Pose-Space Deformation on top of Dual Quaternion Skinning.

by

Damien Murtagh,

Dissertation

Presented to the

University of Dublin, Trinity College

in partial fulfillment

of the requirements

for the Degree of

Master of Science in

Interactive Entertainment Technology

University of Dublin, Trinity College

September 2008

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Damien Murtagh

September 11, 2008

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Damien Murtagh

September 11, 2008

Acknowledgments

Firstly, I would like to thank my supervisor Dr. Ladislav Kavan for his insightful comments and direction during the course of my dissertation. His knowledge and guidance has been of great help to me. I would like to thank Barry Gormley for his help on the artwork side of the project. His contributions, including the creation of a number of character models, along with lending me his help and expertise in trying to master the various 3d animation tools, have been invaluable. I also pay tribute to the Trinity College department of computer science staff involved in the running of the Interactive Entertainment Technology course over the year, especially Dr. Steven Collins whose teaching and mentorship has been second to none. I also mention my classmates who, over the year, have showed tireless enthusiasm for the subject. I thank the many friends I have made during the course, who have no doubt made the year an enjoyable one. I thank all my close friends, especially my house mates Marc and Padhraig, who have been there for me when the going was tough. Lastly, I must pay the biggest gratitude to my family, especially my parents. Without their constant moral and motivational support it would not have been possible for me to pursue these studies.

Damien Murtagh

University of Dublin, Trinity College
September 2008

Pose-Space Deformation on top of Dual Quaternion Skinning.

Damien Murtagh,
University of Dublin, Trinity College, 2008

Supervisor: Ladislav Kavan

Real time character animation is an area of on going research which attracts much attention. The geometric method, skeletal subspace deformation has set the standard in real-time animation due to its simplicity and relative effectiveness. Other methods such as dual-quaternion skinning have become popular as a superior replacement to skeletal subspace deformation. Example based methods like pose space deformation, which have their roots in the motion picture industry have become popular too as a means of offering greater control over the animation than geometric methods. Because of the ever increasing processing power of modern gaming machines example based methods are now feasible in real time animation. The major contribution of this thesis is the implementation of a novel skinning algorithm based on pose space deformation on top of dual quaternion skinning. The algorithm is benchmarked against the current standard of example based methods which use pose space deformation on top of skeletal subspace deformation. The results show our method to be superior in the area of extreme joint rotations. The rotational regression method is also found to correct the problems of extreme joint rotations, but in a manner which is more complex than ours.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Character skinning is the process of deforming a virtual character model to simulate the motion of that character. A character model usually consists of a list of vertices, and a defined connectivity, which together form a 3 dimensional triangular mesh. The vertices have a number of attributes such as position, surface normal, bone weights and texture co-ordinate. There are a large number of digital content creation (DCC) tools available to artists, which can be used to create 3d character models. Skinning differs from rigid-body deformation in that the deformation is not applied uniformly across the entirety of the mesh, rather in a manner which allows individual body parts such as the arms, legs and head to move independently of each other.

The most popular skinning method used in real-time character animation at the moment is a skeleton based algorithm known as skeletal subspace deformation (SSD). SSD has a number of flaws which have been overcome by other more complex algorithms such as the skeleton based dual-quaternion skinning and the example based pose space deformation. The aim of this research is to implement the two aforementioned methods in tandem and study the properties of the resulting algorithm, which to our knowledge has not been tried before. It is hypothesized that the method will correct some of the rotational problems associated with current pose space deformation techniques, which will lead to a reduction in the number of poses needed and an associated increase in performance and decrease in memory requirements. Due to the lower number of poses needed the artist's workload should also be reduced as a result.

1.1 Assumptions of Prior Knowledge

This thesis assumes that the reader has a certain amount of prior knowledge in the areas of mathematics and computer graphics. On the mathematics side a familiarity with linear algebra, the application of matrices as a geometric transform, and the least squares method of solving linear systems is assumed. On the computer graphics side the reader is assumed to have some knowledge of the modern graphics pipeline. In particular, an appreciation of the difference between CPU and GPU implementations is assumed. Some prior exposure to pixel and fragment shader programming would also be

of help. Although some previous exposure to character animation techniques such as SSD and shape interpolation would also be an advantage, the document assumes no prior knowledge of these concepts and endeavors to explain them thoroughly from the ground up. Some knowledge of quaternions is also assumed as they are a commonly used construct in computer graphics.

1.2 Conventions Used

Scalar values are represented as lower case letters from the Greek or Latin alphabets, for example x. Vector quantities are represented by boldface lower case letters and are assumed to be column vectors by convention, for example $\mathbf{w}$. Matrices are represented by upper case letters in boldface, for example $\mathbf{M}$. When referring to quaternion values the letter q is used, and for dual quantities the dual unit ϵ is used. When referring to estimated quantities the hat symbol is used, such as $\hat{d}$.

1.3 Aims of the Thesis

The thesis has the following aims:

- To provide a summary of the currently available geometry based and pose based skinning algorithms.
- To implement a newly developed skinning method which is suitable for use in real-time animation, in particular in the video game industry.
- To provide a critical analysis of the newly developed technique. The main goal of this research is to investigate the properties of the new method, and to benchmark it against the current state of the art and discuss the results.

1.4 Organization of the Thesis

Chapter 2 provides a review of the current state of the art in skinning algorithms. Section 2.2 deals with skeleton based skinning methods, first describing skeletal subspace deformation and its associated artifacts. Then a number of advanced transform blending methods are discussed which solve these artifacts in various ways, but have some drawbacks. Finally dual-quaternion skinning is discussed as an artifact free replacement for SSD which is easy to implement. Section 2.3 provides a review of various example based skinning methods which interpolate between a number of example poses. These methods can also be used to correct SSD artifacts and can also capture some details, such as muscle bulge, which skeleton based methods cannot. The original implementation of pose space deformation is described in detail, along with some of the related mathematics and a number of improvements to the technique which have emerged since it was first published.

Chapter 3 discusses the design of the system. In section 3.1 on the technical specification the framework in which the project will run (Microsoft's XNA framework) is discussed in terms of its

animation capabilities. The hardware specification for the project is also described. The high level design is described in section 3.2 showing the major components of the system as well as the system's interface to the digital content creation tool.

Chapter 4 discusses the implementation of the system. Section 4.1 gives a detailed class and method level description of the system. Section 4.2 describes a number of important aspects of the implementation such as the definition of the pose space, the solving and application of corrections and the use of the inverse PSD method. Section 4.3 describes a number of other issues affecting the implementation, such as the choice of falloff value for the radial basis function and the generation of poses.

Chapter 5 presents our results. Section 5.1 concentrates on the correctness and simplicity of our method compared to others, and provides visual verification of our findings. Section 5.2 presents some performance data for our method and analyses our frame rate and memory usage data with respect to the alternative methods available. Section 5.3 provides some further analysis into why PSD on top of SSD does not correct rotational errors, and why our method does.

Chapter 6 provides the conclusion to our research and suggests some topics which may warrant further investigation.

Chapter 2

State of the Art

2.1 Skinning Overview

The skinning techniques presented in this chapter are broadly divided into two categories, namely skeleton based skinning and example based skinning. Section 2.2 deals with the skeleton based methods which use an underling skeleton structure attached to the character mesh to perform skin deformations. Section 2.3 describes the example based methods which perform character mesh deformation by interpolating between a number of example meshes which depict the character in different poses.

2.2 Skeleton Based Methods

2.2.1 Skeletal Subspace Deformation

This algorithm has appeared under many names such as skeletal subspace deformation, linear blend skinning, matrix palette skinning and enveloping, but for the remainder of this document will be referred to as skeletal subspace deformation or by the acronym SSD. The algorithm itself is unpublished but can be found in early works such as those of Magnenat-Thalmann et al. [17] from 1988, which focuses specifically on deformations of the hand.

If one considers the problem of animating an articulated body part such as an arm, then the simplest deformation model would be to treat the upper arm and forearm as two separate parts, and to apply a rigid transformation to each. This is illustrated in figure 2.1, which shows the independent rigid transformation of the forearm and upper arm. Transformations like this where no blending occurs are not suitable for realistic character animation because of the obvious discontinuities created at the joint, where the skin appears cracked or overlapping. A better solution is to define a system of bone transforms and perform a weighted blending. As before the vertices which are clearly located in the upper arm are transformed according to the upper arm matrix transform, and the vertices which are clearly located in the forearm are transformed using the forearm matrix transform. The vertices which are located in the elbow region are blended using a combination of matrices as illustrated in

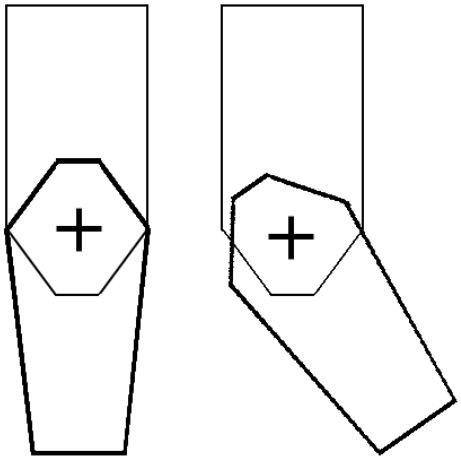

Figure 2.1: Rigid body blending in which the forearm and upper arm are essentially treated as two separate meshes and may become disjointed or display self-intersections when moved.

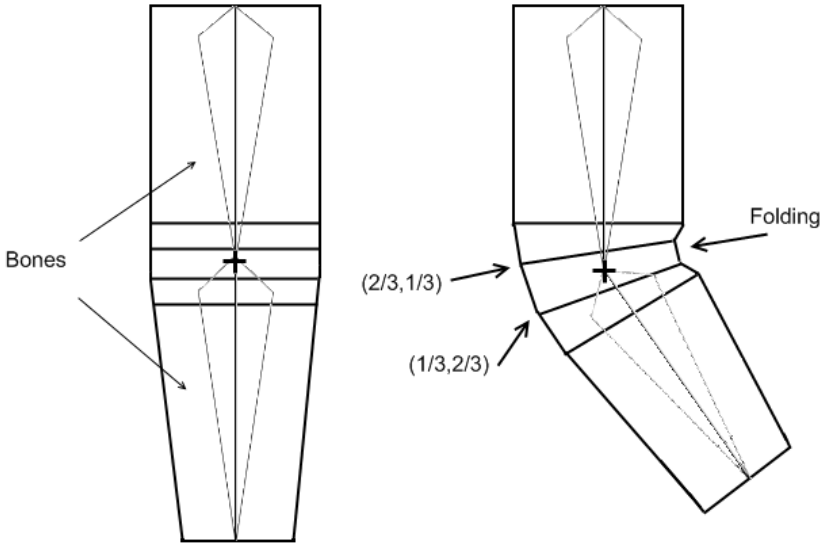

Figure 2.2: The forearm and upper arm bones are blended at the elbow using different weights, with the vertices closer to the upper arm blended (2/3,1/3) in favour of the upper-arm and the vertices closer to the forearm blended (1/3,2/3) in favour of the forearm. This simple example also illustrates the folding defect which is a consequence of using SSD.

figure 2.2.

Mathematically the SSD algorithm is expressed as the following equation:

$$\mathbf{v}_t = \sum_{i=1}^{n} w_i \mathbf{M}_{i,t} \mathbf{M}_{i,r}^{-1} \mathbf{v}_r \quad where \quad \sum_{i=1}^{n} w_i = 1, \; w_i \geq 0 \tag{2.1}$$

Here $\mathbf{v}_r$ is the original vertex position in what is known as the rest post and is specified in world coordinates. $\mathbf{v}_t$ is the vertex position after transformation at time t. There are n bones influencing the vertex $\mathbf{v}_r$ and there are n corresponding weights w_i which are convex and normalized to sum to 1. The matrix $\mathbf{M}_{i,r}$ transforms from the initial bones coordinate system to the world coordinate system for bone i in the rest pose. The matrix $\mathbf{M}_{i,t}$ is the world transform of the i^{th} bone at time t and changes in order to animate the mesh. The $\mathbf{M}_{i,r}$ term is often left out of discussions on SSD with a single matrix equivalent to $\mathbf{M}_{i,t}\mathbf{M}_{i,r}^{-1}$ being used instead. In practice this concatenated matrix is used to transform the vertices. The weights w_i are usually hand crafted by an artist using a DCC tool, often by setting envelopes of influence around each bone, by painting weight values on to the mesh surface or by setting the values manually. Some automated weighting systems have also been tried such as, 'Building efficient, accurate skins from example,'[19] which is discussed in section 2.3.

