Abstract

The graphical fidelity obtained in computer video games is increasing as the power of computer hardware increases. When two objects are required to interact, such as a bat and a ball, the increased graphical fidelity of modern games means that the simulated interaction must be of a suitably high quality. Motion-capture is often used to produce realistic animations of humans interacting with objects, but each recording is only suitable for the specific placement of that object. In an interactive environment, such as a game, it is desirable for the character to be able to interact with the object even when the object is in a novel location. This requires that the source animation is adapted in real-time to suit the object location.

This project provides a solution to this problem in the example of a cricket batsman hitting a ball by adapting motion-capture animations with the use inverse kinematics (pp. 27-47). The solution is adjusted and optimised to ensure natural-looking new animations are generated (pp. 35-38) and that natural variation can be incorporated in the solutions (pp. 38-40).

Secondly, producing motion-capture animations is an expensive technique; so reducing the number of actions needing to be captured is advantageous. Being able to synthesise new animations based on motion-capture animations reduces this need, and a technique is developed that allows variations of a batting animation to be generated without extra motion-capture sessions or manual adjustment. A method of generating these animation variations is developed (pp. 50-58). Making alterations to the animations is found to be both predictable and interactive (pp. 54-58).

Table of Contents

1 Introduction and Background

1.1 Motivation

Creating realistic animations for use in games is often achieved by using motion-capture data. This is especially true when the animations are complex, with a great deal of rapid, time-critical movements. Cricket batting strokes are a good example of this. Whilst motion-capture data for cricket strokes will provide a high degree of realism in the movement, there are inherent problems with it. The first problem is that a recorded shot animation is only applicable to a particular ball position (or a very small range of ball positions). In order to counteract this problem, one can simply record more and more animations to deal with the ball in slightly differing locations until sufficient coverage has been achieved. However, this approach exacerbates another problem with using, as the motion-capture session will need to be longer, and therefore will be more expensive. Therefore, a solution that minimises the amount of motion-capture data necessary yet providing sufficient 'shot coverage' is desirable.

With respect to animations being used in computer games, this is a relatively recent problem. With early 3D games, the problem of getting the bat to precisely hit the ball could be largely ignored (Figure 1.1a), as the graphics were at a sufficiently low resolution that the user would not notice the lack of realistic interaction. Also, the levels of precision and realism in other aspects of the graphics and animations were of an equally low level. As computer hardware became more powerful, there was an increased need for precision in animation, as the higher-resolution graphics made large errors noticeable. The solutions employed were often simplistic. Using the example of cricket, the bat could be made to intercept the ball by moving the entire batsman character in the scene (Figure 1.1b), or by interpolating the ball's motion so that it meets the bat (Figure 1.1c). Also, a combination of the two could be used. This is, of course, not physically plausible, but for the time, it was an acceptable solution, and implementable on the hardware of the time.

With modern computer hardware, the graphics are of a sufficiently high quality that these kinds of approaches become noticeable. Although these methods can ensure the bat and ball intercept, there are likely to be numerous animation artefacts which can break the immersion, such as the character sliding to meet the ball, or the ball bending. These artefacts are far more noticeable when the graphics resolution is increased, and are also more likely to be noticed if the rest of the multimedia experience is of a higher quality – if the textures, sounds and animations are low quality, games developers can 'get away' with some things. As soon as the overall quality is improved, smaller details such as the aforementioned artefacts are more likely to be noticed.

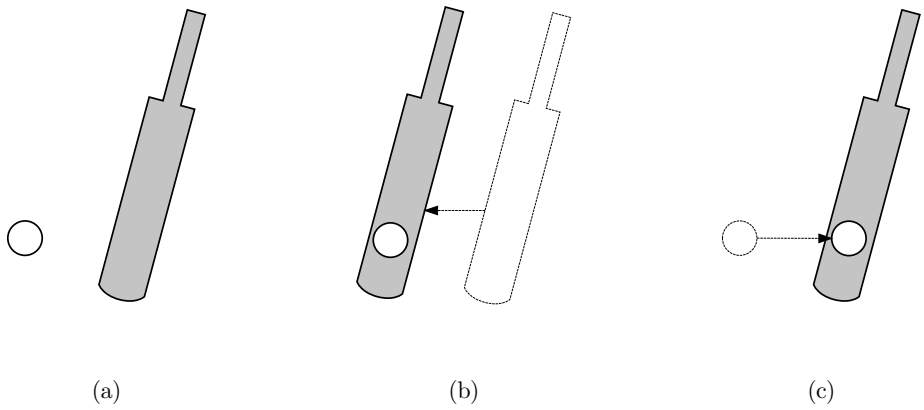

Figure 1.1 – Methods of making a bat intercept a ball (adapted from Watt & Policarpo, p463)

When considering sports games in particular, these problems can be magnified by the use of slow-motion action replays. What previously might have occurred in the blink of an eye can be slowed to the point where the smallest animation discrepancies can be seen. This means that even more accurate object interaction is required in order to maintain the illusion of realism. Although the increase in computer hardware power means that these errors may be more likely to be spotted, it also provides an opportunity to employ more complex algorithms to solve the problems it creates.

Secondly, in order to create an ever more convincing depiction of real-life in a computer game, characters are equipped with ever more animations that they can perform when required. Taking the example of cricket, an old game might have had only one animation for the batsmen – a simple 'hit' animation – due to memory limitations. A modern game may have hundreds of shot animations, with several variations of each shot. Creating the animations becomes the limitation, rather than the previous memory limitation. For this reason, a solution which allows shot variations to be produced easily is desirable.

1.2 Project Overview

The project aims to develop a system that enables novel 3D character animations to be generated from source animation data. In particular, the project will deal with the sport of cricket and animating a batsman character. There are two distinct areas of work involved in this dissertation. The first aim is to provide a solution to the problem of making a batsman's bat make contact with the ball when there is no existing animation that would allow the bat the intercept the ball (hereafter referred to as the hit-point problem). The second aim is to devise a method of generating new animations from a limited set of source animation data (referred to as shot synthesis). An example of this

might be to create a more extravagant version of a particular shot, or a more muted version.

Although the project focuses on cricket, the solution developed will be general enough to be able to be applied to other actions such as tennis and baseball hits, and even actions not requiring a bat or racquet, such as the act of catching a ball. In this document, cricket and its related terms are used, though the concepts could equally well apply to other fields.

In this report, the research conducted into the field of 3D character animation is detailed. Initially, the research focussed on the various techniques that can be used to animate characters, such as morph target animation or skeletal animation. Skeletal animation turned out to be a clear choice for starting point of the project, so different ways of being able to force a bone to reach a specific target were investigated.

After this investigation, it was decided some implementation of inverse kinematics (IK) combined with forward kinematics (FK) would most likely enable the project aims to be achieved. The research was then focussed on different IK implementations which have been used to adapt existing animation to fit new scenarios.

Throughout this report, "games" is used as a synonym for "computer/video games", as well as other real-time applications of computer graphics.

1.3 Project Aims

This project aims to produce a computationally cheap and efficient method of adjusting motion-capture data to create new animations from existing ones. More specifically, the project concentrates on the animation of human characters performing 'hit' actions, such as a player hitting a tennis ball, a batter hitting a baseball, or a batsman hitting a cricket ball, in order to use them in interactive applications, such as computer games.

The system developed will attempt to adjust motion-capture animations to allow the character to hit a variable point in space, as opposed to the single point defined by the motion-capture data. The project will focus on cricket batting, as it provides a good example of a scenario where the ball arrives at a given point and the batsman has to react to hit it. This provides the primary aim of the project – the character being able to hit a ball at a given point in space and time, whilst retaining the characteristics of the original animation.

The secondary aim of the project is to devise a method of synthesising new shot animations based on source animations. The aim here is not to create animations that reach specific hit points, but rather to create animations that are slight deviations from the source shot – such as a harder or softer shot, or a more exaggerated version of a shot. The motivation for this is that the number of source animations which need to be motion-captured is reduced – a smaller selection can be recorded, with variations being generated without the need for more motion-capture sessions.

As this project is tended towards the production of animations to be used in computer games, the focus will be on a system that could be optimised to work in real-time. However, actually getting the system to operate in real-time is a subsidiary aim of the project.

1.4 Background

In this section, an overview of various 3D animation techniques is given, as they will be referred to extensively later in the document. There is an assumed knowledge of certain 3D graphics and animation concepts, such as 3D geometry and keyframing.

1.4.1 Character animation

Animating anything in 3D computer programs is a case of changing its position as time elapses. The points that are animated depend on the animation method used, as well as the type of object being animated. For example, to animate a ball moving through a scene, the position of the ball is changed, and thus the ball's geometry is moved. When animating a more complex object, which actually changes its shape over time – such as a human – the description of the motion is much more complex. For example, if a character moves his hand whilst the rest of his body remains still, the geometry making up the hand, arm and shoulder must be animated to look like a natural motion. In essence, to perform an animation of a character, the position of each vertex of that character's model must be described at certain time intervals. The way in which these vertex locations are defined depends on the method used to animate the character.

There are two main methods used to animate characters in games: vertex animation (also called 'morphing', or 'morph-target animation'); and skeletal animation. Vertex animation is the simplest method of describing 3D animation in that for every keyframe, the position of every vertex in the geometry is stored. The advantages of this method are its simplicity, and also that it can provide very high levels of detail such as skin and

muscle deformation. However, there are several important drawbacks to this approach. Firstly, animations stored like this use a lot of memory due to storing every vertex position at frequent time intervals, and the memory requirement increases with the geometry complexity. Secondly, there is an issue with deformation of the geometry when interpolating vertex positions between keyframes. A simplified example is of this is shown in Figure 1.2.

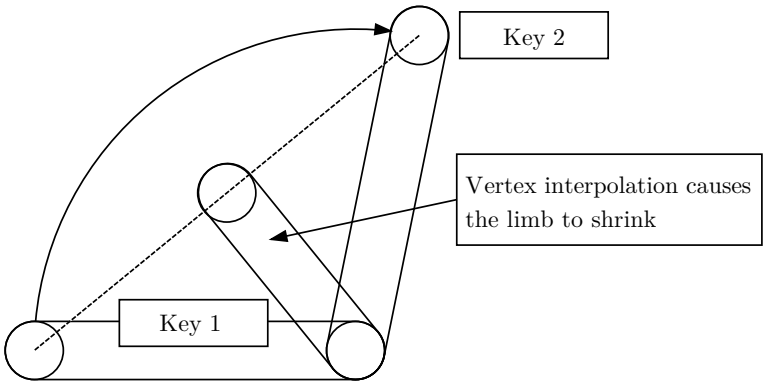

Figure 1.2 – Distortion due to vertex interpolation (Watt & Policarpo, 2003)

Another undesirable property of this kind of animation is that the geometry and animation are intrinsically linked, meaning that the same actions cannot be applied to a different character mesh. Finally, making adjustments to animations stored as morph-targets is very difficult to do procedurally, as whole clusters of vertices must be dealt with to, say, move a character's hand slightly. All of these downfalls are dealt with by using skeletal animation.

With skeletal animation, instead of defining the positions of every vertex at every keyframe, instead the geometry is driven by an articulated structure, called the skeleton. A skeleton is a defined as a hierarchical structure of bones, beginning at some arbitrary root (usually the hips, as this is close to the character's centre of gravity). The character mesh is then attached to the skeleton in a process called 'skinning', so that manipulating the skeleton results in the geometry changing appropriately. The animation is no longer coupled to the geometry as with vertex animation, and therefore the complexity of the animation data is greatly reduced: instead of specifying the positions of thousands of vertices, skeletal animation stores the rotations of the joints, which typically number in the tens, rather than the thousands.

Of course, there is extra computational complexity in attaching the geometry to the skeleton, but that is an off-line process, meaning that to procedurally animate the skeleton, these skin bindings are left alone. This also means that one animation can be used to drive multiple different geometries.

There is one final – and very important – aspect of skeletal animation that makes it highly desirable, and that the skeleton animations can be derived from a process called motion-capture, which is commonly used for creating animations for use in games.

1.4.2 Motion-capture animation

Motion-capture is a system whereby an actor is filmed performing an action with markers placed on certain points on his body. The positions of these markers in 3D space are calculated, and processed into joint rotation angles so that they can be applied to a character skeleton in order to animate it. The main advantage of using motion-capture data is that it captures human motion relatively quickly in a more realistic manner than is possible by traditional keyframing methods. There are, however, two main problems with using motion-capture data. Firstly, producing motion-capture data is expensive, as it requires expensive equipment and the hiring of an actor of sportsperson (in the case of recording sporting actions). Secondly, each animation generated is only valid for one specific circumstance, which Watt & Policarpo (2003) describe as "an inherently limiting process" adding that it would be desirable to be able to alter them to produce "new sequences from existing material."

In the case of cricket batting animations, the motion-capture session would involve the performer executing a wide variety of cricket shots with the ball in various locations. (In fact, there is not really any need for the ball to be present during the motion-capture, as the performer can simply pretend the ball is in a certain location) The motion-capture data is then processed to convert the marker locations into a skeleton and joint angles which vary over time, which is called a forward kinematics animation.

1.4.3 Forward Kinematics and Inverse Kinematics

When discussing skeletal animation, forward kinematics (FK) refers to the technique of animating a skeleton by explicitly specifying the joint angles for each joint in the structure at several points in time. More specifically, FK implies that the position and orientation of the end effector (the end of the last bone in a joint chain – see Figure 1.3) is calculated from the joint angles and bone lengths of all the joints in a joint chain. Motion-capture data is an example of animation defined via FK.

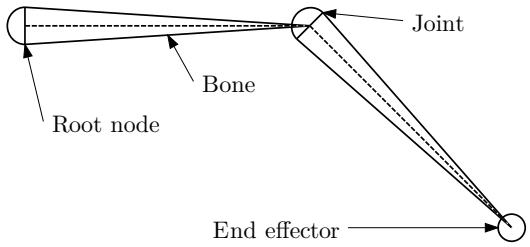

Figure 1.3 – A simple joint chain

Inverse kinematics is essentially the reverse of FK, in that instead of calculating the position of the end effector by specifying joint angles, the position and orientation of the end effector is specified, and the joint angles required to meet these criteria are calculated. The process of calculating the necessary joint angles is referred to as 'IK solving'. When solving IK systems, there are essentially two different methods of solution available: analytic and numeric (Ho, Komura, & Lau, 2005). Analytical solutions provide a single solution to the system and are very fast to compute, and therefore are ideal for real-time computing. However, the more complex a joint chain, the harder it becomes to solve it analytically (Watt & Policarpo, 2003). Numerical solutions for IK chains involve iterating towards a solution. These are more computationally-expensive than analytical methods, but allow solutions to more complex systems. When dealing with systems that have more than six DOFs, there are an infinite number of solutions to the problem, which leaves the solver with the task of choosing the 'best' solution.

This is where the notion of choosing solutions that provide 'realistic-looking' results arises, and forms the basis of this project.

1.5 Generating new motions from existing motion data

There are several different approaches used to create new motions from an existing library of motion data. Which approach is used depends on aspects such as how the new motions will relate to existing motions, the size of the motion data library, and whether the processing is to be off-line or on-line.

One of the most popular approaches is to use motion graphs. With this method, each motion is spliced up in some way, and each segment is represented as a node in the graph. These nodes are connected by directed edges, which represent motion sequences that can plausibly follow on from each other. In order to generate a new motion which takes the character from pose A to pose B, the graph is searched for the 'best' route

through it, and this route represents the new motion sequence. The 'best' route through a graph is dependent on the specific implementation. For example, Arikan & Forsyth (2002) assign each of the directed edges a 'cost' value, and create a set of 'costed' constraints for the motion. Valid routes through the graph are tested and scored in order to choose the best resulting animation. The results are generated in real-time, and are described by Arikan & Forsyth as "natural looking". However, the motions generated can only be routed to and through poses that were in the original dataset. This means that without a very large dataset, this approach would not be able to solve the hit-point problem. As reducing the size of the data set required is part of the problem, this is not a viable solution.

Another method for generating novel motions from existing data is to use IK. IK is generally used when there is some particular goal to be reached, such as a character's hand reaching for an object. Simply making a specific end effector reach a target position and orientation is a solved problem in IK. When it comes to using IK for character animation, however, the problem arises that IK systems are underdetermined – although there are many (usually infinite) poses which will allow the end effector to reach the target position, some are more likely for a given scenario than others (Grochow, Martin, & Hertzmann, 2004).

Given that IK produces poses that ensure a limb reaches a specific position, it seems clear that IK is a very good candidate to help achieve the aims of this project. Thus, this dissertation now details the current state-of-the-art with regards to solving IK systems, especially with respect to solving for human motion.

1.6 Solving IK Analytically

1.6.1 A simple example

One of the simplest examples of solving an IK system analytically, is a two-bone structure where the two bones are joined by a 1-DOF joint, and the root joint is on a 1-DOF joint. If the two bones are of length L_1 and L_2 and their joint angles are θ_1 and θ_2, the end of the first (x_1, y_1) bone can be written as:

$$x_1 = L_1 \cos\theta_1$$
$$y_1 = L_1 \sin\theta_1$$

Equation 1.1

The end of the second bone, and therefore the end effector in the system can be written as:

$$x = L_1\cos\theta_1 + L_2\cos(\theta_1 + \theta_2)$$
$$y = L_1\sin\theta_1 + L_2\sin(\theta_1 + \theta_2)$$

Equation 1.2

This is the forward kinematic description of the system, and is shown in Figure 1.4.

