK animations (b).

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.

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. Fig. 5.9a 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

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 Fig. 5.9b 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.

# 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 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] Andreas Aristidou, Joan Lasenby, Yiorgos Chrysanthou, and Ariel Shamir. Inverse kinematics techniques in computer graphics: A survey. *Comput. Graph. Forum*, 37(6):35–58, September 2018.

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

[5] 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.

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

[7] Samuel Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Transactions in Robotics and Automation*, 17, 05 2004.

[8] Samuel R. Buss and Jin-Su Kim. Selectively damped least squares for inverse kinematics. *Journal of Graphics Tools*, 10(3):37–49, 2005.

[9] 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.

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

[11] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, USA, 1996.

[12] Alec Jacobson, Zhigang Deng, Ladislav Kavan, and JP Lewis. Skinning: Real-time shape deformation. In *ACM SIGGRAPH 2014 Courses*, 2014.

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

[14] 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.

[15] A.A. Maciejewski. Dealing with the ill-conditioned equations of motion for articulated figures. *IEEE Computer Graphics and Applications*, 10(3):63–71, 1990.

[16] 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.

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

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

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

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

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

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

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

[24] Unity User Manual. Sculpting & painting - weight paint. https://docs.blender.org/manual/en/latest/sculpt_paint/weight_paint/introduction.html, 2023. Accessed: 2023-01-18.

[25] Mojang. Minecraft. https://www.minecraft.net/en-us, 2022. Accessed: 2023-01-02.

[26] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, 1999.

[27] 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.

[28] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge University Press, USA, 1992.

[29] 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.

[30] Charles W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 16:93–101, 1986.

[31] Xiaoqi Wang, Xing Liu, Lerui Chen, and Heyu Hu. Deep-learning damped least squares method for inverse kinematics of redundant robots. *Measurement*, 171:108821, 2021.

[32] 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

# List of Tables