One serious limitation of the SSD technique is illustrated using the following simple example. Take two 2-dimensional matrices $\mathbf{M}_1$ and $\mathbf{M}_1$ which rotate the point $\mathbf{v}$ by different amounts. If the resultant transformed points $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly blended, then the blended point $\mathbf{p}$ lies on the line between $\mathbf{v}_1$ and $\mathbf{v}_2$ and therefore acquires a scaling factor and is no longer a pure rotation as shown in figure 2.3. The problem is most pronounced when the angle between the two rotations is 180 degrees. The ideal blended position is that of $\mathbf{p}'$ lies on the arc through $\mathbf{v}_1$ and $\mathbf{v}_2$ and is a pure rotation.

Because of the associativity of vector-matrix multiplication equation 2.1 may be re-written as:

$$\mathbf{v}_t = \left(\sum_{i=1}^{n} w_i \mathbf{M}_i \right) \mathbf{v}_r \tag{2.2}$$

This equation represents the standard method of implementing SSD where the weighted sum of matrices is calculated, then applied to the rest-pose position. The matrix $\mathbf{M}$ is typically a rigid transformation, that is it is composed of rotation and a translation components but without any scaling or skewing components. As an implication of figure 2.3 it can be seen that the weighted sum of two or more rigid transformations does not necessarily result in a rigid transformation. This statement is equally valid in three dimensions as it is in two dimensions. More formally, the set of rigid transformations is not closed under addition. The resultant artifact which is present in SSD skinned characters when the angle of blending between two adjacent bones approaches 180 degrees is known as the, 'Candy wrapper effect'. Shown in figure 2.4, the, 'Candy wrapper,' effect is a persistent problem in SSD.

Despite the drawbacks of SSD, it's use is widespread in the game industry. The obvious benefits are it's simplicity, which makes it well suited to GPU implementations, since the only operations needed are matrix multiplications and additions. Another benefit is the widespread availability of

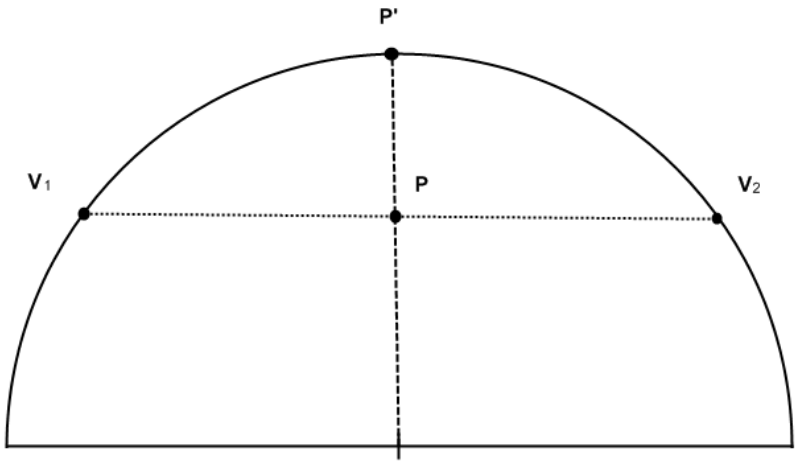

Figure 2.3: Blending of rotations in 2 dimensions.

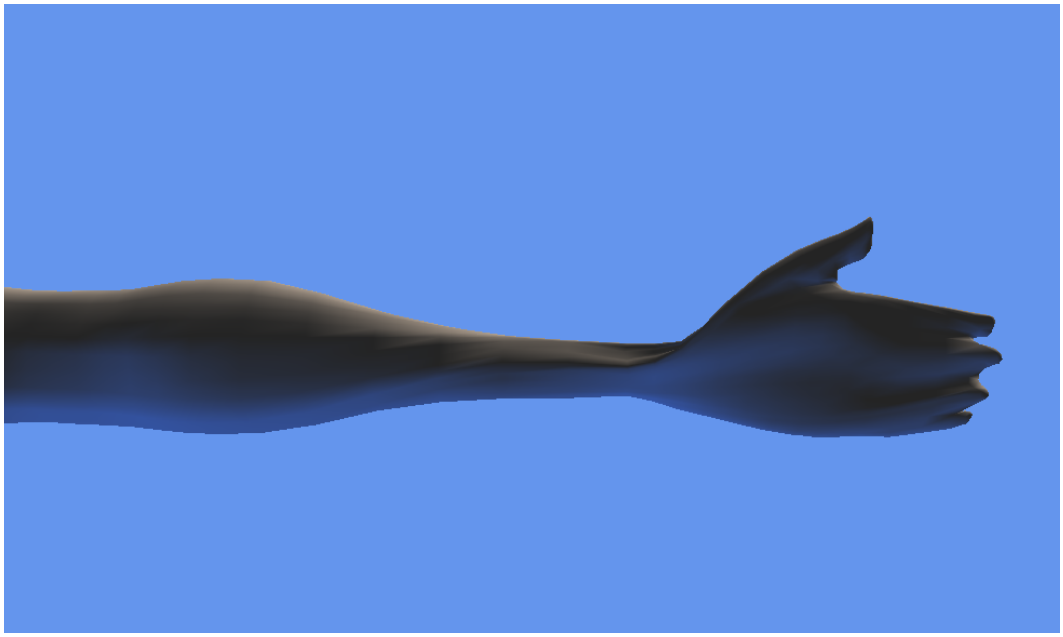

Figure 2.4: The candy wrapper effect

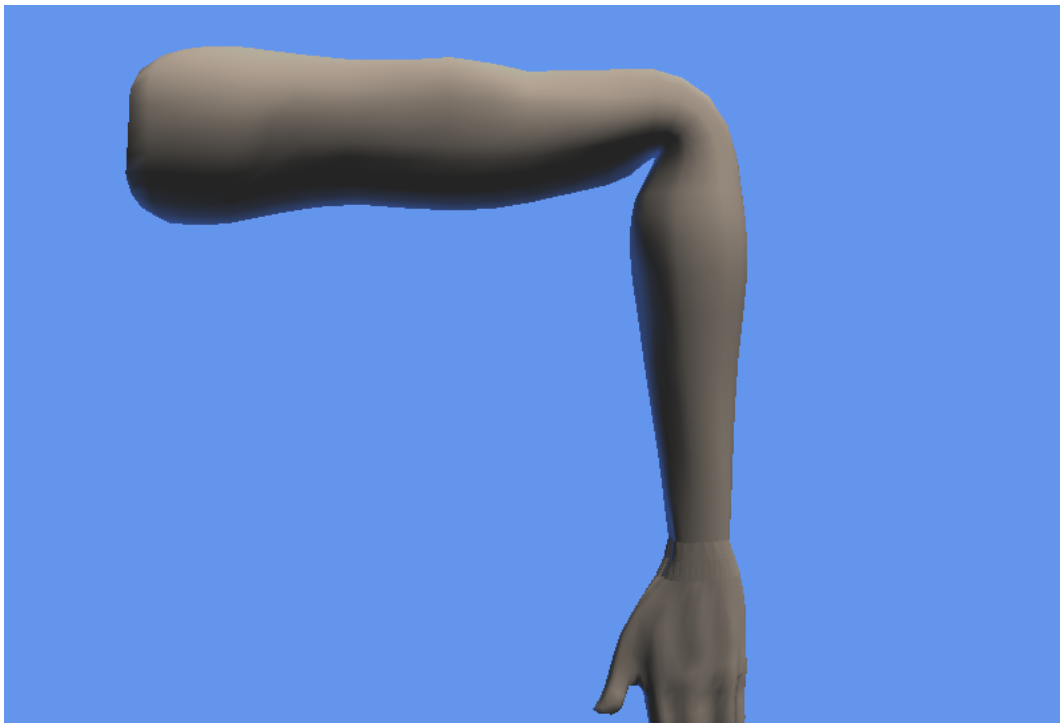

Figure 2.5: Folding at the elbow: The SSD equation 2.1 restricts the possible deformations to a particular subspace, which is equivalent to the linear hull of all the individual bone transforms of a particular vertex. In the case of the elbow joint the desired deformation for the inside of the elbow does not lie on this subspace, hence no amount of weight adjusting will produce the desired deformation. This can be a major source of frustration for artists using SSD, because re-adjusting the bone weights will sometimes fix a problem and other times can never fix it.

SSD in the common animation and modeling tools, allowing an intuitive work-flow between artist and programmer as the same skinning algorithm is being used at authoring-time and run-time.

2.2.2 Advanced Transform Blending

[10] describes a hardware implementation of a spherical blending algorithm in which the rotation element of rigid bone transforms are represented in their quaternion format. Their GPU implementation is an approximation of the well known spherical linear interpolation of quaternions (SLERP) algorithm, which provides a solution to the collapsing of joints associated with SSD, by blending the (quaternion,translation) pair in a manner which guarantees that the blended transform is a rigid one. Whereas the SLERP algorithm traces out an arc across a 4 dimensional hypersphere which smoothly blends rotations with respect to the angle of rotation, for efficiency this algorithm uses a linear blend of quaternions, which are then re-normalized so that they reside on the unit quaternion circle, and provides a good approximation of spherical blending. The article outlines, in Microsoft DirectX vertex shader assembly code, an efficient implementation of the algorithm. The problem with the blending of (quaternion, translation) pairs is that it is not co-ordinate system invariant and can produce poor results due to the center of rotation being fixed.

[7] describes a transform blending algorithm which also guarantees that the blending of rigid transforms results in a rigid transform, and is co-ordinate system invariant. The method is based around matrix exponentials and logarithms. The downside of the method is that in general transformations are not blended along the shortest path, which is a desirable property of any skinning algorithm.

2.2.3 Dual Quaternion Skinning

Dual-quaternion skinning is described in depth in [12]. Here rigid transformations are represented as pairs of quaternions of the form $q_0 + \epsilon q_\epsilon$ where q_0 is the ordinary quaternion part, q_ϵ is the quaternion dual part and ϵ is the imaginary dual unit, where $\epsilon^2 = 0$. Dual-quaternion skinning has the following properties which are desirable for any skinning algorithm:

- Blending of rigid transformations always returns a rigid transformation.
- Blending is co-ordinate system invariant.
- Blending always occurs along the shortest path.
- The algorithm is GPU friendly.

Dual-quaternion skinning is seen as a viable, artifact free alternative to SSD because of its simplicity of implementation and because it can use the same bone weight data as SSD without constraint. The performance of dual-quaternion skinning is also similar to that of SSD, with SSD being only slightly faster.

2.3 Example Based Methods

Example based skinning methods such as [16, 25, 23] have seen a significant amount of research effort in the last number of years. These methods have been popularized as a means of enabling artists to compensate for the inherent deficiencies of SSD as in [16] while still keeping within the performance limits of a real-time system. In other works such as rotational regression and eigenskin[25, 14], a number of mesh examples have been generated from a costly physically based simulation and used to train a corresponding real-time system to a certain accuracy which is comparable to the original physically based model. In yet another case, Kurihara and Miyata[15] provide the set of training examples from medical scanning data.

[26] introduces an improvement to SSD simulation called Multi-Weight Enveloping. In this technique a separate weight is used for each component of the joint transform. Weights are solved using a number of example meshes and the linear least squares algorithm. Both artist created and physically simulated example meshes are used. While the technique represents a simple and fast addition to SSD at run-time, the pre-processing stage can suffer from the problem of over-fitting, because of the large number of weights per vertex (12 for each joint of influence).

Mohr and Gleicher's, 'Building efficient, accurate skins from examples,'[19] describes a system of exporting sculpted character poses and their respective skeletal configurations from a modeling package to a real time system. Here the distinction is made between the authoring and computation phases of character skinning. Because offline character deformations are tightly coupled to the authoring phase a number of complex and sophisticated skinning approaches have arisen. Sub-structural skinning systems such as muscle and tendon simulations [27, 21] are cited as being commonplace in high end graphics applications such as those used in the film industry. The comparison is drawn between these and SSD based interactive systems which allow for fast computation of skinning but are notoriously difficult to author and exhibit deformation artifacts which are elaborated upon in section 2.2.1. A technique known as mesh animation is also described which involves the storing of static mesh definitions for each key frame of animation, and then interpolating between them at run-time. This simple method decouples the authoring and simulation steps and means that animation content may be created in any digital content creation (DCC) tool. The following limitations, however, mean that its popularity has declined in recent years:

- The possible poses must be known a-priori.
- The need to store many poses leads to a large memory requirement.
- It cannot be used to predict new poses which may be needed at run-time, such as would be required by many computer games.