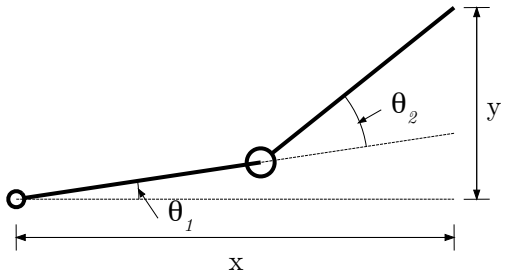

Figure 1.4 – A simple joint chain marked up for an analytical solution

By using trigonometric identities and rearranging , the joint angles required for the system to reach a goal at location (x, y) are calculated with:

$$\theta_2 = \cos^{-1}\left(\frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}\right)$$
$$\theta_1 = \tan^{-1}\left(\frac{-L_2\sin\theta_2 x + (L_1 + L_2\cos\theta_2)y}{L_2\sin\theta_2 y + (L_1 + L_2\cos\theta_2)x}\right)$$

Equation 1.3 - from Watt & Policarpo (2003)

Considering that this is only solving a 2-DOF system, it can be imagined that finding a closed-form solution for higher DOFs will be significantly more difficult. Aside from this, there is also a problem with these methods in that if the goal cannot be reached (i.e. the distance to the goal is more than the combined length of the bones in the system) then a

solution is not even attempted, which significantly impedes the usefulness of this method in practice.

Watt & Policarpo state that closed-form solutions are only possible when the system's DOF is less than six. As the human arm is 7-DOF, and this project is concerned with solving an IK system for the human arm, these methods seem unlikely to be useful for the purpose of this project. However, the primary project aim might be simplified by constraining the shoulder in some way, so that the DOF of the system is reduced to six or less. Whether or not acceptable results are possible by doing this would require experimentation, although it is less likely that the secondary aim could be achieved by restricting the shoulder motion in some way.

At this point it is worth mentioning that for the proposed animation system to work, both arms must have their motion adjusted, so that both hands remain on the bat handle. Rather than this being a 14-DOF system, it can be split into two separate systems of 7-DOF that are solved independently (Watt & Policarpo, 2003; Choi & Ko, 2000).

1.6.2 Solving for human limbs

In their paper *Real-time inverse kinematic techniques for anthropomorphic limbs*, Tolani, Goswami & Badler (2000) introduce an analytical method for solving IK chains representing a *human-arm-like* (HAL) chain. The chains solved satisfy a 6-DOF constraint for the end effector – 3-DOF for its position, and 3-DOF for its orientation. A HAL chain is 7-DOF, which means that there is one redundant degree of freedom in the system. In order to solve analytically, the redundant degree of freedom must be eliminated, and the system proposed by the authors uses this redundancy to allow an extra constraint, either on joint limits, or to constrain the elbow position to be as close to a given position as possible.

As mentioned in the previous section, analytical solutions tend to break down when the end effector's goal is not reachable, and the system developed by Tolani et al. is no different. In order to solve for a system where the goal is unreachable, the system has to fall back to a constrained optimisation method, but the problem is simplified, resulting in a faster solution. They show that solving for a reachable goal specifying position and orientation, their method is at least 20 times faster than numerical methods. Also, the errors in calculating the position and orientation are orders of magnitude less (in fact, the only error in the analytical solution appears to be due to floating point number precision) and a solution is always found, whereas numerical methods are shown to fail in around 1% of cases. Even when the goal is unreachable and numerical methods are

necessary, the simplifications which can be made ensure that the solution is found at least 12 times quicker than standard numerical methods.

Aside from the apparent speed and accuracy of the solution, there are a number of other features that seem to make this a more desirable solution than numerical methods. One advantage is in the way the analytical approach handles singularities in the system. A singularity occurs when the limb is fully outstretched, and this will tend to cause problems with numerical solutions, manifesting itself as joint oscillations, causing further goals to become unreachable, or resulting in termination of the algorithm. In 3D animation software that can solve IK systems (such as *Maya*), this is why it is normally necessary to ensure that all joints in a skeleton begin with a non-zero joint angle.

Another advantage of the solution developed is repeatability. With numerical methods, the calculated pose is dependent on the starting pose, resulting in different solutions for the same goal position, depending on the previous limb configuration. This is demonstrated in the following figures, taken from Tolani et al. (Tolani, Goswami, & Badler, 2000)

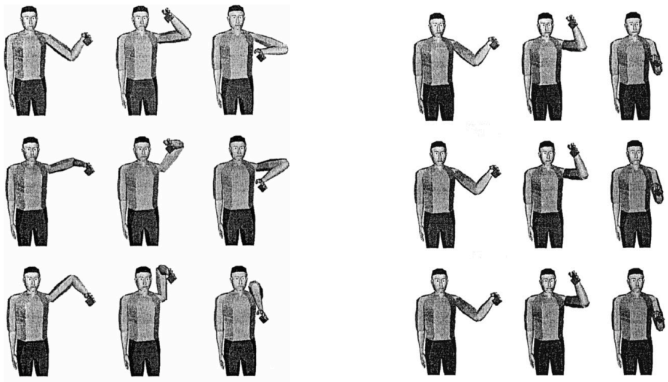

Figure 1.5 – The difference between a numerical solution (left) and an analytical solution (right) when performing a cyclical motion

The first set of stills shows how the limb positions quickly degrade and become unrealistic and non-repeatable using a numeric method, whilst the analytic method is repeatable, and more realistic for human arm motion. The situation in the numeric method example can be improved by imposing joint angle constraints to the structure, so the example is somewhat exaggerated, but the problem does still exist.

For the purposes of this project, there are two problems with the solution provided by Tolani et al. The first problem relates to the feature of the solution just mentioned –

that the solution provided does not depend on the initial pose of the limb. One of the key features of the project is that the resulting limb configuration *should* depend on the source configuration. That is, the calculated result should draw on information available in the forward kinematics data, in order that the characteristics of it are preserved in the generated pose. Additionally, the solution does not take into account any kind of biomechanics, and also does not attempt to ensure realistic joint trajectories are created. Instead, the onus is placed on the animator to ensure that the animations generated are realistic, which implies that a good deal of evaluation and adjustment is required to create useful results.

Another consideration that is not taken into account in this solution is the notion of joint stiffness and damping. Take the example of turning a doorknob. This can be achieved by just turning the wrist, whilst the shoulder remains still. This is because the wrist is essentially less stiff than the shoulder. If the stiffness of the wrist is increased so it becomes locked, a certain amount of rotation can be achieved by just using the shoulder joint. With regards to cricket shots, certain shots will have different stiffness in each of the joints, not through anatomical differences, but differences purely in the style of the motion. Therefore, whilst the approach of Tolani et al. may produce feasible results, they may not adhere to the style of the source motion.

One final problem is that limiting the pose adaptation to just the arms will not be sufficient for this application. In order to create more realistic-looking adjustments, it is likely that the changes in pose will extend further than just the arms. Consider the situation where the new hit position is nearer the character's body than the source animation. In this instance, a real person will likely move his entire upper body, rather than simply bringing his arms closer to his torso.

1.7 Solving IK Numerically

When IK systems reach a certain level of complexity, analytical methods cannot be used to solve them, and numerical methods must be employed. Some of the most prevalent methods are explained briefly here.

1.7.1 Cyclic-Coordinate Descent

Cyclic-Coordinate Descent (CCD) is an iterative method of solving IK chains by minimizing the difference between the end effector and the goal, one joint at a time, starting with the last joint in the chain and working up it. Dealing with the simple case of a 2D chain (i.e. one which contains several 1-DOF rotational joints, all aligned in the same plane), in each step, a rotation is applied to the joint such that the end effector is placed along the line segment between the root of the joint and the goal.

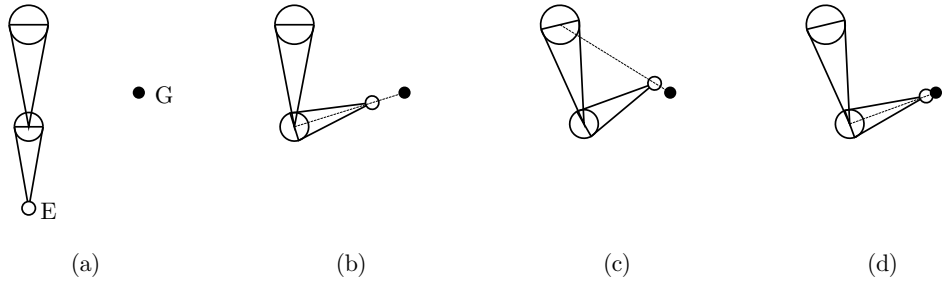

Figure 1.6 – The CCD algorithm

In Figure 1.6a, the initial state is shown, along with the end effector location, E, and the goal location, G. In Figure 1.6b, the last joint is rotated to make E lie on the line between the joint root and G (the dashed line). In Figure 1.6c, the first joint is rotated, again so that E lies on the line between its root and G. In Figure 1.6d, the process is repeated for the last joint, but this time, the joint is rotated in the opposite direction. This process is repeated until the end effector and the goal are within a predefined distance, or until a certain number of iterations have been attempted. This second caveat is required in case the goal in unattainable.

This method becomes more complex when dealing with three dimensions, and also becomes more complex if G specifies not only a position, but an orientation as well. The method scales well in that its complexity is not increased as the joint chain lengthens, and the time required for the algorithm to complete is linear with the number of joints in the chain. As the algorithm begins with the final joint in the chain and works up it, it is biased towards lower joints in the chain, meaning that the result may not always look natural when considering human limbs (Lander, 1998; Fêdor, 2003). To reduce this problem, it is easy to apply joint limits in CCD, so that a joint's rotation is clamped to be less than a certain value. The late-joint bias can also be reduced by introducing damping to the joints, by stating that a joint may only rotate a given amount in each iteration.

Given that this project requires an algorithm to go from a source pose to a destination pose which should be similar to the source, it is possible that CCD may provide a good method of achieving this. Again, experiments will need to be carried out to test the efficacy of this method for the project.

1.7.2 Jacobian Pseudoinverse

Using the pseudoinverse of a Jacobian matrix to solve IK systems is well-documented (Buss, 2004). It is a differential method of finding a solution, which means that small

steps are taken from the start position to the goal position. The algorithm begins by computing a Jacobian matrix, **J**, which relates the change in the joint angles in the system to the resulting change in the end effector position. This results in a $m \times n$ matrix where n is the number of DOF in the joint chain, and m is the number of DOF in the end effector. In order to compute the required joint angle changes to create a necessary end effector change, the inverse of **J** must be computed in the following steps:

Let:
e = current end effector position and orientation;
g = goal position and orientation;
Δe = change in end effector position;
Δθ = change in joint angles.

```
while (e - g) < some small error
        compute J for the current pose
        compute J^-1
        pick Δe so e + Δe is nearer g
        Δθ = J^-1 Δe (compute required change in joint angles)
        θ = θ + Δθ (apply changes to joint angles)
        compute e by using FK
```

However, as **J** is not a square matrix in the case of a human arm system (it is 7x6), it is not directly invertible, so instead the pseudoinverse is computed, written $\mathbf{J}^+$. The pseudoinverse is given by:

$$\mathbf{J}^+ = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T$$

Equation 1.4

As a system with a Jacobian of size 7x6 includes a redundant DOF, there are an infinite number of solutions to the system. In order to choose a specific, desirable result, instead of computing $\Delta\theta$ as $\mathbf{J}^+\Delta\mathbf{e}$, it is computed as $\mathbf{J}^+\Delta\mathbf{e} + (\mathbf{I} - \mathbf{J}^+\mathbf{J})\Delta\mathbf{Z}$, where **I** is the identity matrix of joint-space dimension, and $\Delta\mathbf{Z}$ is a secondary task (Watt & Policarpo, 2003). The secondary task is an extra calculated constraint, such as ensuring specific joint angles. This secondary task can also be made to simulate joint stiffness, by adding weightings to the different joints when calculating $\Delta\mathbf{Z}$.

The Jacobian pseudoinverse method does not provide a repeatable solution, as it depends on the initial pose of the limb. In order to provide repeatability, the new pose can be

calculated from a neutral, start pose each time, though this introduces the possibility of creating sharp transitions from frame-to-frame in an animation. However, should the initial pose be set to a pose from a frame of motion-capture data for each frame, the results are likely to be smooth. However, experimentation would need to be carried out to test this.

1.8 Importance-based IK

For a system which uses motion-capture data and solves IK joint chains, it is necessary to be able to control how much effect each different animation method has on the skeleton at any one time. Taking the example of cricket, at the point when the bat impacts the ball, the skeleton will need to be defined purely by the IK system, so that the positions of the limbs can be precisely defined. At the beginning of the motion-capture sequence, when the character is in a rest pose, the skeleton should be controlled entirely by motion-capture, as at this point the character does not know where the ball will be when he tries to hit it.

As a result, it is apparent that a number of heuristics need to be developed that defines the relationship between the current position in the animation and the influence of the IK and motion-capture limb configuration. At first, it seems plausible that blending towards fully-IK as the frame number approaches the hit point, and blending back to FK after this frame number would be an acceptable approach (shown in Figure 1.7).

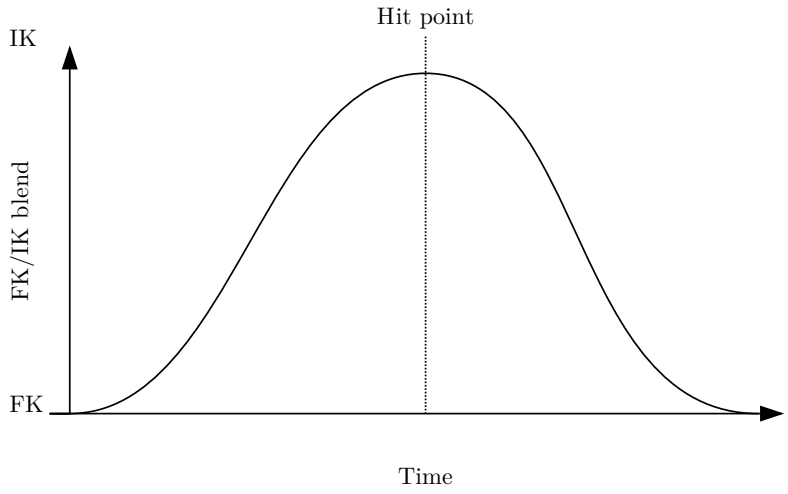

Figure 1.7 – FK/IK blending over time

There is a problem with this approach when considering shots where the bat remains near the hit point long after the actual impact, as might be the case is a gentle, defensive stroke. In this case, after the ball is hit, the skeleton would move back to the motion-capture pose, even though in the original motion-capture sequence, the bat should remain relatively still after the impact takes place. Therefore, this approach is not acceptable, and a new heuristic is required.

Shin et al. (2001) devise a set of heuristics to use when constraining a character's motion with IK. The subject of their research is to do with retargeting motion from one character to another when their skeleton proportions are vastly different, but the majority of their heuristics are applicable to the problems faced in this project. Specifically, they state that:

- The proximity of the end effector to the goal increases the end effector's importance.
- If an end effector is moving towards the goal, it is likely to be important, therefore incorporating the *predicted* proximity in the measure of the importance.
- If the end effector is moving away from the goal, it is probably less important than suggested by its proximity.
- If the end effector is far enough away from a goal, the goal is likely to be unimportant.

Using these heuristics, a better approach can be devised. In this project, however, instead of comparing the end effector with the goal in IK, instead the proximity of the end effector in the FK skeleton must be compared with the end effector position in the FK skeleton at the frame of impact.

Zordan & Hodgins (2002) use a blending technique in their work, where at some point in an animation, the sequence diverges from motion-capture source data towards an edited motion, and then moves back to the motion-capture data once again. This transition is triggered by external forces on their character such as being hit, and the motion-capture data does not exist to deal with the scenario, so the motion data is generated dynamically. Although they use both kinematics and dynamics in their motion synthesis, the concept of transitioning from FK to IK and back again is still relevant.

Simple blending of FK and IK skeletons may or may not produce acceptable results for the primary project aim, and instead it may be necessary to calculate an end effector trajectory through the course of the animation to produce better results. Additionally, the secondary aim of the project will almost certainly require some aspect of plotting end effector trajectories, as the shots are more significantly altered.

1.9 Style-based IK

Creating an IK pose that retains the characteristics of an FK pose should be possible by using some iterative method that begins at the FK source, and gradually moves closer to the solution. This might be by the Jacobian pseudoinverse method, CCD, or some other iterative method. Although iterating from the FK pose to the IK pose at each frame will produce a pose at each frame that is similar to the source pose, the resulting poses may have discontinuities between them from one frame to another. To eliminate this problem, the IK solution for the *i*-th frame can be produced by iterating from the (*i-1*)-th frame. However, as described in Figure 1.5, a problem arises in that successive iterations can result in unnatural poses. Even if these poses can be made feasible by adding in joint constraints to the IK solver, there is no guarantee that the resulting IK animation will demonstrate the characteristics of the FK motion.

Attempting to create animations that adhere to a given style is a case of removing the redundancy from the solution in a specific way. Grochow et al. (2004) demonstrate a method whereby source animations are processed off-line in order to obtain style information for the sequence, which can then later be used to choose an IK pose which fits the style best.

The authors propose a method of modelling the likelihood of motion-capture poses, called a Scaled Gaussian Process Latent Variable Model (SGPLVM). Each motion sequence is processed off-line with a learning algorithm, before it can be used to generate novel poses. Each pose in the sequence is assigned a *feature vector*. The feature vector describes all of the features of the character pose that the learning algorithm should take into account. The feature vector includes all of the joint angles of the skeleton, a measurement of the character's orientation with respect to the world up-axis, as well as the velocity and acceleration of these features. The velocity and accelerations are measured, as the new pose should be sensitive to the pose in the previous frame when considering animations. These feature vectors are then processed in order to find a set of parameters that can be used to describe the motion in a low-dimensional space, called *latent spaces*. Examples of these latent spaces are shown below in Figure 1.8.

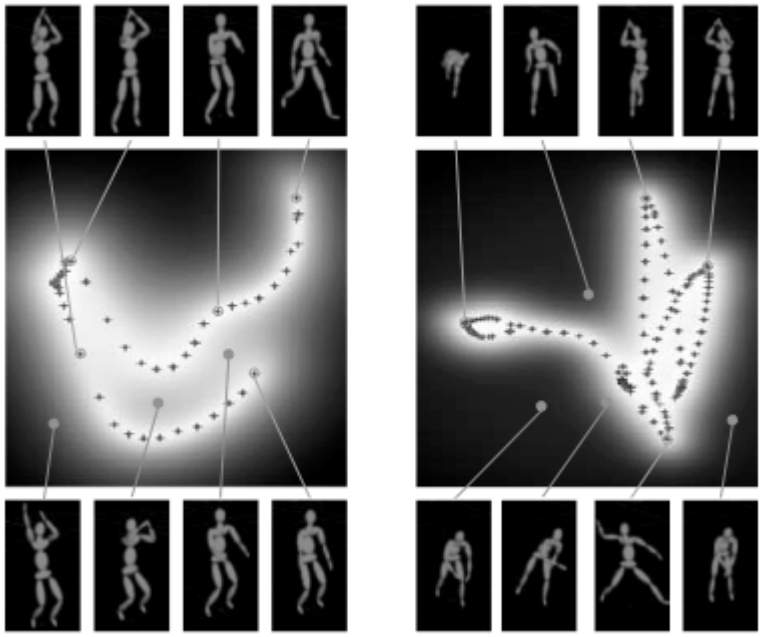

Figure 1.8 – SGPLVM latent spaces from two motion-capture sequences: a basketball jump shot(left); and a baseball pitch (right) Taken from (Grochow, Martin, & Hertzmann, 2004).

In Figure 1.8, the plus sign marks represent poses that are present in the original motion-capture data. The images above and below show either existing poses (marked by a line ending on a plus sign) or novel ones (marked by a line which extends into empty space). The likelihood of a new pose is calculated with a function which takes the original poses and the SGPLVM parameters as arguments. The two spaces in Figure 1.8 show all possible poses for the learnt model, and their likelihood is shown by the brightness of that particular pixel.

This algorithm can be used for many applications, and of particular interest for this project are its applications in interactive character posing and trajectory keyframing. With interactive character posing, an animator can pose a character by moving handles (like end effectors in normal IK), and the system will provide a feasible pose for those constraints based on the learnt model. Trajectory keyframing allows an animator to set keyframes on the handles such that the motion is altered to perform a different, yet similar action. The example given by Grochow et al. is of turning an over-arm baseball pitch into a side-arm one, by altering the trajectory of the handle controlling the character's throwing hand. This example is shown in Figure 1.9.

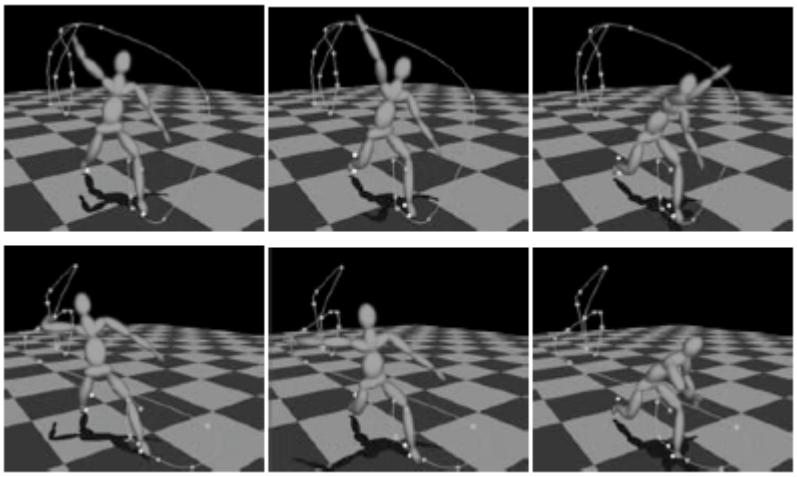

Figure 1.9 - Trajectory keyframing of a baseball pitch. Taken from(Grochow, Martin, & Hertzmann, 2004)

Given two different sets of motion-capture data and the same handle positions, this system would generate two different poses. This is something which is highly relevant to this project, as the arm positions for a certain shot should depend not only on the position at which the ball should be hit (i.e. the handle position, in the SGPLVM system) but also on the overall characteristics of the shot. The demonstration of trajectory keyframing is also relevant to the project, particularly the secondary aim.

The algorithm presented by Grochow et al. is slightly limited by the fact that it does not include any dynamic modelling, and could therefore fall foul of one of the problems with IK, in that animations generated can look slightly mechanical and robotic. With clever keyframing though, it might be possible to reduce this.

1.10 IK-based Style

As style can be derived from motion-capture data to drive an IK model that conforms to that style, it is reasonable to assume that changing parameters in the IK solver can induce style changes.

In section 1.7.2, pseudocode is shown for the normal operation of solving an IK joint chain using the Jacobian inverse (or pseudoinverse). Meredith & Maddock (2005) describe a simple modification where each angle in the vector $\boldsymbol{\theta}$ is assigned a weighting value between 0 and 1. When the weighting for a particular joint is small, the changes in angle are smaller, resulting in the joint appearing stiffer. This effect can be seen in Figure 1.10.

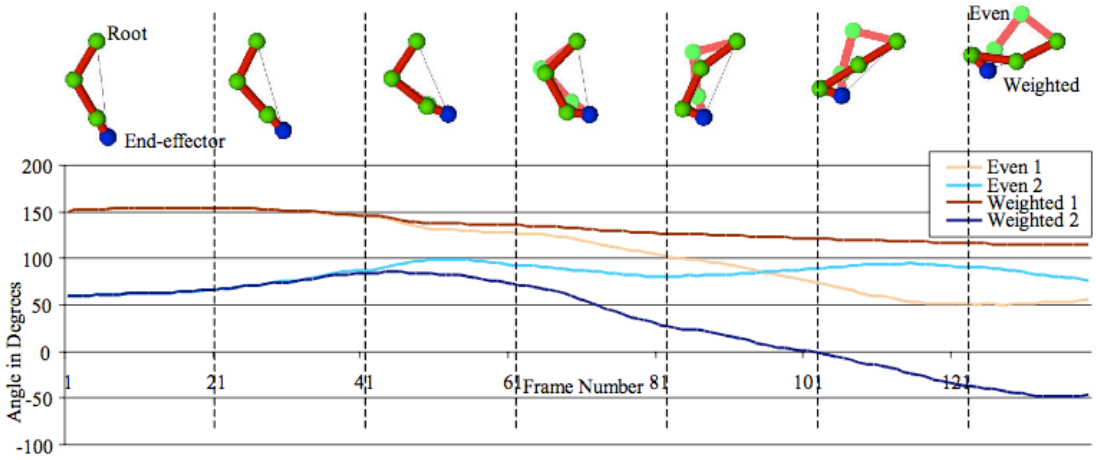

Figure 1.10 - Demonstration of a weighted IK chain. Taken from Meredith & Maddock (2005).

In Figure 1.10, the root joint has a lower weight than the second joint in the chain, and as a result the final pose shows that the first joint in the chain moves far less than when the weights are not applied. The graph shows that the joint with the higher weighting value (the bottom line) has to move a lot further in order for the system to reach the goal.

The paper then goes on to show that with careful control of the weightings, it is possible to produce predictable changes in motion style, by using a weighted IK system with motion adjustments made with inverse kinematics. The alterations to the weightings are can be adjusted heuristically to produce certain changes in motion: the paper gives the example of making a character limp by increasing the stiffness of the either the knee or the hip.

Combining the findings of Meredith & Maddock's paper with the findings of Grochow et al. (2005), it is reasonable to assume that there is scope for investigating the possibility of applying a learning method to motion-captured cricket shots in order to estimate suitable values for joint stiffness in the IK skeleton. For example, if a particular shot is played with minimal wrist movement, this could be detected by measuring the velocity and acceleration of the joints in the motion-capture data, and applied to the skeleton used when solving the IK chain. Thus, when adjusting the motion to fit the new constraints (such as bat position or angle), the IK solver should tend to make more adjustment in the elbow and shoulder joints than the wrist joints. Once again, the effectiveness of this solution in producing realistic results would have to be tested via experimentation.

It should also be noted that joint stiffness can be simulated in the CCD algorithm. At each step of the solution, the rotation applied to a joint can be clamped to only rotate by a certain amount. For example, a joint can be constrained so that each time the algorithm visits it, it can only rotate by, say 10 degrees. However, if the joints are constrained too much, the large speed advantage provided by CCD compared to the Jacobian pseudoinverse method can be lost, as the number of iterations required to reach the goal position will increase.

2 Solving the hit-point problem

This section describes the step undertaken to solve the problem of adapting the FK animation to allow the bat to hit the ball in a new location.

2.1 The Skeleton

The kinematic skeleton used for this project is defined by the skeleton in the source shot animation files provided by Codemasters, with some minor changes. Its structure is shown in Figure 3.1.

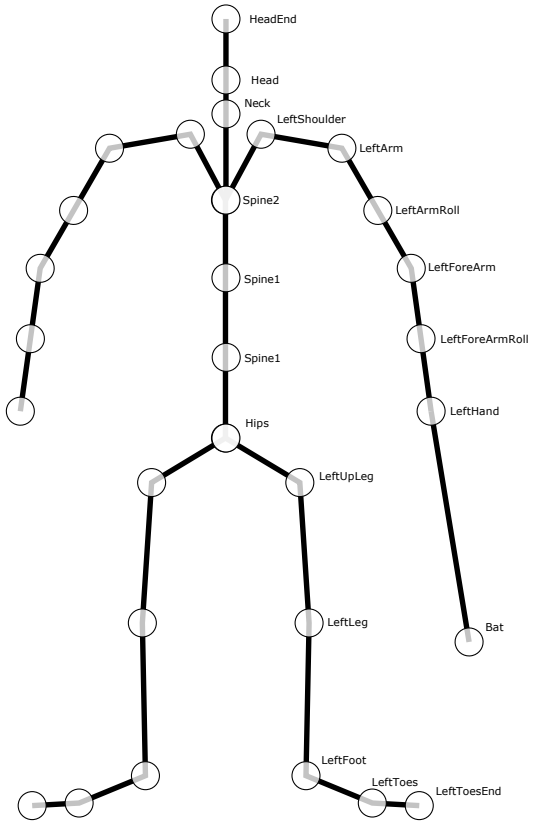

Figure 2.1 - The Kinematic Skeleton with joint names (all LeftXxx joint names are mirrored as RightXxx in the right side of the skeleton)

The skeleton defined in the source animations also included joints superfluous to this application, such as joint for facial animation. These joints were removed from the structure. The skeleton also includes finger joints. These were left in the skeleton, and remain untouched in the target animations, as they do not contribute to the hit-point solution in any way.

Each joint is defined as a set of Euler angles. Euler rotations describe 3D rotation by a rotation (in degrees, in this case) around the x-, y- and z-axes. With this joint structure, the joints are oriented such that the x-axis points directly down the bone. The rotation order is ZYX, which means that the Z rotation is applied first, followed by the Y rotation, then the X rotation.

2.1.1 Adding a joint to represent the hit point

The *Bat* joint was not in the source data. Instead, the source data included an entirely separate skeleton consisting of just one joint to represent the bat. The position and orientation of this joint was found to be in a fixed location and orientation relative to the left hand joint, so simply adding a new child joint to the left hand to represent the middle of the bat does not change the animation in any way. Instead, it provides an end effector that can be used in the IK solution for the desired hit point on the bat. The source files provided by Codemasters also included the geometry for the bat and ball, so the *bat* joint was placed at roughly the centre of the bat, but lifted away from the surface of the bat by the radius of the ball. This means that if the ball is placed at the location of the bat joint, the ball just touches the surface of the bat. It should be noted that the bat joint position on the bat's face is arbitrary, and could be moved to make the bat hit the ball at a different location, such as making the ball hit nearer the handle, nearer to the edge, or even to miss the ball if so desired.

2.1.2 Joint degrees of freedom

The different joints in the skeleton have certain degrees of freedom, depending on which human joint they represent. For example, the *LeftForeArm* joint represents the elbow, and as such only has one DOF; the *LeftHand* joint represents the wrist, and has two DOFs with the wrist being unable to twist. This seems slightly unintuitive, given that a human wrist can apparently twist around the forearm axis, yet the elbow clearly only has one degree of freedom. However, this apparent wrist twist is actually generated in the forearm by two parallel bones rolling around each other. This is simulated in the skeleton by the placement of a 'roll bone' between the wrist and the elbow, with all of the forearm twist attributed to this joint. There is also a roll bone between the shoulder joint and the elbow, but this is more for smoothing the geometry skinning (where a character mesh is attached to the skeleton structure) than for representing the real life human joints. There are other joints in this structure which do not map directly to bones in the human skeleton, such as the spine joints, and the 'Shoulder' joints, which are approximate representations of the clavicle (collar bone).

2.1.3 Joint rotation limits

As well as certain joints only having certain degrees of freedom, the joints are constrained in their range of movement to roughly correspond to human motion limits. For example, the elbow is constrained so that it cannot flex backwards (i.e. the Y component of the Euler rotation is constrained to be greater than zero). When considering the bones which do not have an equivalent in the human body, the joint constraints are set heuristically, by rotating the joints to what seems feasible for a maximum and minimum in each of the three axes, and setting these as the constraints. These simple constraints ensure that all poses are humanly possible.

2.2 The IK solution

2.2.1 IK algorithm choice

The IK algorithm chosen for the application was a CCD algorithm, as described in section 1.7.1. The reasons for choosing this over the Jacobian methods were:

- CCD tends to converge faster than Jacobian methods, which makes it a more suitable solution for real-time applications.
- CCD is quicker and easier to implement.
- CCD is intuitively comprehensible, which makes tweaking variables in the algorithm (such as joint stiffness) more predictable.
- There is a natural lower-joint bias in CCD (effectively making the lower joints have less stiffness than ones higher up the chain) which would seem to mimic the human arm and torso (i.e. the wrist is more flexible then the elbow, which is more flexible than the shoulder, which is more flexible than the collar bone, and so on.)
- CCD is, unlike the Jacobian methods, immune to singularities (i.e. when the joint chain is in a straight line).

As previously discussed, IK solvers can be made to solve for goal position, or for goal position and orientation. Initially, the IK solver was programmed to solve for position only. Orientation constraint could be added if required.

2.2.2 IK setup

It was decided that the IK joint chain would not simply affect the character's arms, and that the upper-body should also be allowed to move in order to adapt the shot to the new hit point. The start joint for the IK chain is *Spine* (from Figure 2.1) and the end effector is the *Bat* joint. The reason for choosing *Spine* as the start joint is that this is the longest chain possible without including the skeleton's root node, which in this case is *Hips*. A long chain is desirable because it means that the joint angle changes are

shared over more of the skeleton, which should lead to poses more similar to the source pose than if a short joint chain is used. For instance, if the IK chain only extended from the elbow downwards, there would need to be much larger changes in those joint angles than if other joints could 'help' reach the solution.

Also, the start joint must not be the root node, otherwise the entire rest of the skeleton would be affected by the solution. In this instance, if *Hips* was included in the IK chain, then its adjusted rotations would cause the legs and feet to rotate, and they would penetrate the floor, slide around, float, etc. which would not lead to a good solution. The skeleton could be re-rooted so that one of the feet became the root, which would prevent the aforementioned problems for one of the feet, but the other foot would still have the same problems. These problems could be fixed by introducing another IK chain to fix the legs back in position, but this just adds extra steps in the computation of the overall solution, and provides more opportunities for further problems to arise and destroy the realism of the animation.