The algorithm in [19] is an extension of SSD which incorporates arbitrary poses created in any DCC tool of the artists choosing. The examples presented are created in Maya, and are sampled at regular intervals. A new skeleton is then programmatically computed, with extra joints so as to represent the deformations which the original skeleton cannot. Fine tuning of these joint positions by

the user is allowed. In this new skeleton the extra joints are placed so as to fix the joint-collapsing artifacts around hinge joints (see figure 2.5 in section 2.2.1) and candy-wrapper effects seen around joint rotations (see figure 2.4). Adding extra joints may also allow the representation of muscle bulge, wrinkles in the skin or indeed any concievable deformation if the number of joints is equal to the number of vertices. This is impractical however, so a small number of strategically placed joints are added. To solve the candy-wrapper problem a single new joint may be added, which is computed to be half way between the rotations of the two existing joints using the spherical linear interpolation (SLERP) algorithm [22]. To represent muscle bulge new scaling joints may be added which are controlled by a nearby joint angle. E.g. the inverse of the cosine of the elbow joint angle may be used to control scaling of the bicep muscle.

Prior to calculating the vertex weights for the newly generated skeleton, the set of joints which influence each vertex is calculated. This calculation is performed by taking each of the posed versions of a particular vertex and applying the inverse transform for a joint to translate the cluster of points into the local space of that joint. This process is repeated for every joint in the skeleton and the joints which produce the smallest clusters of points are considered to be most influential to that particular vertex. Once the influence sets have been calculated the following SSD equation must be solved for the rest pose positions and joint weights (see section 2.2.1 for a further discussion on SSD).

$$\bar{\mathbf{v}}_e = \sum_{i=1}^{n} w_i \mathbf{M}_{i,e} \mathbf{M}_{i,d}^{-1} \mathbf{v}_d$$

Here $\bar{\mathbf{v}}_e$ is the position of a particular vertex after the skinning deformation has been performed for example pose e, n is the number of bones of influence of that vertex. w_i is the weight of influence of bone transformation i, $\mathbf{M}_{i,e}$ is the matrix transform of the i^{th} bone of example pose e, $\mathbf{M}_{i,d}^{-1}$ is the inverse transform of the dress pose (also known as the rest pose) for bone i and $\mathbf{v}_d$ is the rest pose position of the vertex. The solution aims to minimize the following:

$$\min \| \sum_{i=1}^{n} \bar{\mathbf{v}}_{e_i} - \mathbf{v}_{e_i} \|$$

Only the example poses and not the rest poses are known to the system, making the system bilinear, i.e. both weights and vertices must be solved. The solution is found using a process known as alternation. First the vertex positions are set and the system is solved for the weights, then the weights are set and the system is solved for the vertex positions. The process is then repeated until the solution converges which usually takes one or two iterations.

The technique presents impressive solving speeds and runs on existing hardware which supports SSD so it can be easily retrofitted into existing hardware accelerated systems. It can also be used in the process of re-rigging characters or as an interactive feedback loop for high end animation which may be to computationally intensive for real-time execution. Since it is an SSD based technique it is more efficient than some other example based animation techniques such as PSD/Eigenskin[16, 14], though it lacks the robustness of either. Another limitation, evident from the process of adding extra joint transforms, is the hard limit on the number of constant registers available to store bone transforms on the GPU. Prior to Direct3d 10 there were 256 4-tuple registers available to the vertex shader restricting the maximum number of bone transforms to 60, if stored in 4x4 matrix format. This restriction does not apply to direct3d 10 which provides 16x4094 constant registers [9].

2.3.1 Pose Space Deformation

Lewis's 2000 paper 'Pose space deformation'[16] describes a general approach to character skinning, which has been used in the motion picture industry. The approach has its origins in an earlier patented algorithm developed by the author at Industrial Light and Magic, and used in scenes in the motion picture films Jumanji and Casper. The basis of the algorithm lies in the combination of a skeleton based approach and a shape interpolation approach into a single unified approach to character skinning. By layering a displacement correction algorithm on top of SSD the algorithm targets the traditional animator work flow, allowing an artist to apply corrections to the traditional SSD algorithm in a manner which provides instant feedback. The simplicity of SSD is retained while the following enhancements are made possible:

- Muscle bulges can be sculpted by the artist and added to the real-time animation.
- Facial animations may be added.
- SSD artifacts may be corrected.
- Sophisticated skin deformations such as muscles, tendons and skin sliding over bone may be represented.

Previous shape interpolation approaches were based on the linear combination of the corresponding vertices in a number of key shapes $\sum_{k=0} w_k S_k$. These methods are widely used for facial animations and have the desirable characteristic that the target shapes are sculpted directly rather than manipulated through some more abstract parameterization. The major disadvantage of shape interpolation is that it is not suitable for use in capturing the range of movement of the regions of the body which rely on an underlying skeleton. Another drawback to using linear interpolation of shapes is that the interpolation is not always be as smooth as the artist would like. That is, while the interpolation guarantees parametric continuity (no discontinuities in the position of the interpolant over time) it does not guarantee geometric continuity (no discontinuities in the derivative of the interpolant over time). This can necessitate the sculpting of extra poses around the critical poses to create a smoother animation, imposes extra workload on the artist, and can lead to a larger than necessary memory footprint.

In order to facilitate the creation of a unified approach Lewis et. al. introduce the concept of a *pose space* to act as the interpolation domain of the displacement corrections which are applied to the model. The range of the interpolation is the desired displacement correction of the surface vertices. The *pose space* may be derived from the underlying skeleton, e.g. the angles which define the range of movement of a particular joint may be used. To illustrate this concept an arm animation which depends only on the shoulder and elbow joints may be visualized. A single angle may be used to represent the possible motions of the elbow joints while two angles are needed to represent the motion of the shoulder joint. The combined *pose space* for the elbow and shoulder joints would be three dimensional. [25] suggests the use of an axis angle representation as the basis for a *pose space*. The *pose space* may also contain a more abstract set of parameters such as the age, sex or emotional state

of the character. In [23] Sloan et. al. build on this idea by allowing the extrapolation of new models from a number of sculpted models in a particular abstract space (Section 2.3.3). One such abstract space exemplified by Sloan et. al. is a two dimensional space with one axis representing the sex of the model and the other representing the angle of rotation of the elbow.

Deformation corrections of a surface are expressed as a function of the underlying *pose space*. The system must allow artists to sculpt desired poses at arbitrarily spaced locations in the *pose space*. The problem is therefore one of scattered interpolation, which cannot be handled by a construct such as a spline which relies on the data points being regularly spaced within the *pose space*. [16] discusses the following possible approaches to scattered interpolation, which are further elaborated upon in section 2.3.2:

- Shepards method - Computes a weighted sum of the surrounding data points. The weight is set to an inverse power of the distance to the data point.
- Radial basis function (RBF) - Computes a weighted linear combination of the output of an RBF for each data point. An RBF is a function which depends only on the distance between two points.
- Energy functionals and non-convex methods - A method used in reconstruction of images, which provides piecewise continuity in the reconstituted function.

Radial basis functions are chosen for the interpolation and take the following form:

$$\hat{d}\left(\mathbf{x}\right) = \sum_{k=1}^{N} w_k \phi\left(\left\|\mathbf{x} - \mathbf{x}_k\right\|\right) \tag{2.3}$$

Here $\hat{d}\left(x\right)$ is the predicted value at location $\mathbf{x}$ in pose space. There are N data points $d(\mathbf{x}_k)$, N pose space locations $\mathbf{x}_k$, and N weights $\mathbf{w}_k$. The linear system may be arranged as:

$$\mathbf{\Phi w} = \mathbf{d}$$

And the solution may be computed using the standard least-squares formulation:

$$\mathbf{w} = \left(\mathbf{\Phi}^T\mathbf{\Phi}\right)^{-1}\mathbf{\Phi}^T\mathbf{d}$$

where $\mathbf{w}$ is a vector containing the weights $\mathbf{w} = \left(\mathbf{w}_1, ..., \mathbf{w}_k\right)^T$, $\mathbf{d}$ is a vector containing the input data points $\mathbf{d} = \left(\mathbf{d}_1, ..., \mathbf{d}_k\right)^T$ and $\mathbf{\Phi}$ is a $k \times k$ matrix containing the RBF values:

$$\mathbf{\Phi} = \begin{bmatrix} \phi\left(\left\|\mathbf{x}_1 - \mathbf{x}_1\right\|\right) & \phi\left(\left\|\mathbf{x}_1 - \mathbf{x}_2\right\|\right) & \cdots & \phi\left(\left\|\mathbf{x}_1 - \mathbf{x}_k\right\|\right) \\ \phi\left(\left\|\mathbf{x}_2 - \mathbf{x}_1\right\|\right) & \phi\left(\left\|\mathbf{x}_2 - \mathbf{x}_2\right\|\right) & \cdots & \phi\left(\left\|\mathbf{x}_2 - \mathbf{x}_k\right\|\right) \\ \vdots & \vdots & \ddots & \vdots \\ \phi\left(\left\|\mathbf{x}_k - \mathbf{x}_1\right\|\right) & \phi\left(\left\|\mathbf{x}_k - \mathbf{x}_2\right\|\right) & \cdots & \phi\left(\left\|\mathbf{x}_k - \mathbf{x}_k\right\|\right) \end{bmatrix}$$

Note that equation 2.3 relates only to the mapping of a pose space to a scalar value. In practice the step must be performed three times, once for each of the x,y and z dimensions, yielding weight values which are actually 3-dimensional vectors.

The next task is to choose a suitable radial basis function. Any nonlinear function $\phi(x)$ will interpolate the data and a smooth $\phi(x)$ will result in a smooth interpolation because a weighted sum of a continuous function is continuous. Lewis[16] chooses the Gaussian radial basis function $\phi_k(\mathbf{x}) = \exp(\frac{-(|\mathbf{x}-\mathbf{x}_k|)^2}{2\sigma^2})$ for the following reasons:

- It is continuous.
- It is reputed to be well behaved and has been used as the basis of RBF interpolation in previous neural network research[8].
- It approaches zero far away from the poses.
- The falloff σ is selectable.

The falloff value σ is exposed to the user and, though its meaning may not be immediately obvious, allowing an animator to interactively change it should facilitate changes to the amount of influence given to each pose.

The following is an outline of the steps involved in the *pose space deformation* algorithm:

1. **Define the pose space.** That is the *pose controls* and the degree of freedom of each control. Pose controls may be the relative angles between the relevant joints, such as the angle of bend of the elbow or some set of abstract manipulators such as the muscle bulge or facial attributes such as 'sadness'. This configuration makes up the input to the interpolater.

2. **Sculpt.** The artist positions a set pose controls and sculpts the deformation for each pose. The artist also sets the Gaussian falloff σ for each pose. The value can be set uniformly across all poses or individually configured for each pose.

3. **Calculate $\vec{\delta}$(pose).** For each pose, find the vertices which are different in the edited pose mesh than in the equivalent SSD deformed mesh. Calculate the difference in position $\vec{\delta}_i$ for each of the vertices to be corrected. The difference should calculated between the SSD transformation of the rest pose vertex v_i^r and the posed vertex v_i^p, where w_j and $\mathbf{M}_j$ are the SSD weights and transforms for joint j respectively:

$$\vec{\delta}_i = \mathbf{v}_i^p - \left(\sum_{j=1}^{n} w_j \mathbf{M}_j \right) \mathbf{v}_i^r \tag{2.4}$$

4. **Solve.** When enough sculpted poses are ready the interpolation problem must be solved for each of the vertices which is different from the underlying SSD deformation in any of the sculpted poses. For each of these vertices, find the poses which contribute to the displacement of this vertex, and the equivalent *pose controls* configuration. Solve the RBF weights $\mathbf{w}$ for the vertex

using the least squares method as described previously. The paper notes that it is possible to pre-calculate $\mathbf{w} = \left(\mathbf{\Phi}^T\mathbf{\Phi}\right)^{-1}\mathbf{\Phi}^T$ because its value does not change between vertices which are effected by the same poses. The pre-calculation would avoid repeating a costly matrix inversion for each vertex, but the optimization is largely irrelevant because the step is preformed prior to runtime. It might, however, be relevant if it is being used in a feedback loop for artists and the evaluation of changes is taking too long to process for a model with a large number of vertices.

5. **Synthesize.** When the model is moved to an arbitrary pose the location in pose space must be calculated. For each deformed vertex, use the previously computed RBF weights $\mathbf{w}$ for that vertex to calculate the interpolation of the displacement offset at that point in pose space using equation 2.3. The interpolated displacement must then be added to the SSD deformed vertex position.

6. **Evaluate and repeat.** The model now interpolates through all of the defined deformations. The extent of each pose around its position in pose space can be visualized and changed using σ and then re-tested. Lewis[16] recommends allowing axis aligned manipulation of the σ value and mentions using the Mahalanobis distance instead of the euclidean distance, as input to the RBF. This is discussed in section 2.3.2. New poses may also be added to fix problematic areas.

One aspect which is not referred to is the handling of surface normal during the interpolation process. Obviously if displacement corrections are being applied to the vertex positions of a model some form of correction must also be applied to the corresponding vertex normals, otherwise the lighting calculations will be incorrect. While this is not explicitly mentioned in Lewis's work [16] it will be considered in this work in section 4.

Though pose space deformation is not GPU friendly a GPU implementation is discussed in [24]. The paper describes an implementation of *weighted* PSD (see section 2.3.3 for an explanation of *weighted* PSD), where the SSD and PSD weights are packed into a texture and passed to the fragment shader. The fragment shader calculates the vertex displacements which are stored to a texture for use in the next rendering pass. The next rendering pass applies these displacements in the vertex shader. The algorithm is summarized by:

- **First pass:** Calculate the displacements to the vertex position in the fragment shader.
- **Second pass:** Calculate the displacements to the vertex normal in the fragment shader.
- **Third pass:** Apply the corrections in the vertex shader.

Principle component analysis of the pose space data is also discussed to reduce the domain of the computation. The paper reports a 20× performance gain over the CPU implementation.

2.3.2 Scattered Interpolation and Related Mathematics

In PSD the deformation of a surface is expressed as a function of the pose of an underlying skeleton or abstract set of parameters. It is necessary to directly sculpt the desired deformations at various

points in the parameter space which are not guaranteed to be evenly spaced. This problem of fitting a function to a number of arbitrarily spaced data points is known scattered interpolation and a number of candidate solutions are discussed in [16, 20, 23]

Inverse distance weighting (Shepards method) is often used for scattered interpolation. It provides a weighted sum of the surrounding data points, with the weight being an inverse power of the distance to a particular data point:

$$\hat{d}(x) = \frac{\sum w_k(x) d_k}{\sum w_k(x)}, \quad w_k(x) = \frac{1}{\|x - x_k\|^p}$$

Shepards method has some drawbacks though. Firstly, as the function moves far away from the data points the weights converge to approximately the same value, meaning that the interpolated value is the average of all the data points. Ideally one would want the interpolation to decrease to zero far away from the data points. Secondly the derivative of the interpolated function is zero at the data points which may not be desired.

Thin plate splines are another common choice for the RBF kernel function where:

$$\phi(x) = x^2 \log(x)$$

Thin plate splines have a more global nature than Gaussian RBF, where a small change to one of the data points will effect the coefficients corresponding to all the other data points. Like the Gaussian function thin plate splines are smooth. They also have the advantage of having no free parameters which need manual tuning. They are considered for use in 'Shape by Example'[23].

'Shape by Example' [23] uses the cubic B-spline cross section as its RBF kernel, because of its compact support which aids extrapolation, i.e. the function reduces to zero as the distance to the pose becomes greater (unlike the thin plate spline). The cubic B-spline is a generalization of the Bezier curve. The Bezier curve defines a curve in terms of a set of control points, passing through the first and last data points but not necessarily the others. The cubic B-spline cross section drops to zero at twice the distance to the nearest example, as opposed to the Gaussian function in which the falloff is selectable.

For a heterogeneous interpolation domain the *Euclidean distance* is sensitive to scale. For example the pose space might consist of a number of joint angles specified in degrees and a 'happiness' factor. The joint angle values may vary from -180 degrees to +180 degrees while the happiness parameter varies from 0.0 to 1.0. This gives greater weight to the 'happiness' parameter, by virtue of its shorter distance in Euclidean space, and may produce some unexpected results. To account for this type of scenario the *euclidean distance* may be generalized in a number of ways, which are well known in statistical analysis. The *normalized Euclidean distance* or the more general *Mahalanobis distance* may be used to take into account the principle axes along which the data lies. The *Mahalanobis distance* is computed using the covariance matrix of the available data points and may be used to correlate data which is distributed along an arbitrary axis or to eliminate outliers from a set of data. The distance measure has been used in research into neural networks[8] as a means of rotating and stretching a

radial basis function and could be used to augment radial basis interpolation in the context of character skinning. The *Mahalanobis distance* is similar to weighted PSD (discussed in section 2.3.3) in that both provide a scaling of the axes which aims to improve the results of PSD. The two are not mutually exclusive though and it is conceivable that they could be used in tandem.

2.3.3 Pose Space Deformation Variations and Improvements

Kurihara and Miyata[15] use a set of medically scanned images as the basis for their hand deformation system. Their images were obtained using computed tomography (CT) scans which show the bone structure and skin shape for a number of hand positions. Their work is comprised of three steps. Firstly the link structure of the hand are estimated using the scanned images. The joint centers and joint angles are computed for each pose. Some simplifications are made, such as replacing the radius and ulna bones of the forearm with a single bone which provides roughly the same functionality. The number of bones in the complex carpel bone system of the wrist is also reduced for simplicity. Secondly a set of model meshes are constructed from the scan data for each pose. The meshes are generated to be topographically consistent to easily allow for shape interpolation. To ensure topographical equivalence of the meshes the original base mesh is deformed using SSD into the pose configuration to create an approximate mesh for a particular pose. A number of equivalent feature points are then manually specified on the approximate mesh and the CT scanned mesh. The approximate mesh is then deformed to the CT mesh using a Radial Basis Function at the feature points. A final step then fits the vertices of the RBF transformed mesh to the CT ground truth by moving each vertex to its nearest neighbor in the ground truth.

During synthesis the RBF interpolation phase has a couple of differences and improvements from the original PSD. Firstly the inverse of the SSD calculated transform is applied to the pose mesh vertices prior to calculating the displacements rather than applying SSD to the base mesh. The comparison is then made between the inverse SSD transformed pose mesh and the base mesh so the order of operations changes from:

```
PSD(SSD(Model), Corrections)
```

to

```
SSD(Model + PSD(Corrections2))
```

The following equation shows the calculation of the offset $\vec{\delta}$, first by applying the SSD inverse transform to the posed vertex v_i^p and then finding the difference between this transformed vertex and the rest pose v_i^r, in contrast to equation 2.4 which performs the reverse:

$$\vec{\delta_i} = \left(\sum_{j=1}^{n} w_j \mathbf{M}_j \right)^{-1} \mathbf{v}_i^p - \mathbf{v}_i^r \tag{2.5}$$

Figure 2.6 illustrates the inverse SSD on some mesh examples. [28] provides a discussion on why this *inverse PSD* method provides better results than regular PSD. The paper presents a simple example of

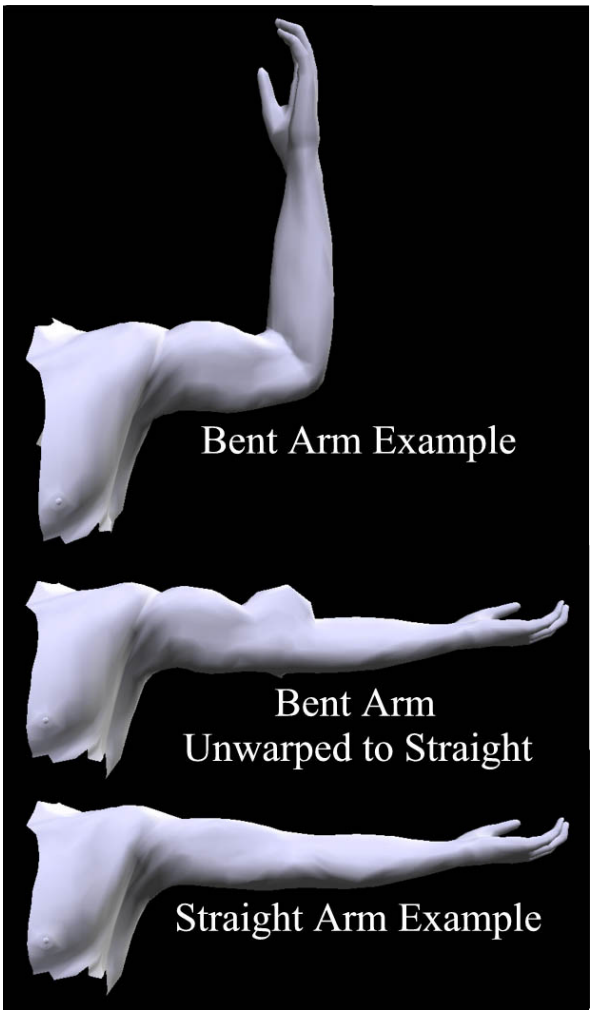

Figure 2.6: Pose examples transformed via Inverse SSD to the rest pose. *(Image courtesy of Michael Cohen [23].)*

where *inverse PSD* provides a better interpolation of a deformed posed vertex over an elbow bending animation. This also has the advantage of allowing the PSD Corrections to be applied on the CPU and then standard SSD to be applied on the GPU.

Another improvement used in [15] is the addition of a weighting factor in the calculation of the pose space distance to allow a small number of poses to be used to generate a larger number of deformations. The principle observation behind this improvement is that the movement of a joint which is far away from a particular vertex has little or no bearing on the correction displacement which should be applied to that vertex. The SSD weights are used when calculating the distance in pose space to ensure that joint movements which have little effect on a vertex have little effect on the pose space distance calculated for that vertex. Thus the pose space distance between two poses x^1 and x^2 becomes:

$$d\left(p,q\right)=\sum_{i=1}^{n}w_{i,v}\left(x_i^1-x_i^2\right)^2 \tag{2.6}$$

where n is the number of degrees of freedom in the pose space, $w_{i,v}$ is the SSD weight of the joint corresponding to i for the vertex v, x_i^1 is the i^{th} component of the pose space location x^1 and x_i^2 is the i^{th} component of the pose space location x^2. This essentially applies a weighting to the pose space axes which is dependent on the vertex in question, as opposed to using the *Mahalanobis distance* (Section 2.3.2) which provides a weighting to the axes which is uniform over all the vertices, but is dependent on the spread of data points within the pose space.

Eigenskin[14] also introduces a number of improvements to pose space deformation. The inputs for this algorithm also consist of a skeletally rigged hand model and a number of key poses, though this time the key poses are computed using a physically based finite element model which takes several hundred CPU hours to execute. Like [15] the vertex displacements are mapped back to a neutral character pose akin to equation 2.5. By using an error-optimal *eigendisplacement* basis for pose corrections the authors develop a method which is suitable for hardware acceleration on modern graphics hardware.

Eigenskin introduces the notion of *joint support* to leverage the fact that localized changes to the pose configuration of a model result in localized deformations. For example, the bending of the index finger of a hand model does not result in any significant deformations to that little finger of that model. The set of vertices forming the *joint support* for each joint is computed using the output of the finite element model, and the authors find this set not to be equivalent to the set of influence joints defined by the SSD weights. Principle component analysis (PCA) is then used to create an orthonormal basis for the displacements termed *eigendisplacements*. The PCA solution is formed for each joint where the data correlating pose configurations to vertex displacements within the *joint support* for that particular joint is reduced. In this manner a large number of poses may be compressed to k principle components where k is less than the original number of poses. The *eigendisplacement* co-ordinates are then interpolated using RBF's in a manner similar to Lewis's method[16] which is described in section 2.3.1. Vertex shader synthesis of now becomes a simple linear weighted combination of the interpolated *eigendisplacements* with the maximum number of *eigendisplacements* per vertex limited by the per-vertex data constraints.