As the location of the bat is parented to the left hand, and the left hand is to be moved by the IK chain, the right hand will come away from the bat handle, and will therefore need to be cleaned up once the position of the left hand is known. It turns out that the location and orientation of the right hand with respect to the left hand is not fixed throughout a shot animation – sometimes it grips the handle tightly, sometimes it comes slightly away as the grip loosens. Therefore, it is not enough to parent the right hand IK goal to the left hand. Instead, the location and orientation of the right hand must be computed on a per-frame basis, using the source skeleton for reference. For each frame, the orientation and location of the right hand is calculated with respect to the left hand in the source skeleton. This relative offset is then applied to the target skeleton with respect to the new left hand position and orientation.

The skeleton is shown below with areas affected by IK chains marked. The red bones show the left arm's IK chain, with the right arm's IK chain marked in blue. The grey bones indicate those that will not be affected at all by the IK system. The green bones are those which will be affected by the IK system as they are below the left arm IK chain root joint. However, their local joint angles will not be altered. Instead, the position of the head and right clavicle depend on the highest two joints in the left arm IK chain.

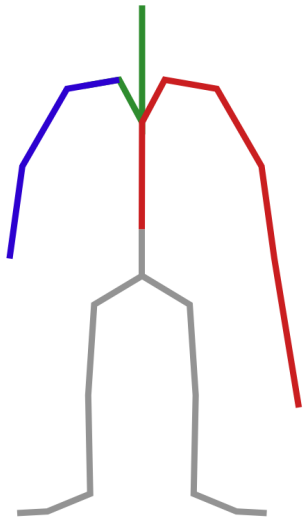

Figure 2.2 - Left arm IK chain (red), right arm IK chain (blue), unaffected bones (grey), affected but not adjusted (green)

2.2.3 The IK start pose

Numerical IK algorithms, like CCD, being with the joints in the chain in one configuration, and end with the joint angles being changed to produce a new pose. With IK systems like this, it is normal to give each joint a 'preferred orientation', resulting in the entire chain having a 'preferred pose'. This pose is usually set to some (essentially arbitrary) 'neutral' pose – with human characters this is normally the T-stance, where the feet are together, legs straight, with arms outstretched.

The reason for this is for repeatability of poses – there is a one-to-one mapping between the IK handle configuration and the resulting pose. With complex human motions such as cricket shots, the poses are not just dependent on the IK handle configuration, but the type of shot being executed. One shot may require a very different pose to another shot, even if the bat is required to be in the same location.

For this reason, instead of beginning at some consistent, neutral pose for each IK calculation, the current FK pose is used in order to ensure as much similarity between the FK and IK poses as possible. As a result of this method, placing the IK handle at the end effector location of the FK joint chain results in the IK solution being equal t the FK solution. In effect, the IK solution is 'found' without having to do any work. This has ramifications when it comes to blending between the IK and FK poses.

2.2.4 Calculating the IK handle position

Before explaining the IK/FK blending employed in this application, it is necessary to understand how the required hit-point offset between the FK shot and the IK shot is calculated and used.

Firstly, the frame in the animation at which the ball is to be contacted is required (call this the hit-frame). The location of the desired hit point is given. At the hit-frame, a vector is created from the FK bat joint to the hit-point – this vector is then converted to be relative to the FK bat joint, so as the FK bat joint moves and reorients throughout the source animation, this offset is known in FK bat joint-space.

This offset represents the difference between where the FK bat joint is, and where the final bat joint must be to result in a correct ball hit.

2.2.5 IK/FK Blending

When using a combination of IK and FK to produce a pose, the pose required will either be fully-FK, fully-IK, or some combination of the two.

The source animations provided all begin with the character in the same initial pose, and all finish at the same final pose. In this example, the animations all have common start and finish poses as they are used in a wider animation context, where the action preceding the shot animation (waiting to receive the delivery, in the case of cricket) and after the shot animation (perhaps starting a run, or going back to some idle pose) must be able to be fit with whatever happens during the shot. Because of this, at the beginning and end of the shot, the solution must be the fully-IK pose. At the point when the ball contacts the bat, the pose must be fully-IK.

Between these points in time, the pose will be somewhere between the two. Typically, this case is dealt with by a process called IK/FK blending. With IK/FK blending, the IK pose is calculated, and a value between 1 and 0 dictates the IK blend. The following pseudocode demonstrates how this parameter is used to calculate the final, blended pose.

```
Vector ikGoalPosition
float ikBlend

Pose fkPose = getFkPose()
Pose ikPose = calculateIkPose(fkPose, ikGoalPosition)
Pose finalPose

for i = 0 to jointChainLength
   finalPose.joint(i).angleX = (ikBlend * ikPose.joint(i).angleX) +
        (1.0 - ikBlend * FKPose.joint(i).angleX)
   finalPose.joint(i).angleY = (ikBlend * ikPose.joint(i).angleY) +
        (1.0 - ikBlend * FKPose.joint(i).angleY)
   finalPose.joint(i).angleZ = (ikBlend * ikPose.joint(i).angleZ) +
        (1.0 - ikBlend * FKPose.joint(i).angleZ)
next i
```

The solution developed does not use this type of interpolated blending – instead, the poses are fully-IK for the entire animation. The 'blending' between the poses is obtained by interpolating the IK handle position between the FK bat joint position and the desired IK hit position. The vector representing the offset between the FK bat joint and the desired hit position in FK bat joint-space (call this the offset vector, described in 2.2.4) is multiplied by the blend parameter to obtain the desired IK bat joint position at a particular frame.

```
Vector3D offsetVector
Vector3D FKBatJointPosition
Float ikBlend

Vector3D ikHandlePosition = FKBatJointPosition + (ikBlend * offsetVector)
```

The question now arises of how the blend parameter is calculated. It is required that at the beginning of the animation the blend parameter must be zero, at the hit-frame it must be one, and at the end of the animation is must return to zero. This could be achieved by assigning a function as shown in Figure 1.7 (p. 10) but this is an inflexible solution and does not take into consideration how much importance is ascribed to the FK and IK bat joint positions. A better measure of this is the distance between the FK bat joint and the FK hit-point at any given time: when the distance between these two points is large, the hit-point is of little significance in terms of the precise positioning of the bat joint; when the distance is very small, the location of the ball relative to the middle of the bat is very important.

As such, the blend parameter is defined as follows:

```
Vector3D FKBatJointPosition
Vector3D FKHitPoint
Distance FkBatToHitPoint
Float    falloffDistance

Float ikBlend = (-FkBatToHitPoint / falloffDistance) + 1.0
```

This blending factor is clamped to be in the range 0 to 1.

The `falloffDistance` parameter describes the distance between the FK bat joint and FK hit point at which the pose will begin to diverge from the FK pose. Above this distance the pose remains fully-FK, whilst below this distance the final pose bat joint position begins to be influenced by the vector offset required for the bat joint to reach the hit position. The value of this parameter must be such that at the start and end of every shot, the poses are fully-FK. Also, the value must be large enough that the transitions between FK and IK poses are smooth. This parameter is measured in arbitrary distance units, according to the size of the skeleton used in the FK animations. The upper limit of this value can be calculated by finding the smallest distance between the FK bat joint and the FK hit point, for the frame of every animation, for every animation in the set. The lower limit of the value has to be judged perceptually – what is the value at which the transition becomes too abrupt? Suitable values for this parameter turn out to be in the range 70-110 units. Of course, these limits could change if the animations in the set were changed, or if the requirement for fully-FK start and end poses were dropped (it just so happens that this is a requirement for this application).

An example of how this importance-based method of calculating the blend parameter differs from a simple time-based approach is shown below. The red line represents an example of how a time-based blending function might look, whereas the blue line shows the blend value calculated with the importance-based approach for a specific shot animation.

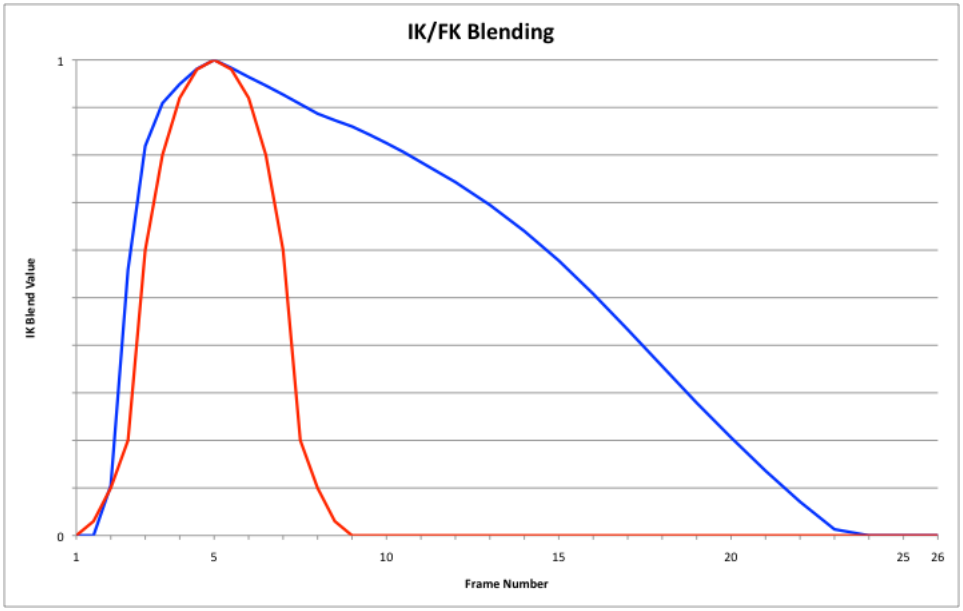

Figure 2.3 - Time-based (red) and importance-based (blue) IK blending

The reason for the marked difference between the two plots is that in the animation in question is of a defensive shot. The bat very quickly moves into position at the hit point (frames 1-5) but moves away from this hit point very slowly. It would be wrong to use the time-based approach here, as the very nature of the stroke means that the position of the hit point remains an important feature of the stroke animation in the subsequent frames, more so than if the bat moved away from the hit-point more quickly.

This method of calculating the blend parameter does not require any knowledge of the nature of the stroke, whereas achieving a good result from a time-based approach does require knowledge of the stroke. For example, consider what would occur with these two approaches if there were a stroke in which the pose remained constant position for a number of frames. The importance-based method would account for this – the distance between the FK bat joint and the FK hit point remains constant and thus the ball importance is constant, so the blend parameter remains constant and the pose does not change. Considering the time-based approach, the blend factor would be changing over time, so the pose would gradually change, which results in the new animation not remaining true to the style of the original.

2.3 Issues encountered

2.3.1 Erroneous solutions

The first issue that was encountered was that some of the solutions provided by the CCD IK solver were highly undesirable. One such extreme example is shown below.

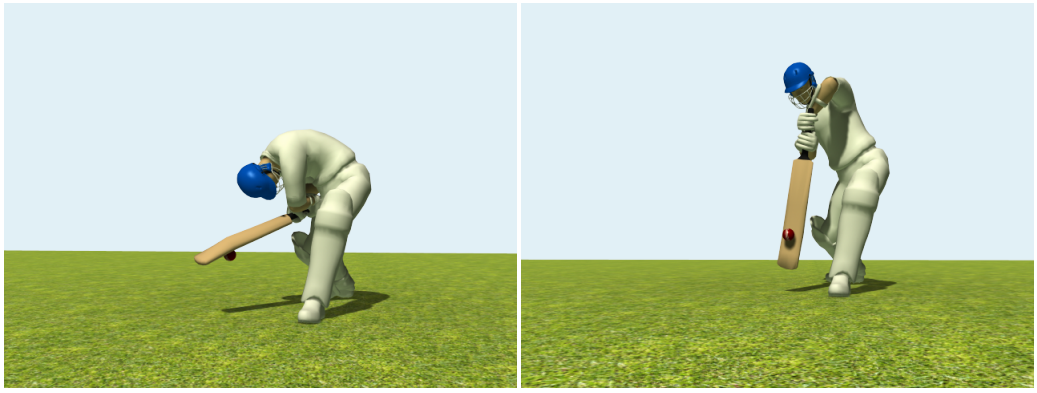

Figure 2.4 - An erroneous pose solution (left), and the source pose (right)

Whilst this pose is humanly possible (as the joint angles are constrained within reasonable ranges) and the bat contacts the ball, it is clearly not remotely similar to source pose. The reason for this problem is the nature of the CCD algorithm – the lowest joint has the highest priority. In this instance, the lowest joint is the left wrist joint, which has two degrees of freedom. These degrees of freedom allow the wrist joint to attempt to solve a good deal of the difference between source and target hit positions by itself. For this particular hit location, the optimal solution is for the wrist joint to assume an orientation that is vastly different to the source orientation. Although this is an extreme example, there are instances where the effect is more subtle, though still undesirable. An example is shown below, along with an example of the solution to this problem.

Figure 2.5 - Erroneous solution (left); source pose (centre); a better solution (right)

The left image shows the solution provided when the wrist has two degrees of freedom. The centre image shows the pose from which the solution is derived – the target hit point is slightly higher, and to the left of the source hit point (as viewed in these images). It is evident that some of the 'style' of the shot has been lost in the new

solution as the angle that the bat forms with the ground plane is very different. Looking more closely at the source pose, the left forearm is almost parallel with the bat whereas there is around a 45-degree difference between the forearm and bat in the left-hand image.

2.3.2 Predictability of solutions

Consider the direction in which the ball is likely to travel after hitting the bat in Figure 2.5. In the source pose, the ball will seemingly travel towards the camera but slightly to the left. In the erroneous IK solution, the bat is facing almost directly at the camera. Given that the hit point is moved to the left, it is reasonable (and intuitive) to assume that the 'same' shot applied to this new ball location should result in the ball being hit further to the left. Similarly, if the hit point was moved to the right, the ball should probably be hit further to the right.

This lack of predictability of the solution is demonstrated in the figure below.

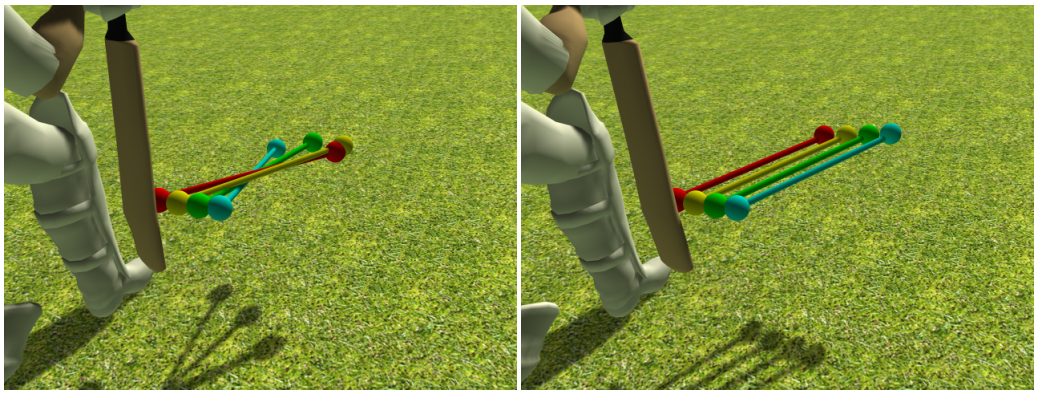

Figure 2.6 - Hit locations and ball trajectories for two different IK chains

In order to produce Figure 2.6, a rudimentary physics simulation was implemented to predict the path of the ball after having been hit by the bat. The physics simulation is limited to taking into account a vector protruding from the face of the bat as well as a small amount of the initial ball trajectory (as it bounces up from the floor). It does not take into account the trajectory of the bat, spin, etc.

Figure 2.6 shows how the hit point maps to the ball trajectory when the source shot is the same. The left image shows four hit points and the ball trajectories when the left wrist has two degrees of freedom. It shows that the mapping between hit location and ball trajectory is not well defined. It appears that there may be a weak correlation in that as the hit point moves one way, the trajectory moves the other way. This is not

what one would expect if the shot being applied is the same and only the hit point is moving.

With the same shot being used to hit the four different ball locations, one would expect to see something more like the right-hand image in Figure 2.6. Here, there is a clear mapping between hit point and ball trajectory.

One thing to note at this point is that, as mentioned in section 2.2.1, the IK solver only solves for position of the end effector rather than the position and orientation. With the wrists constrained, there is a natural tendency for the orientation of the bat to change as the hit point changes, so it seems that there is little need to explicitly dictate the bat's orientation. In fact, defining the difference in orientation based on the offset from the FK hit point to the required hit point would be difficult to quantify.

2.3.3 Solving the problems

In Figure 2.5 and Figure 2.6, the left-hand image shows the problems encountered whilst the right-hand image shows a better solution. The adjustment made to the IK solution was simply to remove all degrees of freedom for the left wrist. This prevents the wrist from taking the vastly different orientations in search of a solution which lead to the poor results.

This adjustment also ensures that the angle between the bat and the forearm is preserved, which is one of the largest differences between the left and centre images in Figure 2.5. The result of this change is shown in the right-hand image in Figure 2.5. The ball is in the exact same location as the left-hand image, yet the solution is unquestionably more similar to the style of the source pose.

In Figure 2.6, the new wrist constraint is demonstrated to produce much more predictable and consistent solutions as the hit point is changed. This empirically shows that the style of the shot is better preserved by ensuring that the IK wrist joint takes its rotation directly from the FK solution.

2.4 Variations in solutions

Although there should be some degree of predictability and consistency in the solutions arrived at (so as to avoid situations like Figure 2.4), when simulating human motion, complete consistency and predictability is not desirable. This is because humans will never repeat the same action in exactly the same fashion – there will be some degree of variability.

There is one way in which this system can produce variable results for the animations generated.

2.4.1 Shot trajectory

The first is to blend the IK hit point in differently, by changing the `falloffDistance` parameter. This will not have any effect on the pose at the point the ball hits the bat, but instead changes the poses before and after the hit point. As long as the parameter is kept within reasonable bounds (as established in section 2.2.5, p. 34) the resulting animations will remain sensible. Figure 2.7 demonstrates a possible variation in shot trajectory by changing the IK falloff distance parameter.

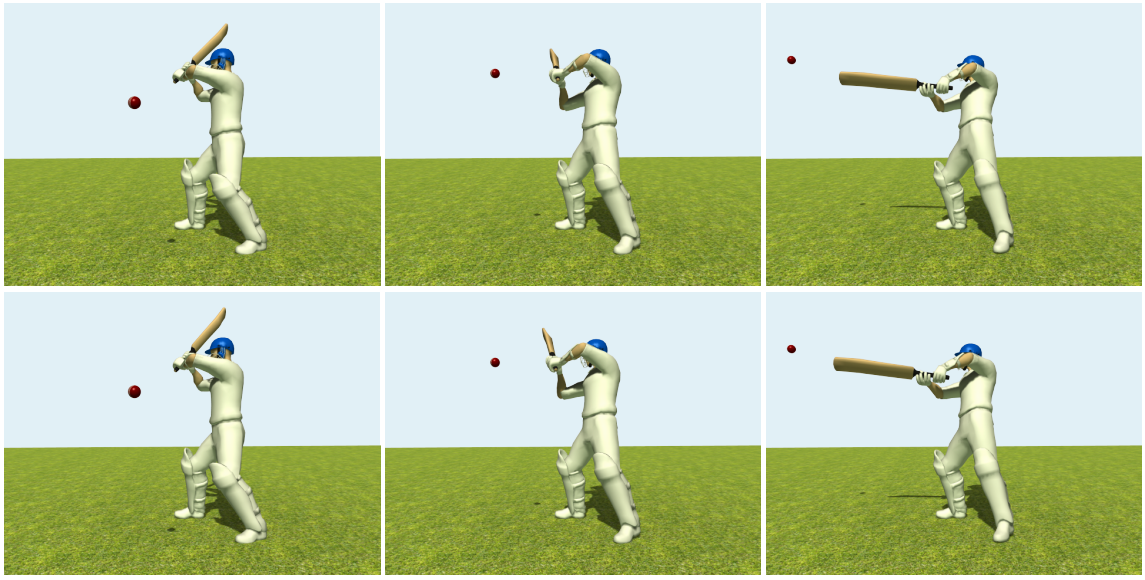

Figure 2.7 - Variations in shot trajectory with varying falloff distance parameter (top row: 70; bottom row: 110)

The frames shown are the two preceding the hit-frame, and the one following it. The top row shows the animation produced with the falloff distance parameter set to 70, whilst the bottom row shows it set to 110. The hit point is the same for both animations shown, and is higher than the source hit point. In the middle frame, the bat can be seen reaching up higher and earlier in the bottom row compared to the top row. Although the difference between the solutions in the first and last frames in Figure 2.7 is not as large as the middle frame, there is still some amount of difference – notice the gap between the helmet and the bat in the first frame, and the angle between the bat and the floor in the last frame.

2.4.2 Limitations

There is one limitation to this method of creating variation in the animations generated. The amount of variation possible is dependent on the distance between the FK hit point and the IK hit point. If the desired hit point is exactly at the FK hit point, then no variation in the solution is possible as the FK animation *is* the solution.

In order to allow variation at the FK hit points, an FK animation with a different hit point would be needed. This could be provided by motion-capturing multiple versions of the same shot with slightly different hit points, by using the IK solution to generate these variations off-line, to be used as new FK animations, or to synthesise variations to the shots off-line in some other way. One possible approach to this is detailed in section 3 (p. 50).

2.5 Results and Analysis

2.5.1 Obtaining the results

The IK system was tested using *Autodesk Maya*, a commercial 3D graphics package, as the way of setting up an interactive scene, rendering in real-time. The basic scene setup is shown in Figure 2.8.

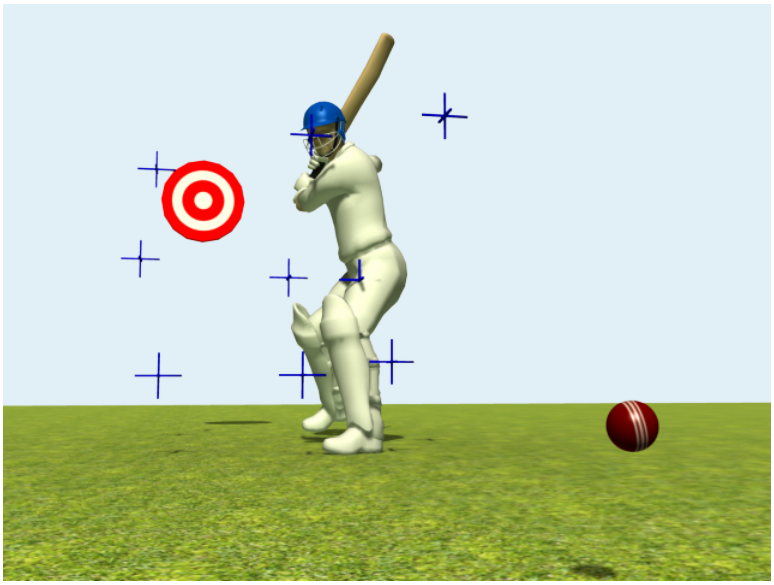

Figure 2.8 - The basic scene setup for testing shot adaptation

The blue crosses represent the hit points for nine different shot animations loaded. The shot animations to load were chosen to enable a reasonably full coverage of possible ball locations – i.e. left-to-right and low-to-high. These shots also represent a good deal of

the possible shot types, which is important in order to test that the system works for all possible styles of shot. For example, the shots include shots played with weight going forward, with weight going backward, with the bat approximately horizontal, with the bat approximately vertical, and with the ball being hit in a range of directions and elevations.

The red and white target represents the destination of the ball, travelling from a fixed point in frame one. When the target is moved, a straight line is calculated between the ball start point and the centre of the target, and the hit point that comes closest to this line determines which animation is to be used as the source. The desired hit point is calculated to be the point on the ball's path which is closest to the FK hit point, and the offset between the FK hit point and desired hit point is calculated as explained in section 2.2.4 (p. 32).

This setup enables the ball path to be changed and the newly generated animations to be calculated and viewed in real-time. Various frames as well as motion sequences can be rendered out for visual comparison, as seen throughout this dissertation. Also, the joint angles of the IK skeleton over the entire shot duration can be exported for numerical and statistical analysis, such as that carried out in section 2.5.4 (on page 44).

2.5.2 Sample results

To demonstrate the IK system working to adapt a shot to different shot locations, several frames of a source animation are shown with two adapted versions in Figure 2.9. The middle strip shows the source animation, and the other two strips show adapted versions. The top row shows the animation when the hit point is moved to the left by around 30cm in real-world units, whilst the bottom row shows the hit point moved to the left and higher. In the first frame, the poses are all identical (the IK blend parameter from section 2.2.5 is zero) and even though the full animation is over 30 frames in length, by frame ten the IK blend parameter is back to zero and the poses are identical to the source animation.

The figure also shows that the CCD IK algorithm produces pose variations that extend far beyond simply adjusting the arm structure. In the hit-frame (frame five) it is clear that the character has adjusted his pose from the torso to reach the ball – the entire upper body is leaning towards the ball to help reach it. This is in stark contrast to the bottom strip. Even though the ball is farther to the left than the source animation, the body leans away from the ball, rather than towards it as in the top strip. This is because the ball is higher up, but the arms cannot 'help' with the height, as the upper-left arm is already vertical in the source pose. One can imagine that attempting to reach

the ball in that position with that style of shot would be quite awkward, which is portrayed by the torso leaning away from the ball with the arms pressed very close to the body.

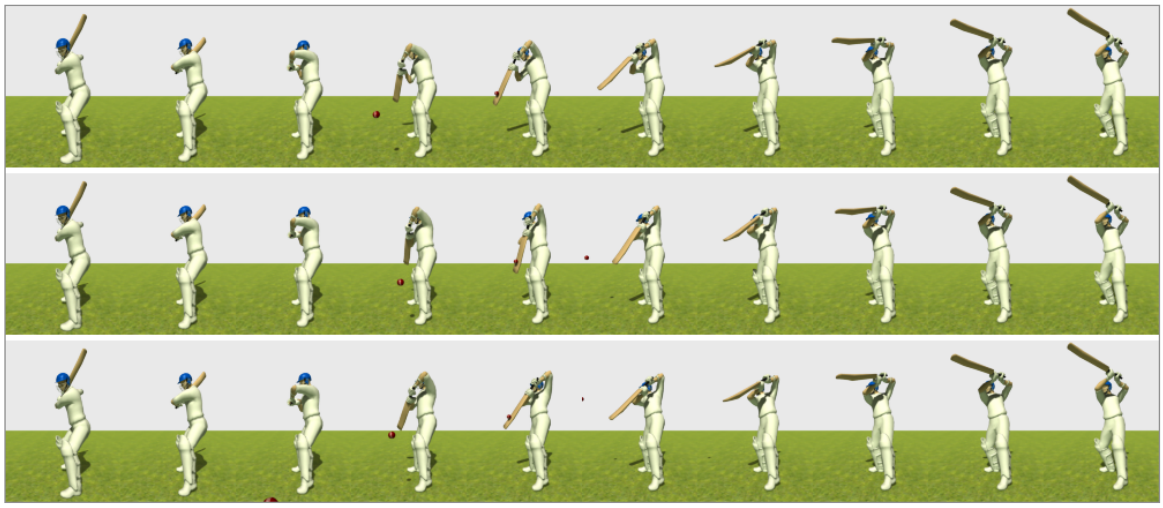

Figure 2.9 - The first ten frames of a source shot (centre) and two adapted shots

Another example of the IK implementation is shown in Figure 2.10. In this figure, only the hit-frame is shown for the source animation and the variations. The position of the image correlates to the hit point adjustment. That is, in the left-hand images the hit point is moved to the left, and in the top row the hit point is higher than the source, and so on.

This figure particularly highlights the way in which the solutions show a realistic, natural variation in pose depending on the hit point position. Again, there is significant movement in the torso when the hit point is nearer the character's body. Just altering the arm configuration cannot solve the problem, so the upper body leans away from the ball to make room for the arms.

Another point to note here is the natural variation in the angle of the bat. In the source shot, the character is attempting to hit the ball slightly downwards, towards the floor. As the hit point is moved, the likely result of the shot changes in the way one might expect of a real person performing the actions. When the ball is close to the body and high up (as in the top-right image) it seems intuitive to imagine that it will be difficult for the player to hit the ball downwards, and this is indeed how it looks in that image. In the opposite situation, when the ball is further away from the body and lower down (as in the lower-left image) the pose indicates that the ball will be hit downwards, and appears to be a more comfortable pose.

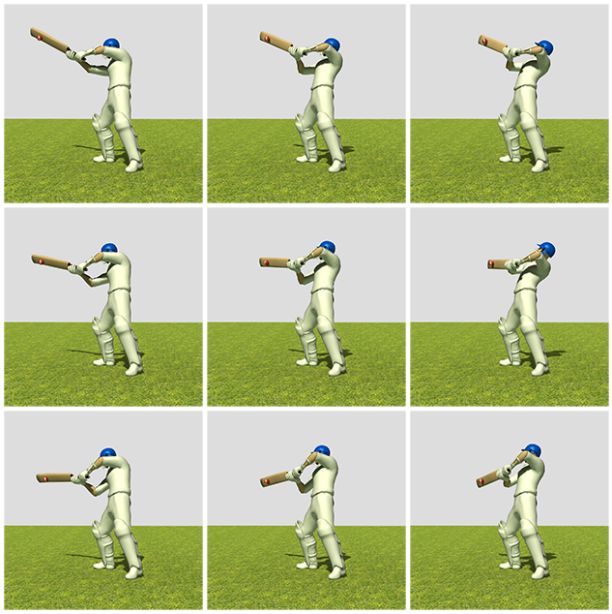

Figure 2.10 - A source hit-frame pose and eight variations

As previously mentioned, this variation in bat angle is occurring without explicitly defining the orientation of the end effector in the IK chain. The left wrist rotation is constrained, and only the end effector's position is defined.

2.5.3 Measuring shot style

A more objective method of measuring shot similarity between a source animation and an adapted one is needed, as opposed to comparing screen shots. For example, Figure 2.5 shows an apparently poor solution on the left, and an apparently better solution on the right. Whilst it might be clear by comparing images of the solutions to the source pose that one solution is better than the other at preserving the nature or style of the shot, that is not an objective measure.

One way to provide a metric by which to measure shot similarity would be to measure the difference in all the joint angles between the source and adapted poses for every frame in the animation. The sum of these differences produces a measure of difference between the source and target pose. However, this would only really serve to demonstrate that different hit points result in different poses. For example, if the hit point could be reached by *only* changing the wrist orientation, leaving the rest of the joints unaltered, this could result in a smaller 'difference score' than if all of the joints were altered by a small amount.

This method has no knowledge of the style of the shot and what equates to a shot of similar style – the method does not know which are the important relationships to maintain in order to maintain shot style. In Figure 2.5, one obvious feature that has already been pointed out is the angle between the forearm and the bat at the hit frame.

2.5.4 Dimensionality reduction to measure shot style

Principal component analysis (PCA) is proposed as a way of discovering the relationships in joint angles which result in a certain shot style. PCA is able to reduce the dimensionality of the data set such that only a small number of dimensions are required to describe the majority of the variance in the joint angles. A shot animation is a large number of observations in a high-dimensional space. Each observation is the measure of all the skeleton's joint angles, along with translation of the root node (this hips, in this case). However, these values may have strong correlations which can be discovered by using PCA. An example of a possible correlation might be the y-position (the height) of the root node and the bend of the knees – when the hips are lower, the knees are likely to be more bent. A less obvious example might be how the wrists rotate as the shoulders rotate for a specific shot. PCA finds these relationships and portrays them as orthogonal vectors (principal components) in the high-dimension space, with the first principal component being the one which accounts for the majority of the variance in the animation.

When the dimensionality of animation is reduced in this way, a simplified version of the motion is described by the top few principal components. This simplified version essentially encapsulates the most 'important parts' of the motion, which could also be described as the style of the shot. The number of principal components required to adequately describe the motion is dependent on the motion itself, and can be judged by reconstructing the animation using a certain number of principal components and observing it to see if the shot is identifiable.

Before performing PCA on the shot data, there are two pre-processing steps performed on the animation data. The first step is to convert the joint angles from Euler angles to quaternion representations. The reason for this is that Euler angles are nonlinear – i.e. the same orientation can be represented by different Euler angles. Also, consider that the angle 359° is only two degrees different from an angle of 1°, though numerically they are 358 degrees apart. Quaternion representations avoid these problems, as the orientation is described as a 3D direction vector and a twist amount (Gould, 2005).

Secondly, the sampled data is standardised by subtracting the average value for each parameter, and dividing by the standard deviation of each parameter. Dividing through

by the standard deviation is necessary as the parameters are not all in the same units – there are three parameters describing a 3D direction vector and another for a twist.

The PCA is performed in *Scilab* (an open-source clone of *MatLab*), and the output from it includes eigenvalues. These show how much of the variance is captured by each principal component, and can be plotted in a graph as in Figure 2.11. The blue bars (on the primary vertical axis) show the eigenvalues (as a percentage of the total variance) for each principal component, whereas the red line (on the secondary vertical axis) shows the cumulative percentage of the variance captured as the number of principal components increases.