The results produced are claimed to be marginally slower than SSD in terms of execution time. A comparison of the correctness of an unseen pose synthesized using Eigenskin to the original finite element model shows impressive results. With a single *eigendisplacement* base per joint the relative error is shown to be 16% of that of the equivalent SSD deformed model. Using 5 eigendisplacements the relative error drops to 6.5%. One limitation of eigenskin is the fact that eigendisplacement bases are constructed from a single joint and do not capture non-linear deformations which are dependent on the configuration of a combination of joints.

In shape by example[23] a shape interpolation technique is presented which allows the efficient run-time interpolation between multiple forms of an model. The method is demonstrated to create multiple distinct models from a few example models as a cheap method of adding variety to a crowd scene. It is also used to create smooth skinning deformations by blending examples. The method has a number of differences from Lewis's PSD[16]. Firstly, whereas Lewis solves a linear system of RBF's per vertex, shape by example solves per example mesh, which leads to reduced solution complexity provided there are fewer example meshes than vertices (which is always the case in practice). Secondly the RBF solution is augmented with a linear hyperplane which is intended to capture linear changes between the examples. An abstract pose space is defined using adjectives such as young versus old and male versus female to define the axes.

The notion of a *cardinal basis* is introduced with one basis function associated with each example. The *cardinal basis* consists of a weighted sum of RBF's and a low order polynomial. The linear part provides an overall approximation, while the RBF part fits the interpolation to the examples. The coefficients which define the linear hyperplane are first estimated using least squares fitting for each example pose. The RBF weights are then fitted to account for the residuals in the *cardinal bases*. Rather than fitting for each vertex as in PSD[16] N RBF's are fitted to each *cardinal base* of which there are N, where N is the number of example poses. This results in $N \times N$ weights which must be solved and used during synthesis, which is significantly lower than the number of weights used in PSD. For their choice of RBF a cross section of a cubic B-spline is used, for its compact support, that is $\phi(x)$ decreases as x gets bigger. This is said to aid extrapolation because as one moves further away from the location of a pose in pose space the the interpolation reduces to the underlying linear basis. Smoother thin-plate interpolations were also considered. For more detail see section 2.3.2. The technique is layered on top of SSD in a manner similar to Eigenskin [14] and [15] which are described above. The examples are transformed back to the base pose using inverse SSD and then blended.

In shape by example[23] solving takes a fraction of a second due to the smaller number of weights to be solved. This allows the method to be used in an interactive feedback loop for artists. Runtime memory requirements are also reduced because of the lower number of weights.

[25] introduces a system of corrections called rotational regression which is layered on top of an SSD system. Rotational regression is based on the mapping of an underlying skeleton pose to a system of deformation gradient predictors, consisting of the rotational, shearing and scaling errors for each triangle in a mesh. It is therefore a more sophisticated system than simply mapping mesh poses to mesh displacements. The system is trained using a number of example meshes, and the resulting mapping is simplified using dimensionality reduction. The paper also discusses a GPU implementation

of the algorithm. The results compare favorably to PSD in terms of the correctness of the predicted mesh. The technique is shown to provide better prediction of unseen poses than either the PSD or Eigenskin methods [16, 14], in certain instances where the rotational error is very pronounced. It also compares favorably to SSD in terms of speed of execution, with performance shown to be less than a factor of 2 times slower than SSD. Also during the dimensionality reduction stage a value may be set to control the trade-off between performance and correctness. It is however a much more complex algorithm to understand and implement than SSD. Also unlike PSD or Eigenskin it does not reproduce the training poses exactly.

Chapter 3

Design

For this dissertation the intention is to implement a version of *pose space deformation* which would be suitable for use in video games or other interactive application which require real-time character animation. Using Lewis's 2000 paper [16] as a base point, the project will implement radial basis function solver to interpolate pose space corrections. The paper implements displacement corrections on top of SSD, which will be reproduced by this work, along with the dual-quaternion based equivalent for the purposes of comparison.

3.1 Technical Specifications

The work will be implemented using Microsoft's XNA framework 2.0 on an Intel dual-core 1.86Ghz processor Windows XP machine with 2Gb ram and an Nvidia GeForce 8600 GTS graphics card. XNA is a games development toolkit which runs under the .NET framework and is aimed at students, hobbyists and independent game developers. It provides a set of managed code development libraries which support the common set of tasks necessary for games development such as graphics, animation and networking. The XNA framework 2.0 provides only partial support for key frame animation as standard. The class framework for supporting the input of skeletal information exists but no content pipeline importer is shipped with the framework. The framework supports the importation of the following elements of a skeletally rigged model:

- The initial rest pose mesh definition containing vertex positions, vertex normals and mesh connectivity (a list of indices connecting the vertices).
- A skeleton tree structure representing the skeleton in the rest pose. This consists a list of transform matrices corresponding to the bone transforms with respect to their parent bone, the index of the parent bone for each bone, and a single root bone index.
- A list of bone indices for each vertex, which shows the particular bone transforms which are relevant to that vertex.

- A list of bone weights for each vertex, which corresponds to the list of bone indeces and determines what weighting is to be applied to each bone.
- A key-framed sequence of skeleton definitions which is used to store the skeleton transform matrices at each timestep.

Support is available for the .fbx and .x file formats. There is no support shipped with the framework for motion synthesis through the vertex shader. There are, however, examples for both an animation content pipeline and an SSD vertex shader available in the 'Skinned Model' example on the XNA Creators Club website [6], along with the more advanced 'XNAnimation Library' available on the Codeplex website[1]. A HLSL example of skinning using dual-quaternion blending, which runs under XNA, has been written by Alexander Jhin[4] and will need to be adapted to make it suitable for CPU execution.

Models are to be authored using the 3D Studio Max 9 modeling tool, which provides a huge number of modeling and animation features, including SSD skinning (via the skin modifier), sub-structural skinning features such as muscle bulges and tendons (via the physique modifier), key frame animation and model re-sculpting. Static meshes and rigged models will be exported to the directX model file format (.x), for use in XNA, using the third party *Panda DirectX Exporter* [3] plug-in for 3D Studio Max, which is available under a freeware license. The Panda exporter provides facilities for the export of mesh definitions, materials, animations and bones as well as providing some mesh optimization options.

The linear least squares solution segment of the radial basis solution will require an implementation of the linear least squares algorithm, which specifically requires the inversion of potentially large matrices. Rather than writing code to do this an 'off-the-shelf' linear algebra toolkit will be used. The *Extreme Optimization Numerical Libraries* for .NET provide such a toolkit, including a number of options for matrix inversion.

The implementation will initially target the CPU, with RBF solving and the application of displacement corrections being applied on the CPU. This is because the original pose space deformation methods are not suitable for GPU implementation. Some GPU implementations have been explored in the literature[24], and these will be discussed at least in theory. See section 2.3.3 for a review of the current state of the art regarding GPU implementations.

3.2 High Level Design

Figure 3.1 shows a high level overview of the system. Here a rigged character is created and exported to a .x file in 3d studio max. A number of pose meshes are also created and exported to the directX format. All the model files are imported through the content import pipeline to the XNA application. The application loads these files as content at startup time, and, once all poses have been loaded, the solution step is performed to solve the RBF weights for position and normal corrections. After this initialization step is complete the update/draw loop uses the PSD Engine to apply corrections

periodically, according to the current state of the animation. The deformed model is then rendered on the GPU and shown on screen.

With regard to the individual components of the application, the 'Content Importers' component consists of the default *XNA Model Pipeline* which is packaged with the XNA 2.0 distribution, and the *Skinned Model Pipeline* which is to be obtained from a third party source as discussed in section 3.1 and should not need any significant modification. The 'Application Main' component is to be written from scratch, and will contain the methods `Initialize()`, `LoadContent()`, `Update()` and `Draw()`, which are inherited from the XNA *Game* class and are commonplace in graphical applications. The 'PSD Engine' component contains the capabilities necessary for the parsing the vertex, normal and weight data from the model files, performing linear blend skinning and dual quaternion skinning, storing the data associated with each pose, calculating RBF values and RBF weights and applying the RBF corrections during synthesis. The only third party component within the 'PSD Engine' component is the *Extreme Optimization Numerical Libraries* [3] least squares solver. The 'GPU' component of the architecture performs the rendering tasks such as perspective projection and lighting as is standard in 3D applications.

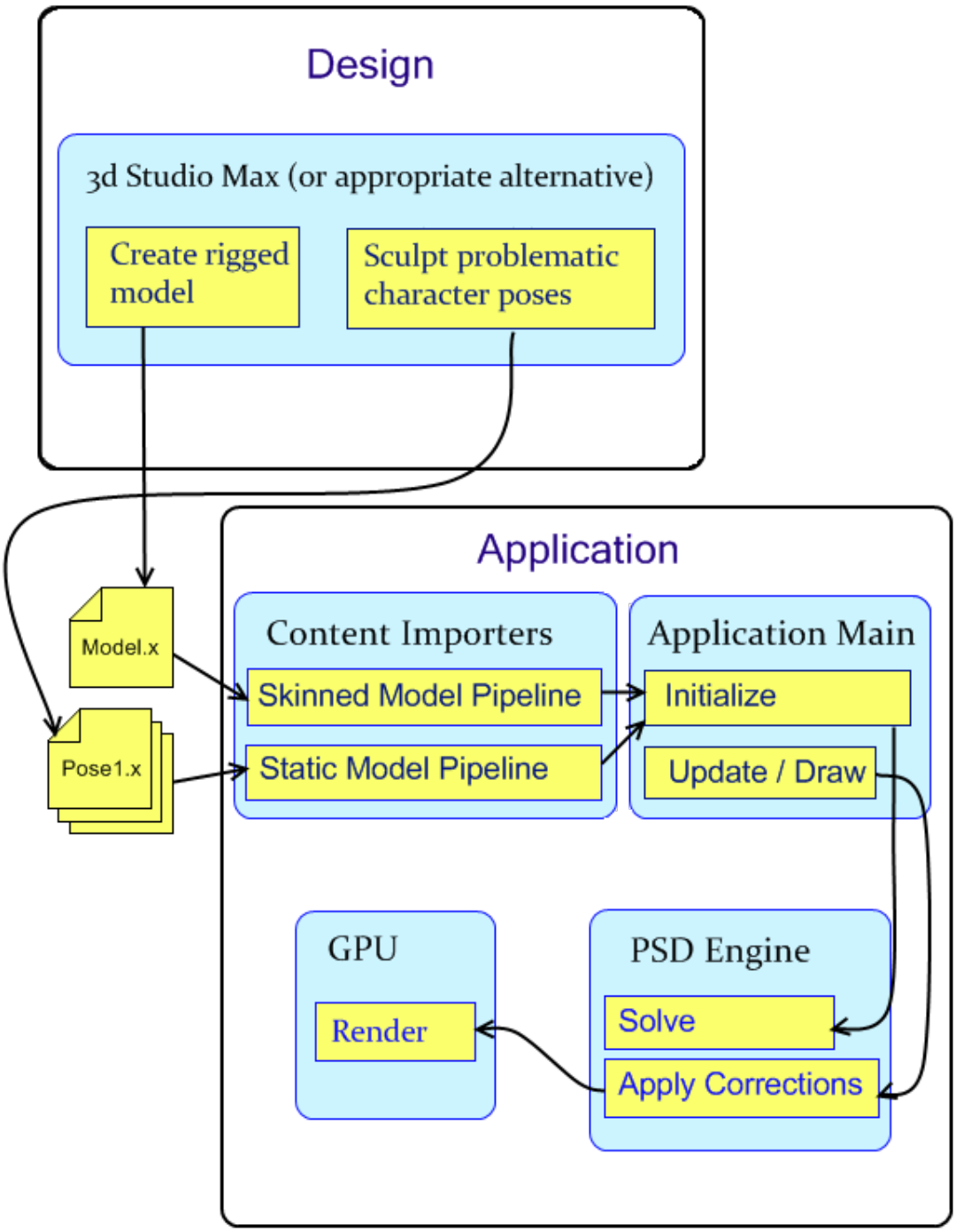

Figure 3.1: A high level overview of the system.

Chapter 4

Implementation

This chapter details the implementation which forms part of this thesis. The class level design details of the system is discussed in section 4.1. For a higher level architectural discussion see section 3.2. The details of the algorithm implementation are discussed in section 4.2. Variations on the algorithm, such as forward versus inverse *Pose Space Deformation*, are discussed in section 4.2.4 and the choices made in this implementation are justified. A discussion on the performance and memory considerations of the algorithm and the issue of choosing an appropriate value of σ are also discussed in section 4.3.

4.1 Class Level Design Details

Figure 4.1 shows a class diagram of the system. Only methods and attributes which are relevant to this system are shown, with boilerplate code such as graphics device and sprite batch variables being omitted. Attributes and methods which are used for debugging the system are also omitted. In figure 4.1 the ApplicationMain class contains, along with the standard `LoadContent()`, `Update()` and `Draw()` methods, the following important attributes:

- **animationContent**: A reference to the AnimationContent class which abstracts the details of loading individual animations, including which model and poses are to be loaded and their positions in pose space. The setup details for *pose space deformation* are also abstracted by the AnimationContent class.
- **camera**: The camera class calculates the relevant view and projection matrices and handles input from the keyboard or Xbox360 controller which is used to move the camera.
- **animationTime**: Stores the current position in time of the animation. This variable may be incremented or decremented using keyboard input.

The AnimationContent and AnimationPlayer class work together to co-ordinate the definition, loading and playback of animations. Animations are chosen by calling the relevant LoadAnimation method of AnimationContent. This loads the content relevant to that animation, initializes the

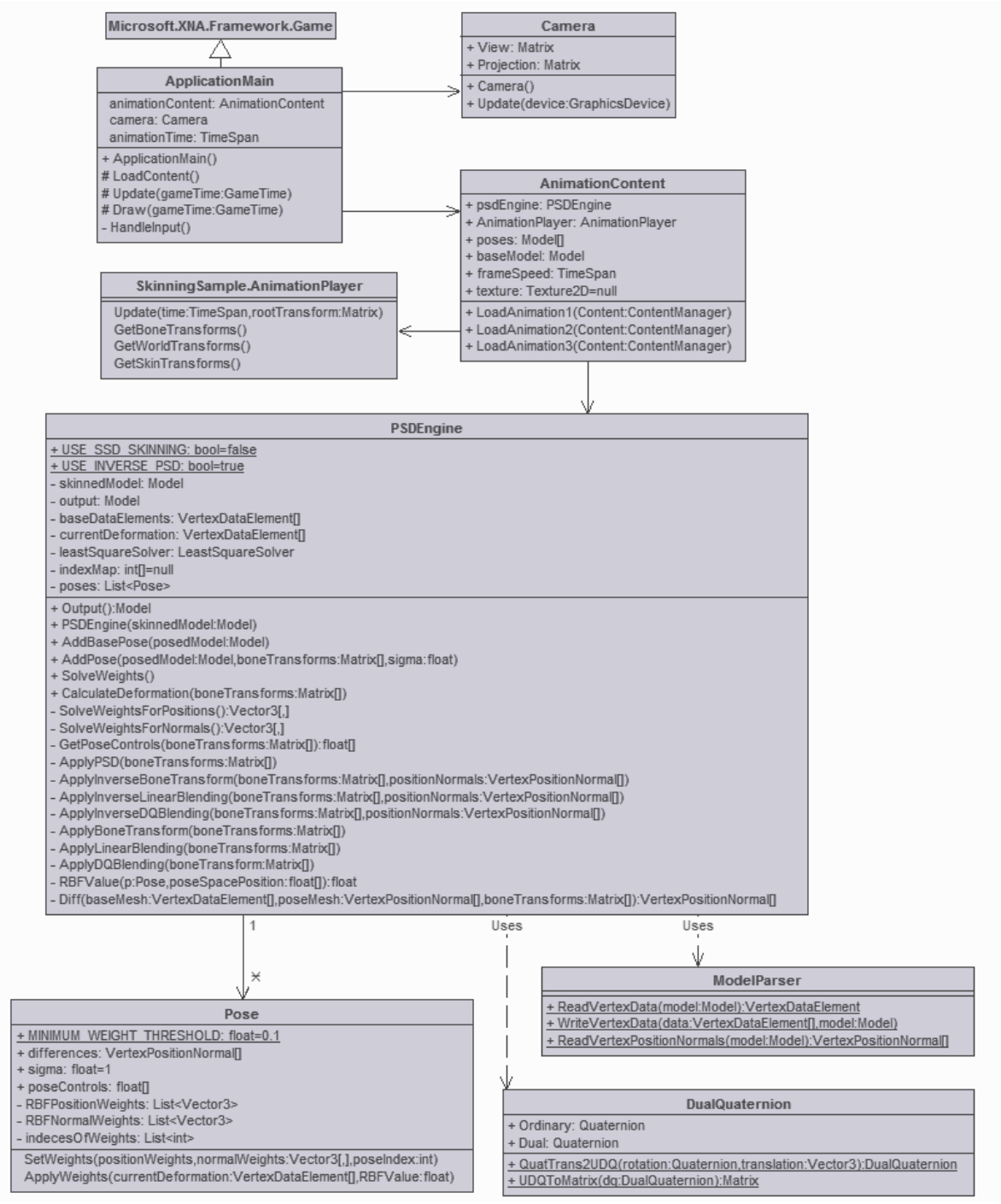

Figure 4.1: Class diagram of the system.

PSDEngine and handles the solving of the RBF weights through the PSDEngine class. The AnimationPlayer class is used to obtain the bone transforms for a particular time-step into the animation. The class was obtained from the skinning sample on the XNA creators club website[6] and contains only minor modifications. The bone transforms are available in the following formats:

- **Bone transforms**: The bone transform matrices, relative to their parent bone coordinate system. While this transform is not used directly in SSD it is necessary to compute the location in *pose space*.
- **World transform**: Returns the world transform of each bone. This consists of the concatenation of a particular bone, its parent bone, and its parent bone, all the way back to the root bone.
- **Skin transform**: This matrix specifies the transform with respect to the rest pose, and is applied to the rest pose mesh to perform skinning. The transform consists of the world transform concatenated with the inverse rest pose transform. See equation 2.1 in section 2.2.1 on SSD for more details.

The PSDEngine class contains most of the important functionality of this implementation. The important fields of the PSDEngine class are:

- Boolean switches to select between SSD and dual-quaternion skinning and to select between forward and inverse PSD.
- References to the base and output model. The base model is the skinned model to which PSD is applied. The output model is an XNA model type which has the model correction applied to it. It is then rendered by the `Draw()` method of the ApplicationMain class.
- The variables baseDataElements and currentDeformation. BaseDataElements contains the vertex position, normal and weight data which is extracted from the base model when the PSDEngine class is initialized. The currentDeformation stores updates to this data as SSD (or DQ) and PSD deformations are applied during each update step. The data in currentDeformation is eventually written to the output model so it may be displayed.
- The leastSquaresSolver is used to solve the linear least squares problem when calculating RBF weights.
- The indexMap field may be used to store a correspondence between the vertices in the base model and the pose models if their topologies differ.
- The list of poses stores the data associated with each individual pose.

The important methods of the PSDEngine are:

- The constructor and the `AddBasePose()` method which are used to add the base rigged model.

- `AddPose()` which is used to add a posed model, specify its bone transforms and specify a sigma falloff value for each pose.
- `SolveWeights()` which is used to solve the RBF weights after all the relevant poses have been added.
- `CalculateDeformation()` calculates the deformation given a list of bone transforms, performing the relevant SSD (or DQ) and PSD deformations as required.
- Further private access methods are provided to perform the following:
  - Calculate the RBF weights for both vertices and normals.
  - calculate the pose controls from a given set of bone transforms.
  - Apply forward and inverse transform blending using both SSD and dual-quaternions.
  - Calculate the RBF value given a particular pose and an arbitrary location in pose space.
  - Compute the difference offsets between two corresponding arrays of vertices.

A code example is provided in appendix A1 showing how to set up and use the PSDEngine class.

The Pose class provides a storage area for the data associated with each pose and provides methods to set and apply correction weights. The constituents of the pose space class are:

- MINIMUM_WEIGHT_THRESHOLD: Allows the setting of a threshold below which RBF weights are ignored.
- differences: An array to store the offsets between the pose mesh and the base mesh in a given pose.
- poseControls: The pose controls corresponding to a particular pose.
- RBFPositionWeights: A list of RBF position weight which are relevant because they are above the MINIMUM_WEIGHT_THRESHOLD.
- RBFNormalWeights: A list of RBF normal weight which are relevant because they are above the MINIMUM_WEIGHT_THRESHOLD.
- indecesOfWeights: A list of indeces mapping the RBF weights to their corresponding vertices. This list is needed because of the thresholding operation which means that not all vertices have corresponding weights.

The `ModelParser` class has the following methods:

- `ReadVertexData()`: Used to extract the vertex position, normal, bone and bone-weight data from an XNA Model object. Given a Model type this method returns an array containing the vertex data. This method should be called only on the fully rigged model, and not on the pose models which do not contain bone-weight data.

- `WriteVertexData()`: This method is used to write vertex data back to the Model type. It should be used after corrections have been applied so the corrected model may be rendered.
- `ReadVertexPositionNormals()`: Used to extract vertex position and normal data from a Model object. The data is returned as a `VertexPositionNormal` array. This method should be used on the pose models and not on the fully rigged model which contains bone-weight data.

4.2 Algorithm Implementation Details

4.2.1 Pose Space Definition

A number of formulations were considered for the definition of the pose space. The original PSD paper appears to use the euler angles of a joint rotation as its basis. The degree of freedom of the joint angle is set a-priori with, for example, the elbow joint having one degree of freedom so contributing one dimension to the pose space. The shoulder joint likewise contributes two values to the pose space. While this predefined assignment of joint DOFs appears to work well in the context of the original algorithm which was aimed at the motion picture industry, it does not appear to generalize well to real-time character animation, or characters which have more exotic skeleton types than the biped, where the DOF of the joints may not be immediately obvious.

For this reason we chose the axis-angle representation of the joint angle as the basis for the pose space, as is done in [25]. Using this formulation each joint contributes exactly 3 degrees of freedom to the 0pose space. We experimented with pose spaces made up of only the joints pertaining to a single arm, and pose spaces which contain all the joints for a given skeleton. Both appear to work well using the axis-angle representation. Since the rotation is available to the application in matrix format only the following conversions are needed:

$$
\begin{aligned}
\theta &= \arccos((m00 + m11 + m22 - 1)/2) \\
x &= (m21 - m12)\sqrt{(m21 - m12)^2 + (m02 - m20)^2 + (m10 - m01)^2} \\
y &= (m02 - m20)\sqrt{(m21 - m12)^2 + (m02 - m20)^2 + (m10 - m01)^2} \\
z &= (m10 - m01)\sqrt{(m21 - m12)^2 + (m02 - m20)^2 + (m10 - m01)^2}
\end{aligned}
$$

The final result is the normalized axis vector multiplied by the angle.

4.2.2 Solving the Radial Basis Function

At the solution stage the RBF weights must be calculated for each vertex. The weights consist of a 3 component vector and one weight is needed per pose, per vertex. For each vertex the weights are solved in the least squares sense as defined in equations 2.3.1. Since the matrix $\mathbf{\Phi}$ is the same for each vertex, the computation $\left(\mathbf{\Phi}^T\mathbf{\Phi}\right)^{-1}\mathbf{\Phi}^T$ may be pre-computed to save time by computing it for each vertex. Since the calculation is only performed once at startup time and does not affect the run-time

performance, the optimization is not too important. It would be more important in a system where real-time feedback is needed when authoring the animation though.

4.2.3 Application of Corrections

The algorithm for the application of PSD corrections is as follows:

- Calculate the current position $\mathbf{x}$ in *pose space*.
- Calculate the RBF values $\phi(|\mathbf{x} - \mathbf{x_j}|)$ for each pose j, using the distance between the current *pose space* location $\mathbf{x}$ and the *pose space* location of each pose $\mathbf{x_j}$. Note that the square root operation does not need to be performed when calculating the distance because the Gaussian function uses the distance squared value anyway.
- For each pose j:
  - If $\phi(|\mathbf{x}-\mathbf{x_j}|)$ for that pose is greater than ϵ (where ϵ is defined as some very small constant), Apply the pose corrections for each vertex i associated with the pose:
  - $\mathbf{v_i} = \mathbf{v_i} + \mathbf{w_{i,j}}\phi(|\mathbf{x} - \mathbf{x_j}|)$ where $\mathbf{v_i}$ is the position of vertex i and $\mathbf{w_{i,j}}$ is the RBF weight corresponding to vertex i and pose j.
  - The same process is also applied to the vertex normals using the RBF normal weights.