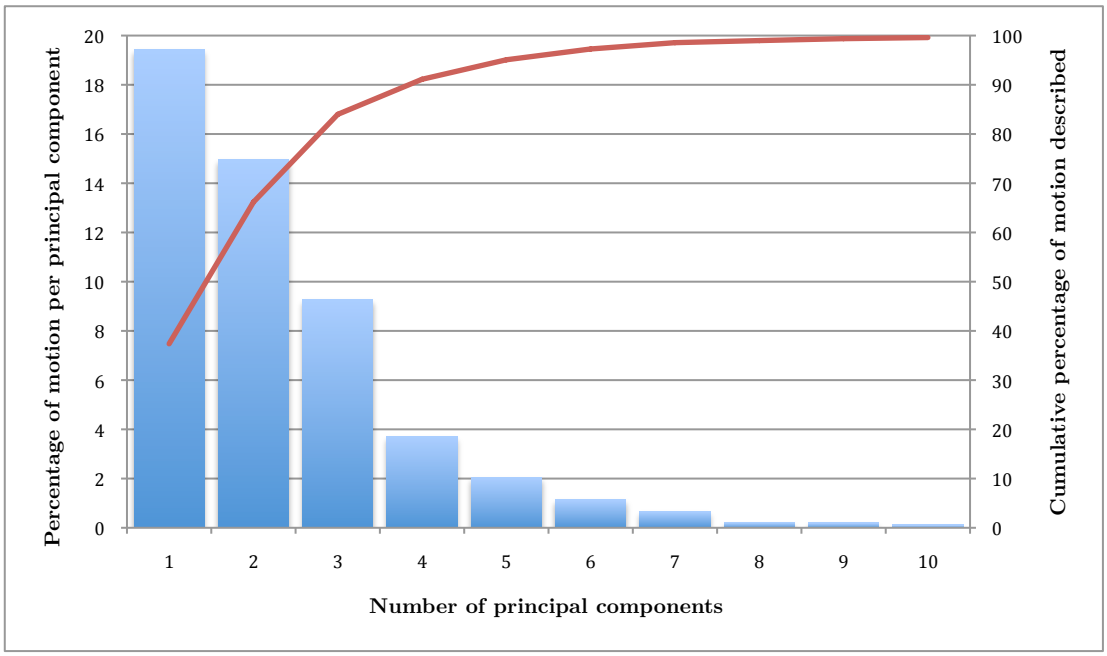

Figure 2.11 - Percentage of motion described by the first ten principal components a shot

This plot is for one particular shot (the source motion from Figure 2.5), but the plots for any given shot turn out to be roughly the same. By reconstructing the animation using an increasing number of principal components and comparing the reconstruction with the source animation, it can be determined how much of the variance must be captured in order to adequately describe the style of the motion. It turns out that anything upwards of 70% of the variance adequately describes the shot, which from Figure 2.11 means that three principal components are required. In fact, for the seven of the nine shots that were tested, three principal components is enough to capture 70% of the variance. This number of principal components can then be used to compare the source animation with the adapted version.

This comparison is produced by projecting the high-dimensional data for an adapted shot animation into the low-dimensional space defined by the top principal components of the source animation. These projections can be shown graphically as a plot of the PCA scores (a score is the representation of the original data in principal component space) for as many principal components as desired.

As an example, the source animation shown in the centre of Figure 2.5 was used to construct a principal component space into which the source animation and the other two animations depicted in Figure 2.5 are projected. The projections are plotted in Figure 2.12.

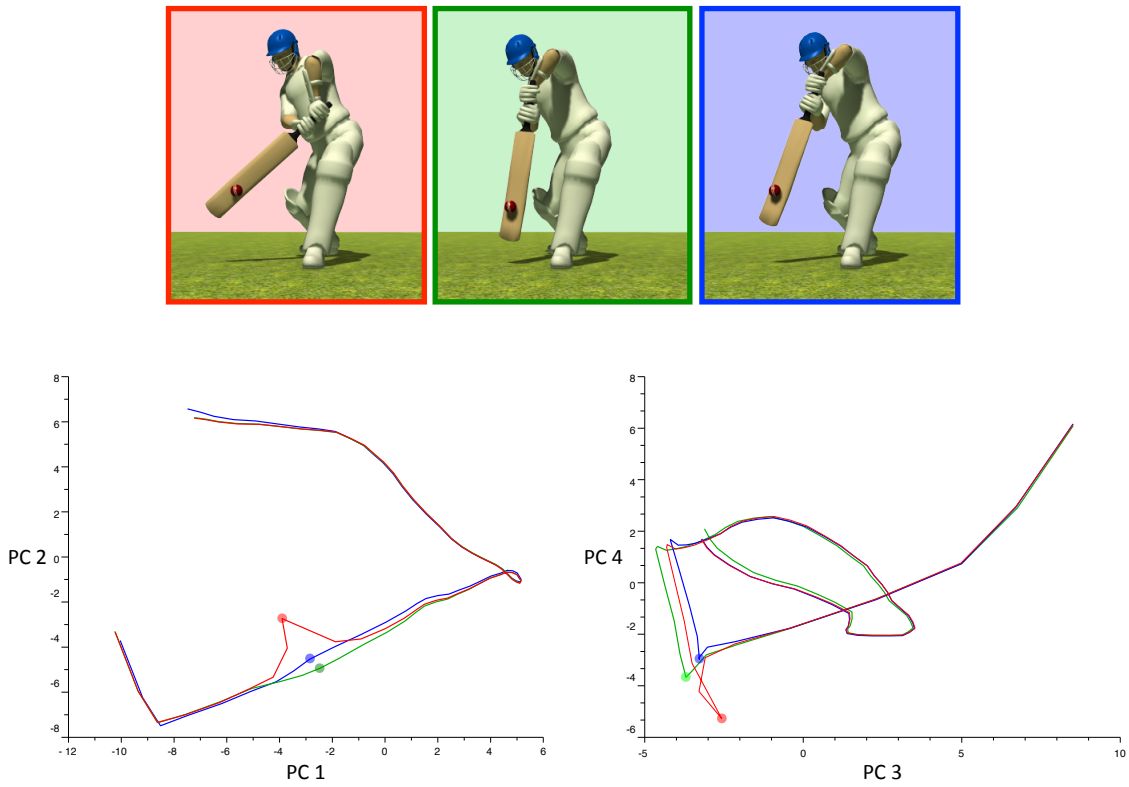

Figure 2.12 - Comparison of different hit point solutions in principal component space

In the above figure, the colours between the screenshots and the plots correspond, and the highlighted points in the plots correspond to the hit point shown in the screenshots. (Although three principal components would be enough to adequately compare the shots, it is easier to show the top four principal components as two 2-D plots rather than one 3-D plot.)

It is clear from this figure that there is a significant divergence between the red and green plots shortly before and after the hit point. This difference is the result of the significant change in the style of the shot at this point – the pose does not conform to the desired shot style. The blue line, however, follows the trajectory of the green line much more closely, with no large, sudden divergences as with the red line. This demonstrates that the adapted animation solution shown by the blue line is a measurably better solution at and around the hit point.

To demonstrate this method further, the top, bottom, left and right shots from Figure 2.10 (p. 43) are plotted in the principal component space of the centre image, along with the plot for the source shot. This plot is shown in Figure 2.13, below.

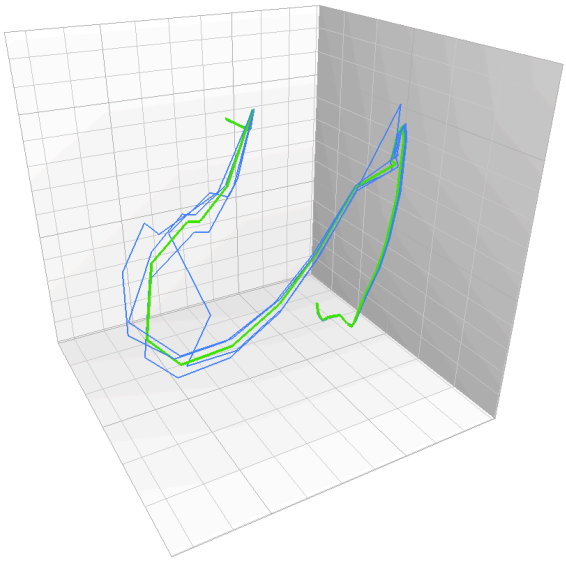

Figure 2.13 - 3D principal component space plot of a source shot (green) and four adaptations (blue)

In a similar way to the 'good' (blue) plot in Figure 2.12, there are no sudden deviations from the source shot plot. The adaptation plots do deviate from the source plot, which is to be expected as the shots *are* different, but the deviations are gradual, and still generally follow the path of the source shot.

2.5.5 Calculating style error

Although the hit point is shown to be better in the shot depicted by the blue plot in Figure 2.12, this does not account for the entire duration of the shot animation. In order to account for the whole animation, it is necessary to measure the difference between the scores for every observation in the set and obtain a sum of all these differences. One

possible way to measure the difference between these two sets of scores is to find the root mean square error, or RMSE.

Let θ_1 and θ_2 be the two data sets to compare:

$$\theta_1 = \begin{vmatrix} x_{1,1} \\ x_{1,2} \\ \vdots \\ x_{1,n} \end{vmatrix} \quad \text{and} \quad \theta_2 = \begin{vmatrix} x_{2,1} \\ x_{2,2} \\ \vdots \\ x_{2,n} \end{vmatrix}$$

The RMSE is then given as:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} \left(x_{1,i} - x_{2,i} \right)^2}{n}}$$

Where n is the number of observations (in the example above, the number of half-frames, 57).

This method allows the difference to be calculated between two one-dimensional vectors, but for this application, the measure must take into account the scores for three principal components. As a result, the $\left(x_{1,i} - x_{2,i} \right)$ term is deemed to be the Euclidean distance between the source and adapted scores in the principal component space defined by the first three principal components of the source animation.

$$\left(x_{1,i} - x_{2,i} \right) = \sqrt{\sum_{p=1}^{m} (s_{i,p} - t_{i,p})^2}$$

where:
i = the observation number;
p = the principal component number;
m = the number of principal components to be used (in this case, m=3);
s = the source animation;
t = the adapted (**t**arget) animation.

Thus, the final equation for calculating the difference between the source and adapted animations becomes:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}\sum_{p=1}^{m}\left(s_{i,p} - t_{i,p}\right)^2}{n}}$$

By using this equation, the total error for the different shot solutions shown in Figure 2.12 is calculated, and shown in the following table.

Shot Type	**RMS Error**	**% difference**
2-DOF wrist	0.398	29.3
0-DOF wrist	0.307	0

Table 2.1 - Calculated shot style error for two different IK setups

This shows that the error between the source and the adapted animations is different depending on whether or not the left wrist has, and that overall, the IK setup where the left wrist is locked produces a measurably better result.

3 Synthesis of new animations

3.1 Low-dimensional representations of animations

Sections 2.5.3 and 2.5.4 describe the process of comparing FK animations with versions adapted using inverse kinematics by way of constructing low-dimensional spaces into which the animation is projected. Each animation can be plotted in its own low-dimensional space, where each point on the plot represents an approximation of a pose struck by the character at some point in time. Some example plots are shown in Figure 3.1.

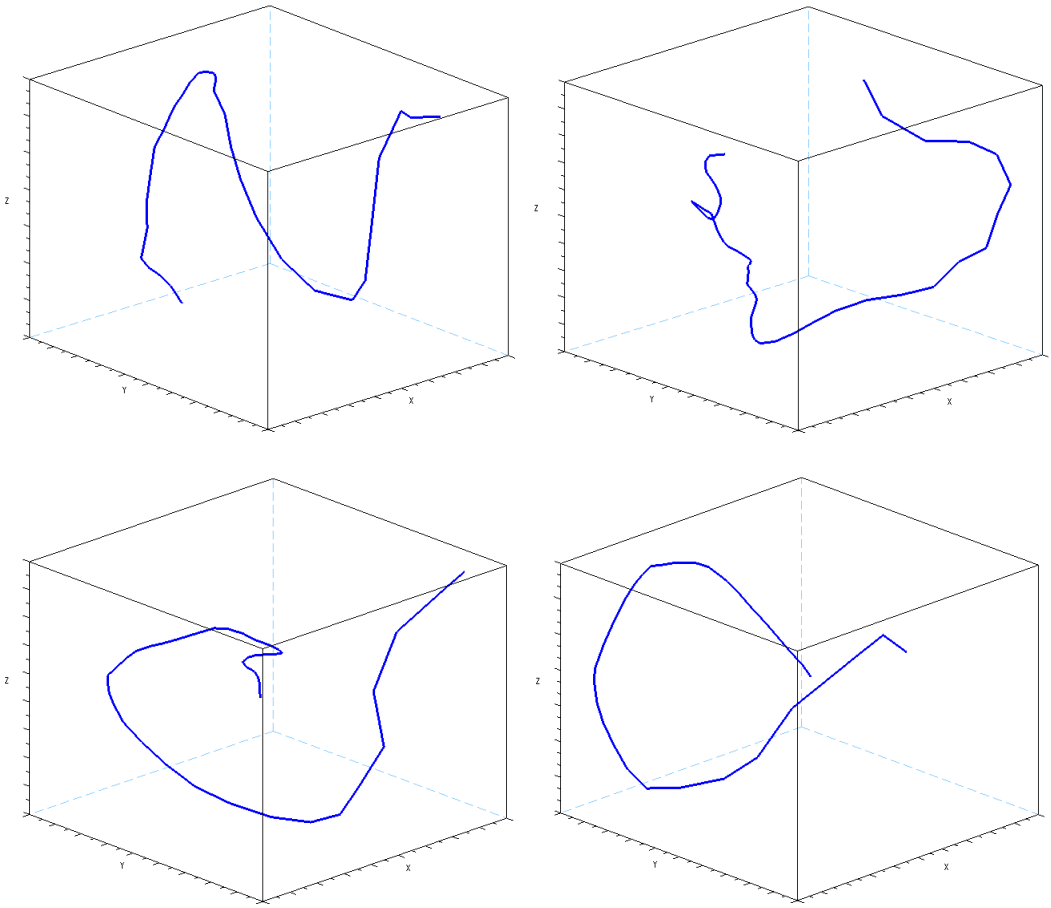

Figure 3.1 - Four sample shots plotted in 3D principal component space

The X-, Y- and Z-axes in the above plots are the first, second and third, principal components, respectively. What each of these axes actually represents in terms of joint rotations is dependent on the shot itself, and they are essentially arbitrary. However, it is possible to use the variable loadings (the quaternion parameter coefficients) produced by the PCA process to visualise the relationships between joint angles that have been

identified. For example, the top-left plot in Figure 3.1 shows the low-dimensional space constructed from the upper-body joint angles from the shot depicted in Figure 2.10. The loadings for principal components one to three are then used to reconstruct the motion defined by each principal component. This reconstruction is shown in Figure 3.2, below.

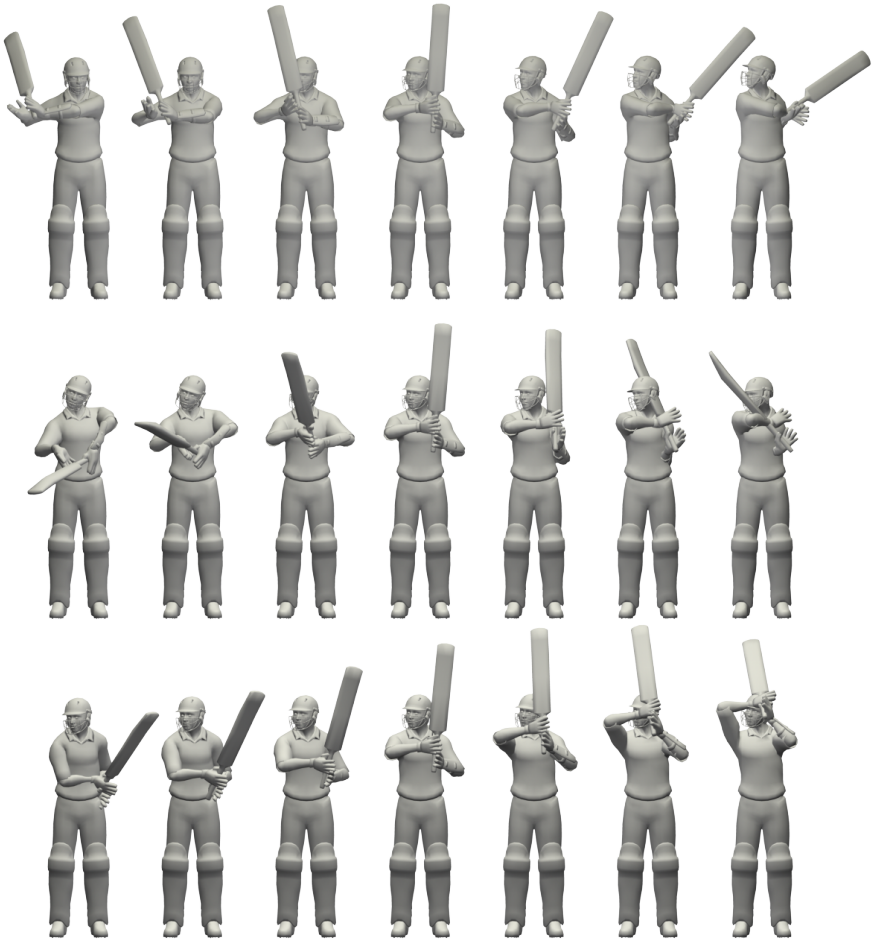

Figure 3.2 - Visualisation of principal components for a particular shot (upper body only)

Each row shows the effect of varying the score for the relevant principal component from -3 to +3 standard deviations of the observed scores for each principal component. The first principal component is shown in the top row; the second in the middle row; and the third in the bottom row. In each row, the score for every other principal component is set to zero, so it has no effect. The central pose in each row shows the score at zero, and this is the shot's average pose. It must be stressed that in Figure 3.2, there is no concept of time shown – the observed scores increase and decrease over time, but the figure simply shows the scores increasing from left to right.

From this figure, it is possible to discern the relationships that have been discovered between joint angles in the animation. In the above example, the first principal component encompasses mainly the lateral shoulder and arm rotation, and how the neck and head rotates laterally in the opposite direction. There is very little wrist movement in this principal component. The second principal component deals more with the wrist rotations as the bat is raised and lowered and the shoulders tilt with respect to the horizontal. The third principal component involves very little wrist movement, but deals more with the twisting of the torso and raising of the arms. From these images, it is possible to imagine how combinations of these components could be used to describe a very complex animation. In fact, these principal components account for 73.6% of the variance in the source animation.

3.2 Using the low-dimensional space for animation synthesis

3.2.1 Method

It reasonably follows from the previous section, that by interacting with the scores for each of the principal components, new animations can be generated based on the source shot. As every point in the low-dimensional space created by PCA represents a pose, picking a new point in this space results in a new pose being generated, and a point moving through this space results in an animation being generated.

In order to test this assertion, it was decided to create a system that would allow for interactive generation of poses by moving a marker through the PCA space. For ease of interaction, it was decided that the PCA space would only be constructed from the first two principal components. Previously, the focus had been on the upper body of the skeleton only, but in order to allow entirely new animations to be synthesised, every joint in the skeleton is considered. Also, as only two principal components are being used to reconstruct the motion to allow easy user interaction, the best shots to use to demonstrate the technique are those that have the highest amount of variance captured by just two principal components.

One of the shots used in this project allows 67.3% of the variance in the skeleton to be captured with only two principal components, so this animation is used as the basis for synthesis. In order to synthesise animations, there are some parameters that need to be obtained. These are obtained whilst performing the PCA and are as follows:

L – a 107x2 matrix of the skeleton parameter loadings for the first two principal components.
M – a 1x107 matrix of the means of the skeleton parameters.
D – a 1x107 matrix containing the standard deviations of the skeleton parameters.

The user-supplied parameter is the score matrix **S**, which is a 2x1 matrix containing the score for each of the two principal components.

These matrices are used to calculate the skeleton pose by using the following equation:

$$\mathbf{P} = (\mathbf{LS})^T \cdot \mathbf{D} + \mathbf{M}$$

where the dot-multiplication operator produces the Hardamad product of the two matrices (i.e. pointwise multiplication of the matrix elements).

The resulting matrix **P**, is a 1x107 matrix which contains all of the resulting skeleton parameters – i.e. the root node translation along with the quaternion parameters for every joint in the skeleton. These parameters are mapped onto the skeleton to produce a pose. The scores in **S**, are provided by moving a marker around inside the two-dimensional principal component space: the x-coordinate of the marker provides the score for principal component one, and the y-coordinate provides the score for principal component two.

3.2.2 Experimentation

The original scores for a shot are plotted as a line in the principal component space as a guide for choosing a new path for the marker. The marker and plot are shown below, along with the resulting pose, with the marker placed on the plotted line. This means that the pose generated is one from the original animation (as close as the two principal component limit will allow).

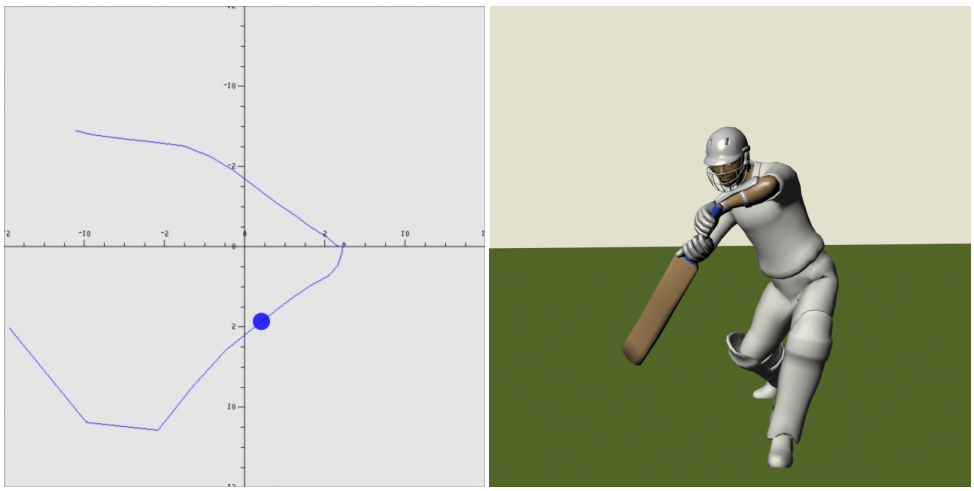

Figure 3.3 - Principal component space animation synthesis

By moving the marker around and observing the pose generated, it becomes clear that there are a number of possibilities for synthesising animations that extrapolate from the shot style given by the source shot. An important factor in this usability is that the poses are generated in real-time, allowing the user to move the marker around and observe the resulting behaviour of the character. From this experimentation, it was found that, in this particular shot, the score plot can be broken into three distinct areas, as shown in Figure 3.4.

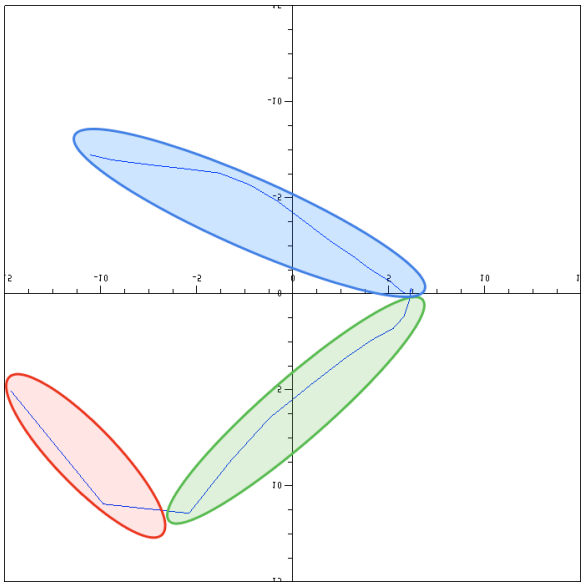

Figure 3.4 - Three distinct sections of the animation

The red section is the movement from the start pose in preparation to play the shot. The green part is the actual action for the striking of the ball, and it then turns a corner into the blue section that moves from the 'end' of the shot motion to the final rest pose.

3.2.3 Creating new animations

With this knowledge it is possible to predict the effects of altering the path of the marker through this space. For example, lengthening the green section might result in a more exaggerated backlift (moving the bat back in preparation to swing through the ball) and a much fuller follow-through after hitting the ball.

This is demonstrated in Figure 3.5. The green section of the plot is lengthened to produce an exaggerated backlift and follow-through. The figure shows the backlift and follow-through poses using the original scores (top-left and bottom-right) that lie on the blue line. The other two poses show the results achieved by picking scores away from the original values. The example of the extended follow-through is particularly striking.

The shoulders rotate further and the bat continues a long way past the original follow-through point. Interestingly, the head remains pointing in the almost the exact same direction, which is what you would expect as the character should be looking at the ball after having hit it.

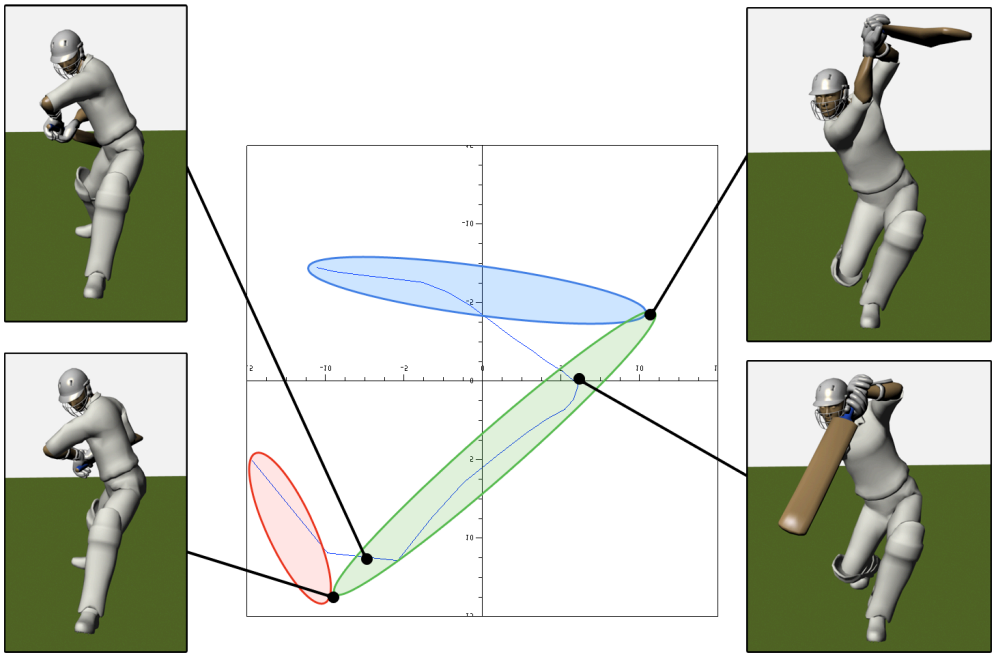

Figure 3.5 - Producing new animations by extrapolation

Another feature of this new shot is how the hips drop lower as the character bends into the exaggerated shot, and the right knee drops towards the floor. This is typical in cricket, as shown in the following comparison. The render on the left shows the top-right pose from Figure 3.5 from a different angle. The photo on the right shows an example of a similar shot being played by an international cricketer.

Figure 3.6 - A comparison between a synthesised pose and a real-life pose

The poses are unquestionably similar, except for the left arm configuration, and therefore the bat orientation, even though this pose was not in the original shot data.

This method can also be used to produce more 'toned down' versions of the same shot, and also to allow the shot to be 'abandoned' in the middle of it – as if the player suddenly decided to let the ball go instead of hit it. This second feature is potentially very useful in terms of increasing the interactivity possible when considering real-time applications, such as computer games. The scores required to produce these different shots are quite predictable and intuitive, and are demonstrated in the following plot.

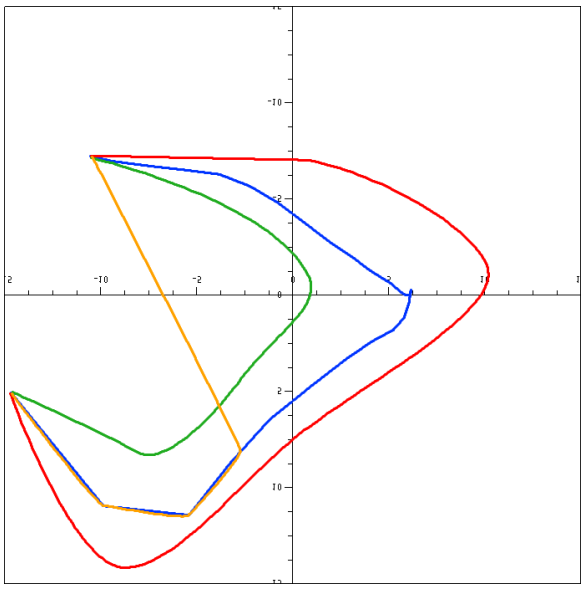

Figure 3.7 - Example plots of scores in principal component space for producing new shot versions

In the above figure, the original scores are shown in blue, whilst the red plot shows a path which leads to a shot similar to the synthesised one in Figure 3.5 and Figure 3.6. The red path resides 'outside' the observed, blue curve, indicating that it is a more exaggerated version of the shot – increasing the radius leads to an increase in 'shotness'. Similarly, decreasing the radius – as with the green curve – results in a lower 'shotness', or a more muted version of the shot. The orange plot demonstrates the required path to produce an 'aborted shot' animation. Here, the path is the same as the blue plot, until the point at which it suddenly diverges and heads straight towards the final rest pose.

These examples are demonstrated in the following figures. The figure colours correspond to the colours on the plot in Figure 3.7.

Figure 3.8 - A synthesised 'toned-down' shot (left: backlift; centre: hit point; right: follow-through)

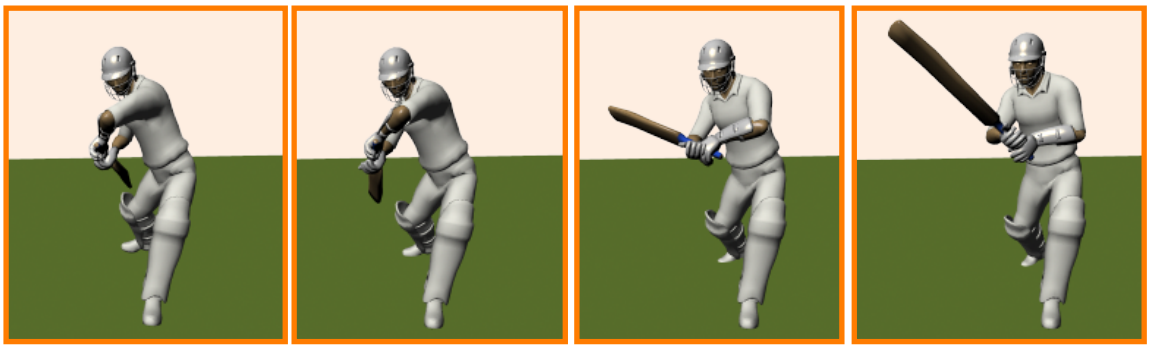

Figure 3.9 - A synthesised 'abandoned shot' animation

Figure 3.8 shows the three key points of the 'toned-down' synthesised shot: the backlift, hit point and follow-through. The backlift is much less pronounced in this example when compared to the backlift shown in Figure 3.5. The combination of the hit point and follow through show the shot to be much more constrained than the shots in Figure 3.5 and Figure 3.6.

Figure 3.9 shows several poses from an aborted shot animation. The first and second images show the poses according to the observed scores, so are the same as in the source animation. The third image shows what happens as the scores head away from the observed values and directly towards the final rest pose scores – the bat is lifted away from its previous trajectory, and the batsman begins to return to the rest pose.

3.3 Analysis

This method of synthesis does not allow for the creation of entirely new cricket shots, but does allow variations on a certain shot. These variations can be controlled in a predictable manner by observing the scores from the source shot and deriving new paths through principal component space based on those.

The examples shown above are not perfect animations. The main problems are that the feet tend to slide along the floor, and occasionally penetrate it or lift off it when they should be grounded. Also, the right hand has a tendency to come away from the bat handle when more extreme scores are input into the synthesis algorithm. It is important to remember, however, that this example was only using two principal components to reconstruct the animation, and as a result, the animation has some noticeable artefacts. These artefacts could be reduced by 'cleanup' of the resulting animation, such as fixing the foot-planting problems and the right hand coming away from the bat. Also, using a higher-dimensional principal component space would lead to better results with fewer problems requiring cleaning.

However, it has been successfully demonstrated that this is a potential alternative to motion-capturing variations of the same shot animation.

4 Conclusions and further work

4.1 Conclusions

The first aim of this project was to devise a way of adapting source 'hit' animations so that a new hit point can be reached whilst maintaining the style of the motion. This was achieved by implementing an IK algorithm that adapts parts of a source skeleton to ensure that the desired hit point is reached. The algorithm was demonstrated to work in real-time, and to produce results that are not only humanly possible but also humanly feasible, and most importantly natural-looking. The results show that reducing the freedom of the left wrist was important in preserving the style of the source shot when adapting the motion to a new hit location.

The IK/FK blending function used was designed to work independently of the content of a particular shot, so that new animations can be added to the application without having to perform any calibration work or pre-processing. Ultimately, the results show that a using a small number of animations with IK adaptation is enough to ensure that any reasonable hit point can be reached.

Secondly, the reduced dimensionality of PCA space was used to synthesise new animations by plotting a new path through the principal component space. This could be alterations to the shot such as more or less exaggerated versions of the shot, or allowing shots to be 'abandoned' or 'aborted' in the middle and returning to the final rest pose. This technique was shown to be able to produce poses which were not in the original animation but that were in keeping with 'cricket style', due to the important relationships between joint correlations being maintained. This means that the secondary aim of being able to reduce the requirement of recording vast quantities of motion-capture data was achieved. Also, the system allows the results of altering the principal component scores to be visualised in real-time, which aids the understanding of how the principal component scores can be changed to produce predictable results.

4.2 Further work

The chosen IK algorithm was found to work well, but it would be interesting to compare its performance with another algorithm, such as the various Jacobian methods, as discussed in section 1.7.2. In addition, altering the stiffness for the various joints in the IK chain could be used to produce variable results, particularly when considering the left wrist joint, which was defined to have 0-DOF in this application.

The IK system developed does not take into account the orientation of the end effector when solving. This means that the orientation is determined by the offset of the FK and

IK hit positions in some way. This relationship is unknown, but does tend to behave naturally. However, Jain and Liu (2009) describe a method of altering human animations in order to produce a certain response from an object being interacted with, such as a ball being kicked or headed. A similar idea could be looked at with regards cricket batting – it would be useful (especially for integration with a games engine) to be able to state "use this source animation, and hit this ball in this direction." The approach used by Jain and Liu does not work in real-time, however, so this leaves an interesting avenue open for exploration. Also, investigating the reverse of this interaction would be an interesting direction to explore – how does the impact of the ball on the bat affect the next few frames of animation?