4.2.4 Inverse PSD

Our implementation uses the inverse PSD variation which is described in section 2.3.3 and has its advantages discussed in [28]. We found that the use of forward PSD was not suitable for real-time application by considering the following example. If pose corrections are sculpted for some wrist vertices and then the shoulder joint is rotated, then in the case of forward PSD the displacement corrections will be oriented in the wrong direction. Inverse PSD would not have this problem because the corrected pose is first transformed the via inverse SSD (or dual-quaternion) transform to its rest pose equivalent. This means that the interpolation is essentially performed in the local joint coordinate frame rather than in world space, as would be the case if forward PSD were used. Using inverse PSD the solution stage has the following order:

- Apply the inverse SSD (or dual-quaternion) transform to the pose data.
- Calculate the difference between the rest pose vertices and the transformed pose data.
- Solve for the RBF weights.

And the synthesis stage has the following order:

- Apply PSD corrections to the rest pose vertices.
- Apply the SSD or dual-quaternion transform.

4.3 Miscellaneous Issues

As mentioned in section 4.1 the pose class stores a list of position and vertex weights associated with that pose, as well as a list of indices to indicate which vertices the corrections apply to. The weights are thresholded to eliminate weights whose contribution is negligible. This has the effect of reducing the amount of run time processing needed to apply the corrections and to reduce the amount of memory needed to store the weights. The threshold is set at 0.1 which has been empirically verified to adequately reduce the number of corrections without compromising the quality of the animation.

Some of the meshes to which we applied our method had a difference in the mesh topography between the underlying rigged model and the pose model, i.e. the vertices were listed in different orders when imported into our system. This problem is mentioned but not addressed in [23]. The solution we found was to export a static pose and its rigged equivalent and compare the two. By comparing the vertices in the rigged model with the vertices in the static pose to find its closest matching vertex, we were able to keep a list of corresponding indices and create a calibration step which overcomes the correspondence problem.

For the generation of a number of character poses the 3d studio max physique modifier was used. The physique modifier uses a bone weight rigging which is similar to SSD but also allows sub-structural effects such as bulges and tendons. By using a 3d model which is well rigged using the physique modifier we cut down on the need to manually sculpt example poses while retaining a rigging structure which is suitable for SSD and dual-quaternion skinning.

The choice of a suitable value for σ was made partly through a process of trial and error and partly through intuition as to what might constitute a good value. For animations such as the bar and arm examples from section 5 (figures 5.1 and 5.2), where the poses are arbitrarily spaced and where no tight cluster of poses exists, a uniform choice of σ seems the obvious choice. Since the *pose space* is defined in terms of the angle of rotation of the joints, which has upper and lower limits at π and $-\pi$ respectively for any natural skeleton movements, a value of σ which is of this order of magnitude seemed a reasonable choice. A value of 1.0 was used and appears to provide a good interpolation in both the bar and arm animation examples. The lower σ value of 0.15 were also tested for the arm example but they did not provide adequate interpolation between the two most distant poses, with the influence of each poses only becoming evident as the animation moved into close proximity of that pose. Further experiments with non-uniform values of σ showed instability in the interpolation indicating that the use of a varying value requires a considerable amount of tweaking. It was also noted that, when using a number of poses which are in close proximity to each other, a smaller value of σ is necessary to avoid an undesired accumulation of pose corrections from a number of surrounding poses at locations close to the cluster of poses.

Chapter 5

Results

This section outlines our results obtained by observing our method running a number of different animations and by benchmarking it against the available alternatives. Sections 5.1 provides some verifications of the improvement seen by using our method in image format. The videos associated with these tests may be found at the authors research website[2]. Section 5.2 presents some performance and memory metrics derived from our implementation and discusses their implications. Section 5.3 provides further analysis as to why our method is better at handling rotational joint blending than PSD on top of SSD.

5.1 Correctness and Simplicity of our Method

It is well known that *pose space deformation* does not adequately correct the rotational errors produced by SSD when the corrections are applied in the rest pose, i.e. *inverse PSD* is used. This has been noted in the work of Robert Wang[25] which describes a sophisticated solution to the problem. Figure 5.1 shows a simple example where a bar is deformed using two bones. The upper bone is rotated through 180 degrees while the lower bone is not moved. Applying corrections to the SSD deformed model results in artifacts when interpolating between the 3 given poses. When corrections are applied on top of the equivalent dual-quaternion deformation no artifacts are visible.

Figure 5.2 shows a real world example of the rotational error, where 4 training poses have been used in an arm animation. Here the SSD version results in artifacts throughout the animation, and only deforms correctly at exactly the location of the example poses. The PSD on top of Dual-quaternions version, however, interpolates correctly all of the way through the animation and displays no rotational artifacts.

5.2 Performance and Memory Requirements of our Method

Table 5.1 shows the performance data in frames per second for a number of test animations. Interestingly the dual-quaternions version outperforms the SSD version by a small margin in each case. This

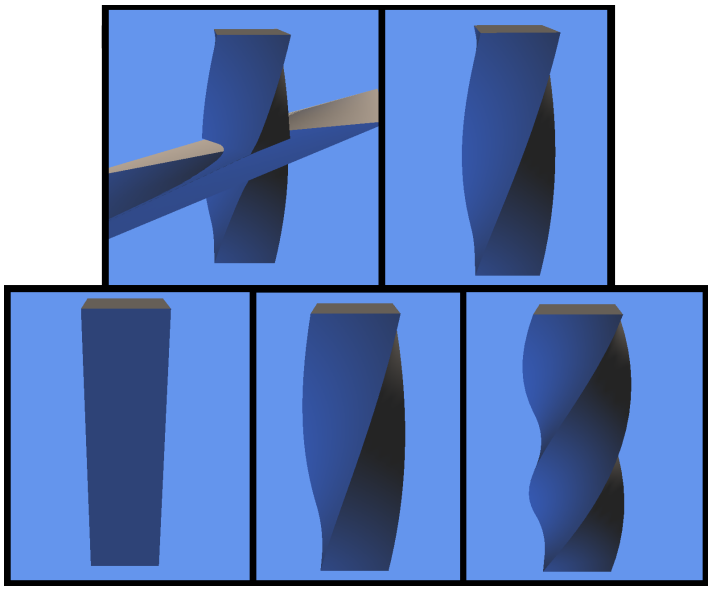

Figure 5.1: Evaluation of the animation of a bar in an unseen test pose. The animation was trained using 3 poses. Top left shows the result in the unseen pose using PSD on top of SSD skinning, and even though the unseen pose is very close to the second training pose large errors are still visible. Top right shows the same pose using PSD on top of dual-quaternion skinning. The bottom row shows the 3 training poses.

appears to be a side effect of performing the skinning computation on the CPU, because when the same animations are run with GPU skinning (with no pose corrections) the SSD version outperforms the dual-quaternion version.

If the slowdown due to adding pose corrections is viewed as a percentage of the original execution time the two techniques show broadly similar performance, with the box animation showing a 22% slowdown for PSD/SSD corrections and a 26% slowdown for PSD/DQ corrections. Similarly the candy wrapper animation shows an 11% and 14% slowdown for PSD/SSD and PSD/DQ respectively. The figure for the wave animation is 11% versus 9.5% and for the rotate wrist animation with 3 pose corrections is 9% versus 13%. From this we can conclude that the cost of applying pose corrections is broadly similar in the case of PSD on top of SSD and PSD on top of dual-quaternions.

In terms of correctness of the interpolation, the rotate wrist animation with 3 pose corrections provides a smooth, artifact free animation of the wrist rotating through 180 degrees, when computed using PSD on top of dual-quaternion skinning. The PSD on top of SSD version of the same animation displays interpolation artifacts between the 3 pose locations. The 5 pose version of the same animation also displays artifacts when skinned using PSD/SSD, and only when 9 equally spaced poses are added for the rotation through 180 degrees does the PSD/SSD animation match the quality of the PSD/DQ version with 3 poses. So it seems a comparison of the 3 pose version using PSD on top of dual-quaternion skinning and the 9 pose version using PSD on top of SSD skinning would be an appropriate one. The 3 pose DQ version runs at 125 frames per second while the 9 pose SSD version runs at 72

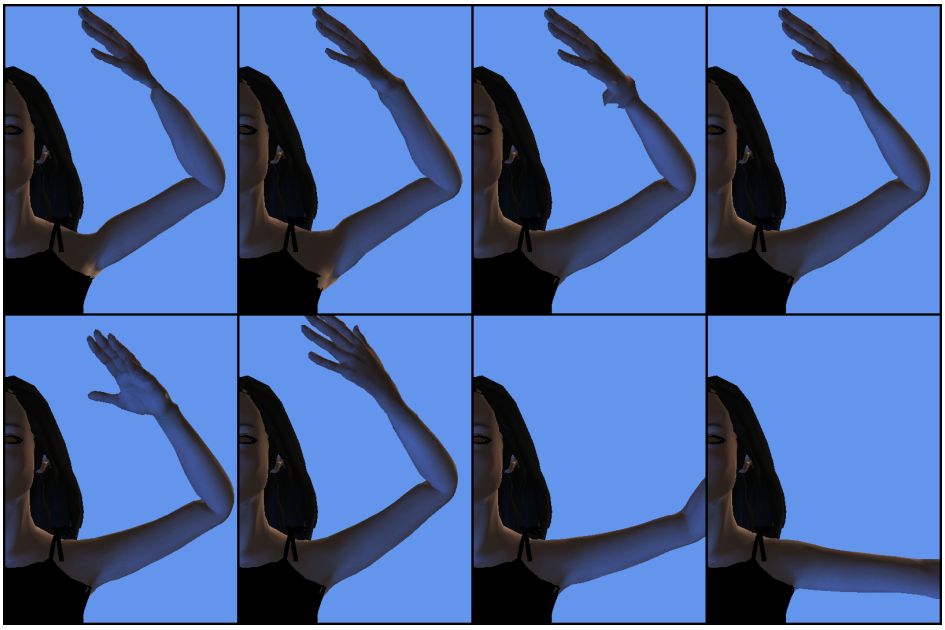

Figure 5.2: Top From Left: SSD deformed model in which joint collapse is visible. Dual-quaternion deformed model which corrects rotational errors but does not allow fine grained re-sculpting of the model. PSD applied on top of SSD, where the rotational error results in an incorrect interpolation. PSD applied on top of dual-quaternion skinning, which displays none of the rotational errors. Bottom Row shows the 4 training poses.

Animation	Vertices	Fixed	Poses	σ	PSD/SSD	SSD	PSD/DQ	DQ
Box Animation	2646	2641	3	1	235	287	267	337
CandyWrapper	7992	2680	4	1	103	114	121	138
Wave	7992	7367	1	1	107	119	126	138
WristRotate	7992	2985	3	1	106	116	125	141
WristRotate	7992	7992	5	4	81	116	91	116
WristRotate	7992	6339	9	1	72	116	78	116

Table 5.1: Performance Data : The performance data in frames per second shown for a number of animations. Also shown are the total number of vertices for the model used in each animation, the average number of vertices fixed (number of vertices fixed did not vary significantly between poses, nor did it vary significantly depending on whether the SSD or DQ method was used). The number of poses used and the value of sigma used is also shown. Frame rate data is given for PSD on top of SSD, SSD with no pose corrections, PSD on top of dual-quaternions and dual-quaternion skinning with no pose corrections.

frames per second, representing a 74% increase in performance. In terms of memory requirements, 28 bytes are needed to store the pose corrections for a vertex, which consists of 2 Vector3 values, one for the position correction and one for the normal correction, and a 4 byte index to denote which vertex the correction belongs to. In order to store the 2985 pose corrections needed per pose, the 3 pose DQ version needs 245Kb ($2985 \times 3 \times 28$ bytes). The 9 pose SSD version, which is found to be visually equivalent needs 1600Kb to store the pose corrections ($6339 \times 9 \times 28$ bytes). Therefore the memory saved by our method in this case is 85%.