When synthesising new animations using PCA space, the shots new shots are based upon one single shot. It would be interesting to see if it would be possible to create a space based upon several different shots, so that hybrid shots could be synthesised. Also, it may be interesting to create a PCA space based upon all observed poses, essentially creating a 'cricket batting space' which could then be used to synthesise any cricket shot imaginable. However, this would most likely require an alternative method of creating these spaces, in order to be able to reduce the required dimensionality more than PCA can. This is due to PCA only being able to detect linear relationships, so non-linear alternatives could be tested.

5 References

Arikan, O., & Forsyth, D. (2002). Interactive motion generation from examples. *ACM Transactions on Graphics , 21* (3), 483-490.

Buss, S. (2004). Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *University of California, San Diego, Typeset manuscript, available from http://math. ucsd. edu/~ sbuss/ResearchWeb* .

Choi, K., & Ko, H. (2000). On-line motion retargetting. *Journal of Visualization and Computer Animation , 11* (5), 223-235.

Fêdor, M. (2003). Application of inverse kinematics for skeleton manipulation in real-time. *Proceedings of the 19th spring conference on Computer graphics* , 203-212.

Gould, D. A. (2005). *Complete Maya Programming: An In-depth Guide to 3D Fundamentals, Gemometry and Modeling* (Vol. 2). San Francisco: Morgan Kaufmann Publishers.

Grochow, K., Martin, S., & Hertzmann, A. (2004). Style-based inverse kinematics. *ACM Transactions on Graphics , 23* (3), 522-531.

Ho, E. S., Komura, T., & Lau, R. W. (2005 йил November). Computing Inverse Kinematics with Linear Programming. *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology* , 163-166.

Lander, J. (1998 йил November). Making Kine More Flexible. *Game Developer Magazine (also available online: http://graphics.cs.cmu.edu/nsp/course/15-464/Spring07/assignments/jlander_gamedev_nov98.pdf)* , 15-22.

Liu, C., & Popović, Z. (2002). Synthesis of complex dynamic character motion from simple animations. *ACM Transactions on Graphics , 21* (3), 408-416.

Meredith, M., & Maddock, S. (2005). Adapting motion capture data using weighted real-time inverse kinematics. *Computers in Entertainment (CIE) , 3* (1), 5.

Shin, H., Lee, J., Shin, S., & Gleicher, M. (2001). Computer puppetry: An importance-based approach. *ACM Transactions on Graphics , 20* (2), 67-94.

Tolani, D., Goswami, A., & Badler, N. (2000). Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical models , 62*, 353-388.

Watt, A., & Policarpo, F. (2003 йил 1-Jan). 3D games: animation and advanced real-time rendering. 547.

Zordan, V., & Hodgins, J. (2002). Motion capture-driven simulations that hit and react. *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation* , 89-96.

6 Appendix – Partial Source Code Listing

```
/*
 *  CCDSolverNode.cpp
 *  Solves IK using CCD algorithm
 *
 *  Created by Tom Hicks on 10/07/2010.
 *
 */

//   USEAGE:
//   ikHandle -sol CCDSolverNode -sj joint1 -ee joint2

#include <maya/MIOStream.h>

#include <maya/MPxIkSolverNode.h>
#include <maya/MString.h>
#include <maya/MFnPlugin.h>
#include <maya/MObject.h>
#include <maya/MIkHandleGroup.h>
#include <maya/MFnIkHandle.h>
#include <maya/MDagPath.h>
#include <maya/MVector.h>
#include <maya/MPoint.h>
#include <maya/MDoubleArray.h>
#include <maya/MGlobal.h>
#include <maya/MDagPathArray.h>
#include <maya/MQuaternion.h>
#include <maya/MEulerRotation.h>
#include <maya/MFnIkJoint.h>

#define MAX_ITERATIONS 200
#define EPSILON 0.001
#define kSolverType "CCDSolverNode"
#define PI 3.141592653589793
#define DEG_TO_RAD 0.017453292519943
#define RAD_TO_DEG 57.295779513082320876798

// Class declaration
class CCDSolverNode : public MPxIkSolverNode {

public:
                     CCDSolverNode();
   virtual           ~CCDSolverNode();
   void              postConstructor();

   virtual  MStatus  doSolve();
   virtual  MString  solverTypeName() const;

   static   void*    creator();
   static   MStatus  initialize();

   static   MTypeId  id;

private:
   MStatus           doCCDSolver();

   int               getShotIndex();
   bool              printDebugNames;
```



```
};

MTypeId CCDSolverNode::id( 0x00333 );

// Implementation
CCDSolverNode::CCDSolverNode()
   : MPxIkSolverNode()
{
   setMaxIterations( MAX_ITERATIONS);
}

CCDSolverNode::~CCDSolverNode() {}

void CCDSolverNode::postConstructor()
{
   setSupportJointLimits(true);
   setPositionOnly (true);
}

void* CCDSolverNode::creator()
{
   return new CCDSolverNode;
}

MStatus CCDSolverNode::initialize()
{
   MGlobal::displayInfo("CCD::SolverNode::initialize");

   return MS::kSuccess;
}

// Return the name of the solver
MString CCDSolverNode::solverTypeName() const
{
   return MString( kSolverType );
}

// Solve function called by Maya
MStatus CCDSolverNode::doSolve()
{
   doCCDSolver();
   return MS::kSuccess;
}

// The actual CCD Solver
MStatus CCDSolverNode::doCCDSolver()
{
   MStatus stat;

   // Get the handle and create a function set for it
   MIkHandleGroup * handle_group = handleGroup();
   if (NULL == handle_group) {
      return MS::kFailure;
   }
   MObject handle = handle_group->handle( 0 );
   MDagPath handlePath = MDagPath::getAPathTo( handle );
```

```
MFnIkHandle fnHandle(handlePath, &stat);

// Get the position of the end_effector
MDagPath end_effector;
fnHandle.getEffector(end_effector);
MFnTransform tran( end_effector );
MPoint effector_position = tran.rotatePivot( MSpace::kWorld );

// Get the position of the handle
MPoint handle_position = fnHandle.rotatePivot( MSpace::kWorld );


// Get the start joint position
MDagPath start_joint;
fnHandle.getStartJoint( start_joint );
MFnTransform start_transform( start_joint );
MPoint start_position = start_transform.rotatePivot( MSpace::kWorld );

//put all Joints into array
MDagPathArray theJoints;
MDagPath toPop(end_effector);

//ignore the end effector
toPop.pop();

if (toPop.fullPathName() != start_joint.fullPathName()) {
//pop up the tree until we find the start joint
do {
   theJoints.append(toPop);
   MStatus popStat = toPop.pop();
   if (popStat != MS::kSuccess) {
      break;
   }
} while (toPop.fullPathName() != start_joint.fullPathName());
}

//add the start joint (for when we only have 1 bone / 2 joints)
theJoints.append(start_joint);

printDebugNames = false;
if (printDebugNames) {
   for (int g = 0; g < theJoints.length(); g++){
      MGlobal::displayInfo(theJoints[g].fullPathName() + "\n");
   }
}

double errorMag;
errorMag = (effector_position.distanceTo(handle_position));

//counters for iterations, and joint numbers
int i;
int numIterations = 0;

//while our error > 0, and we haven't done too many iterations
while (errorMag > EPSILON && numIterations < MAX_ITERATIONS) {
   numIterations++;
```



```
   int startIndex = 0;
   //MVector axis;
   //double  theta;
   //bool    getAAResult;
   bool    DOF[3];
   MEulerRotation initRot;

   //joint[0] is bottom, joint[n] is top joint
   for (i = startIndex; i < theJoints.length(); i++) {
      //get transform node for joint
      MFnTransform jointTransform (theJoints[i]);
      MFnIkJoint IKJoint(theJoints[i]);

      stat = IKJoint.getDegreesOfFreedom(DOF[0], DOF[1], DOF[2]);

      if (DOF[0] || DOF[1] || DOF[2]) {

         MPoint jointPosition = jointTransform.rotatePivot(MSpace::
             kWorld);
         stat = jointTransform.getRotation(initRot);

         //calculate vectors from joint to effector and handle
         MVector jointToHandle = handle_position - jointPosition;
         MVector jointToEffector = effector_position - jointPosition;

         //make a quaternion which rotates joint-effector to joint-
             handle
         MQuaternion qDelta(jointToEffector, jointToHandle);

         //rotate the joint to place effector along joint-handle
             vector
         jointTransform.rotateBy(qDelta, MSpace::kWorld);

         MEulerRotation newRot;
         stat = jointTransform.getRotation(newRot);

         //ensure DOFs are conformed to - Maya bug!
         if (!DOF[0]) newRot.x = initRot.x;
         if (!DOF[1]) newRot.y = initRot.y;
         if (!DOF[2]) newRot.z = initRot.z;

         jointTransform.setRotation(newRot);

         //calculate our new effector position for the next joint to
             work on
         effector_position = tran.rotatePivot(MSpace::kWorld);

         //recalculate the handle-effector error - if < EPISLON we
             know to stop iterating
         errorMag = (effector_position.distanceTo(handle_position));
         if (errorMag < EPSILON) {
            break;
         }
      }
   }

}
```

```
    errorMag = (effector_position.distanceTo(handle_position));
    return MS::kSuccess;
}


// Register the solver
MStatus initializePlugin( MObject obj )
{
    MStatus        status;
    MFnPlugin   plugin( obj, "Tom Hicks", "0.1", "Any");

    status = plugin.registerNode("CCDSolverNode",
                         CCDSolverNode::id,
                         &CCDSolverNode::creator,
                         &CCDSolverNode::initialize,
                         MPxNode::kIkSolverNode);
    if (!status) {
        status.perror("registerNode");
        return status;
    }

    return status;
}

MStatus uninitializePlugin( MObject obj )
{
    MStatus     status;
    MFnPlugin   plugin( obj );

    status = plugin.deregisterNode(CCDSolverNode::id);
    if (!status) {
        status.perror("deregisterNode");
        return status;
    }

    return status;
}
```

```
/*
 *  PCSynthNode.cpp
 *  Converts from PCs to Joint Angles
 *
 *  Created by Tom Hicks on 15/07/2010.
 *
 */

#include <maya/MGlobal.h>
#include <maya/MDagPath.h>
#include <maya/MSelectionList.h>
#include <maya/MFnTransform.h>
#include <maya/MVector.h>
#include <maya/MQuaternion.h>
#include <maya/MEulerRotation.h>

#include "PCSynthNode.h"
#include "matrix.h"
#include "CSVRow.h"


std::istream& operator>>(std::istream& str,CSVRow& data)
{
   data.readNextRow(str);
   return str;
}

MTypeId PCSynthNode::id( 0x00398 );

MObject PCSynthNode::pc1;
MObject PCSynthNode::pc2;
MObject PCSynthNode::dummyOut;

Matrix  PCSynthNode::PCs;
Matrix  PCSynthNode::SDs;
Matrix  PCSynthNode::Means;
vector<NameAttributePair*> PCSynthNode::Heads;

const double PI = 3.1415926535;
const double TWOPI = 2.0 * PI;

void printMatrix(const Matrix& mat, const MString& header) {

   MGlobal::displayInfo(header + "(" + mat.RowNo() + " x " + mat.ColNo()
       + ")");

   MString outString = "";

   for (int j = 0; j < mat.RowNo(); j++) {
      for (int i = 0; i < mat.ColNo(); i++) {
         outString = outString + mat(j, i) + ", ";
      }

      outString += "\n";
   }

   MGlobal::displayInfo(outString);
```



```
}

MStatus PCSynthNode::compute( const MPlug& plug, MDataBlock& data )
{
   MStatus stat;

   if (plug==dummyOut) {
      MDataHandle pc1data = data.inputValue (pc1);
      MDataHandle pc2data   = data.inputValue(pc2);

      double pc_1 = pc1data.asDouble();
      double pc_2  = pc2data.asDouble();

      Matrix score(2, 1);

      score(0, 0) = pc_1;
      score(1, 0) = pc_2;

      MDataHandle rotData = data.outputValue(dummyOut);

      Matrix pc_by_score_T(PCs * score);
      Matrix pc_by_score(~pc_by_score_T);

      int numCols = pc_by_score.ColNo();
      int i;

      for (i = 0; i < numCols; i++) {
         pc_by_score(0, i) = (pc_by_score(0, i) * SDs(0, i)) + Means(0, i
             );
      }

      for (i = 0; i < numCols; i++) {
         MString objName = Heads[i]->getName();
         MString objAttr = Heads[i]->getAttribute();

         if (objAttr == "translateX") {
            double x, y, z;
            x = pc_by_score(0, i);
            y = pc_by_score(0, i+1);
            z = pc_by_score(0, i+2);

            MSelectionList selection;
            selection.clear();
            selection.add(objName);

            MDagPath objPath;
            selection.getDagPath(0, objPath);

            MFnTransform objTransform(objPath);
            objTransform.setTranslation(MVector(x, y, z), MSpace::
                kTransform);
         }

         if (objAttr == "qx") {
            double x, y, z, w;
```

```
            x = pc_by_score(0, i);
            y = pc_by_score(0, i+1);
            z = pc_by_score(0, i+2);
            w = pc_by_score(0, i+3);

            MQuaternion setQuat(x, y, z, w);
            MEulerRotation setEuler;

            setEuler = setQuat;

            MSelectionList selection;
            selection.clear();
            selection.add(objName);

            MDagPath objPath;
            selection.getDagPath (0, objPath);

            MFnTransform objTransform (objPath);
            objTransform.setRotation (setEuler);

         }

      }

      data.setClean(plug);
      stat = MS::kSuccess;
   } else {
      stat = MS::kUnknownParameter;
   }

   return stat;

}

void *PCSynthNode::creator()
{
   return new PCSynthNode();
}


MStatus PCSynthNode::initialize()
{
   MStatus stat;

   MFnNumericAttribute nAttr;
   pc1 = nAttr.create( "PrinComp1", "pc1", MFnNumericData::kDouble, 0.0 )
       ;
   pc2 = nAttr.create("PrinComp2", "pc2", MFnNumericData::kDouble, 0.0 );

   MFnUnitAttribute uAttr;
   dummyOut = uAttr.create( "dummyOut", "dum", MFnUnitAttribute::kAngle,
       0.0 );

   addAttribute(pc1);
   addAttribute(pc2);
   addAttribute(dummyOut);
```



```
   attributeAffects( pc1, dummyOut );
   attributeAffects( pc2, dummyOut );

   MGlobal::displayInfo(MString("Resizing arrays"));

   PCs = Matrix(107, 2);
      MGlobal::displayInfo(MString("Resized PCs"));
   SDs = Matrix(1, 107);
      MGlobal::displayInfo(MString("Resized SDs"));
   Means = Matrix(1, 107);
      MGlobal::displayInfo(MString("Resized Means"));


   MString fileName;
   MString fileOpenCommand;
   int rowNum;
   CSVRow row;


   //GET Principal Components
   fileOpenCommand = "fileDialog -title (\"Open PCs\") -m 0 -dm \"*.csv\"
       -t \"Open CSV File\"";
   stat = MGlobal::executeCommand(fileOpenCommand, fileName, false, false
       );

   //we didn't select a file so don't continue
   if (fileName == "") {
      return MS::kFailure;
   }

   std::ifstream  file(fileName.asChar());

   rowNum = -1;
   while (file >> row) {
      rowNum ++;
      PCs(rowNum, 0)  = MString(row[0].c_str()).asDouble();
      PCs(rowNum, 1)  = MString(row[1].c_str()).asDouble();
   }
   file.close();


   //GET Standard Deviations
   fileOpenCommand = "fileDialog -title (\"Open SDs\") -m 0 -dm \"*.csv\"
       -t \"Open CSV File\"";
   stat = MGlobal::executeCommand(fileOpenCommand, fileName, false, false
       );

   //we didn't select a file so don't continue
   if (fileName == "") {
      return MS::kFailure;
   }

   std::ifstream  file2(fileName.asChar());

   while (file2 >> row) {
      for (int i = 0; i < row.size(); i++) {
         SDs(0, i)  = MString(row[i].c_str()).asDouble();
```

```
         }
      }
      file2.close();


      //GET Means
      fileOpenCommand = "fileDialog -title (\"Open Means\") -m 0 -dm \"*.csv
          \" -t \"Open CSV File\"";
      stat = MGlobal::executeCommand(fileOpenCommand, fileName, false, false
          );

      //we didn't select a file so don't continue
      if (fileName == "") {
         return MS::kFailure;
      }

      std::ifstream  file3(fileName.asChar());

      while (file3 >> row) {
         for (int i = 0; i < row.size(); i++) {
            Means(0, i)  = MString(row[i].c_str()).asDouble();
         }
      }
      file3.close();

      //hard-coded headings snipped out
         return MS::kSuccess;
   }
```