5.3 Why PSD on top of SSD does not Correct Rotational Errors

The inverse transform of example poses to their rest pose equivalent is dependent on the nature of the blended matrices, which must be inverted then applied to the vertices of the pose. In the case where SSD blending is used between two joints, as the angle between the two joints approaches 180 degrees the blended matrix approaches singularity, and the inverse of the matrix approaches infinity. This fact is responsible for the PSD on top of SSD rotation errors in the bar rotation example in figure 5.1, and in the arm example in figure 5.2. It can be easily visualized in figure 5.3 which shows the inverse SSD transformed poses from the bar example. Our method using dual-quaternion skinning does not suffer the same problem because the DQ blending always results in a rigid-body transformation which is easily invertible.

To help quantify the error a simple animation was created which rotates the wrist through 180-degrees. It was found that using our method with 3 example poses is visually equivalent to using the SSD on top of PSD method with 9 example poses. When used with 3 example poses the SSD based method displayed artifacts due to the extreme nature of the inverse SSD transformed poses as the angle of rotation approaches 180 degrees.

To further analyze why a correct interpolation between the poses was not possible using the SSD

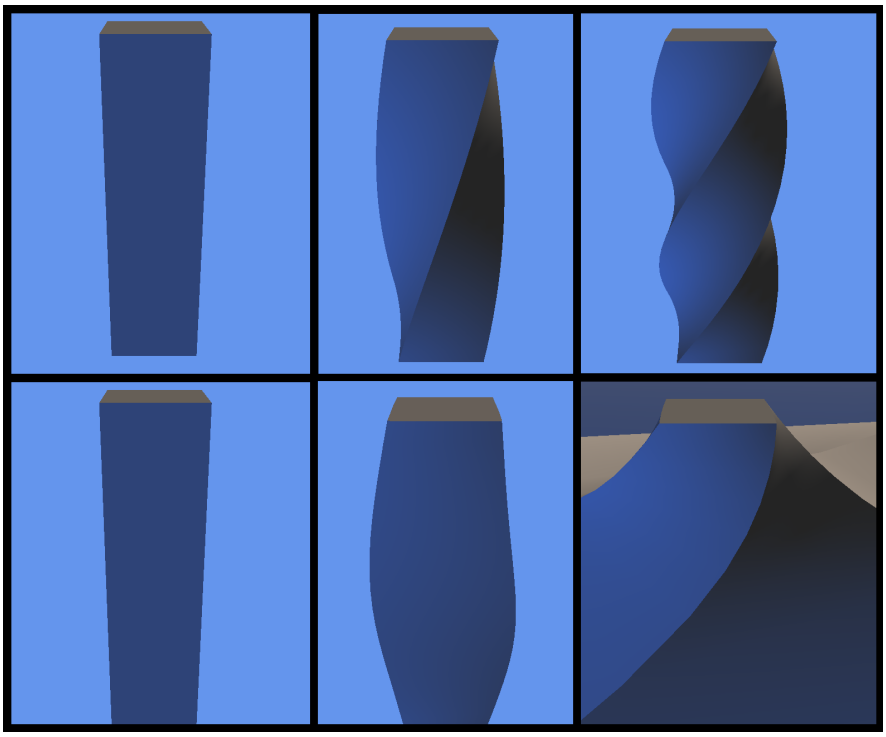

Figure 5.3: Inverse SSD transform of three poses. The top row shows the example poses used to train the bar rotation animation and the bottom row shows their equivalent deformations when the inverse SSD transformation is applied.

method with 3 examples, we tracked the position of the vertex which showed the largest displacement from the underlying mesh and plotted its displacement correction values against the interpolated displacement correction. The interpolation was recreated using the RBF weight values from our application. This is shown for 9 poses using the SSD method in figure 5.4, for 3 poses using the SSD method in figure 5.5 and for 3 poses using the dual-quaternion method in figure 5.6.

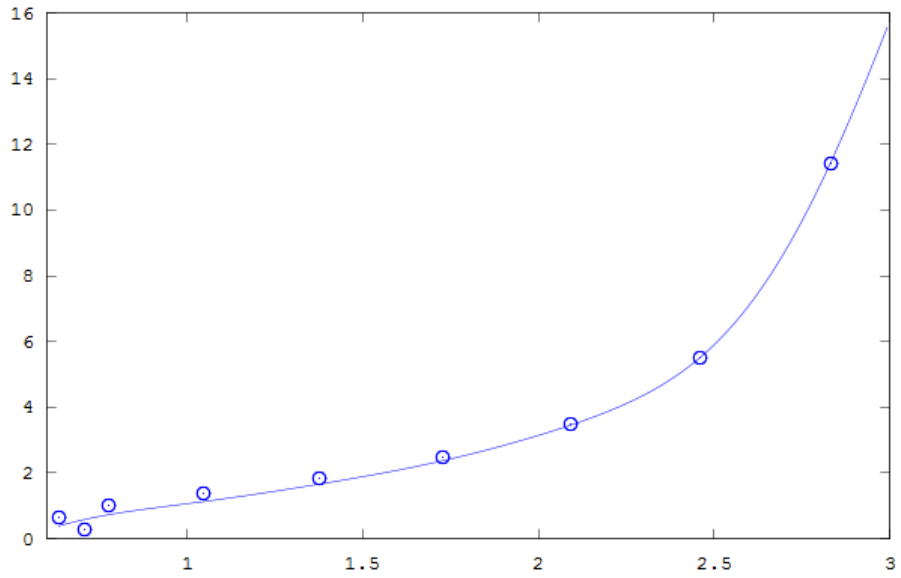

Figure 5.4: Graph of 9 example poses using SSD. The vertical axis represents the y-value of the vertex displacement correction. The horizontal axis represents the angle of rotation in radians. The circular points are the data points corresponding to the 9 poses and the function graphed is the interpolation of the displacements. Notice that as the angle approaches π (180 degrees) the function tends to infinity, yet the use of 9 data points provides a reasonable approximation of this function.

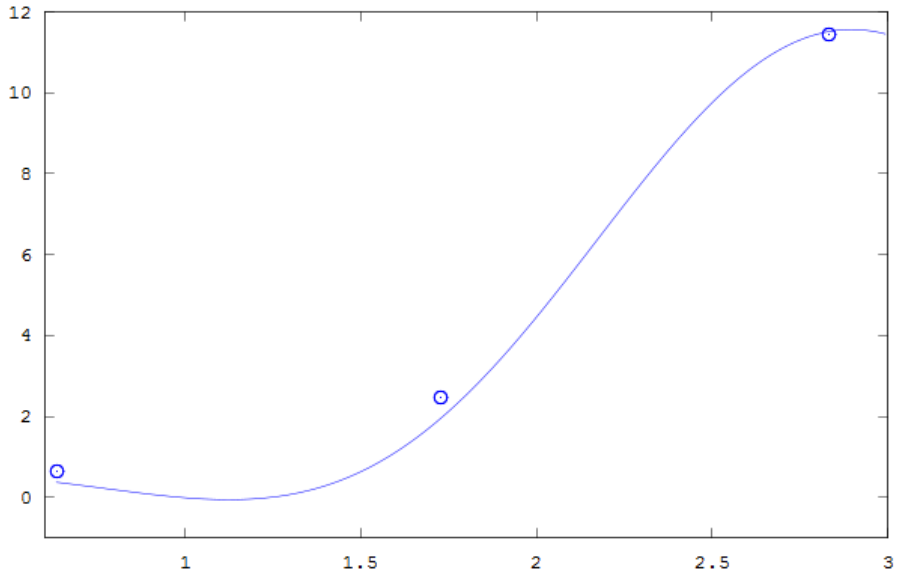

Figure 5.5: Graph of 3 example poses using SSD. Due to under-sampling the interpolated function does not provide a reasonable approximation when compared to figure 5.4.

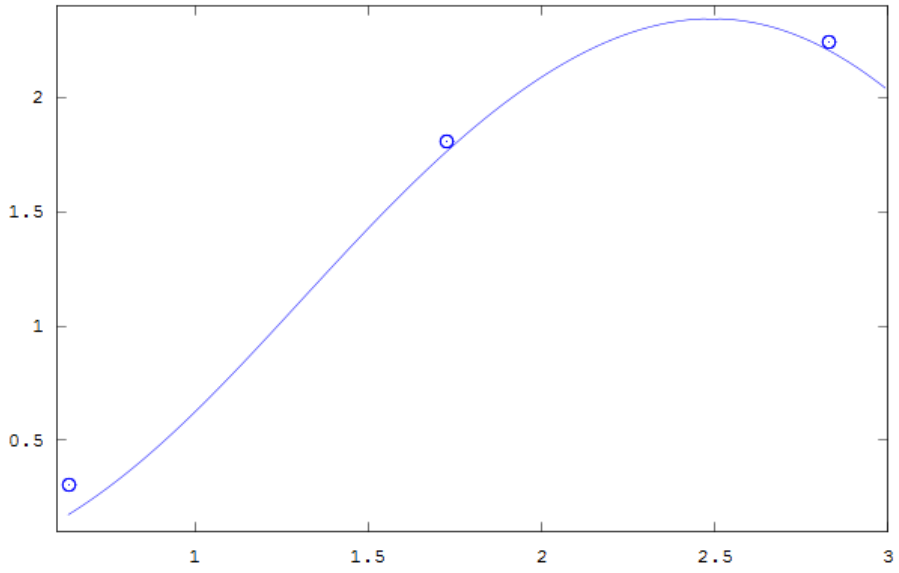

Figure 5.6: Graph of 3 example poses using dual-quaternions provides a reasonable approximation of the displacements unlike the SSD equivalent because the function does not approach infinity at any stage. Also the displacements are of a smaller magnitude than the SSD equivalent.

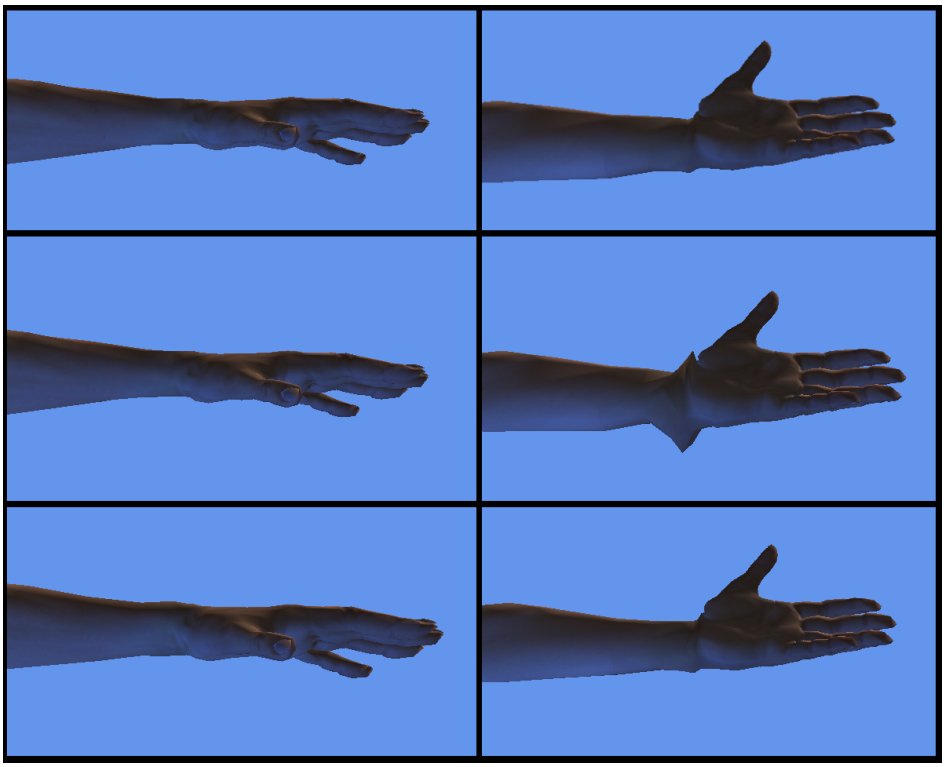

Figure 5.7: Results of the wrist rotation animation in various configurations. Top row: PSD on top of dual-quaternions with 3 example poses. Middle row: PSD on top of SSD using 3 example poses. Bottom row: PSD on top of SSD using 9 example poses.

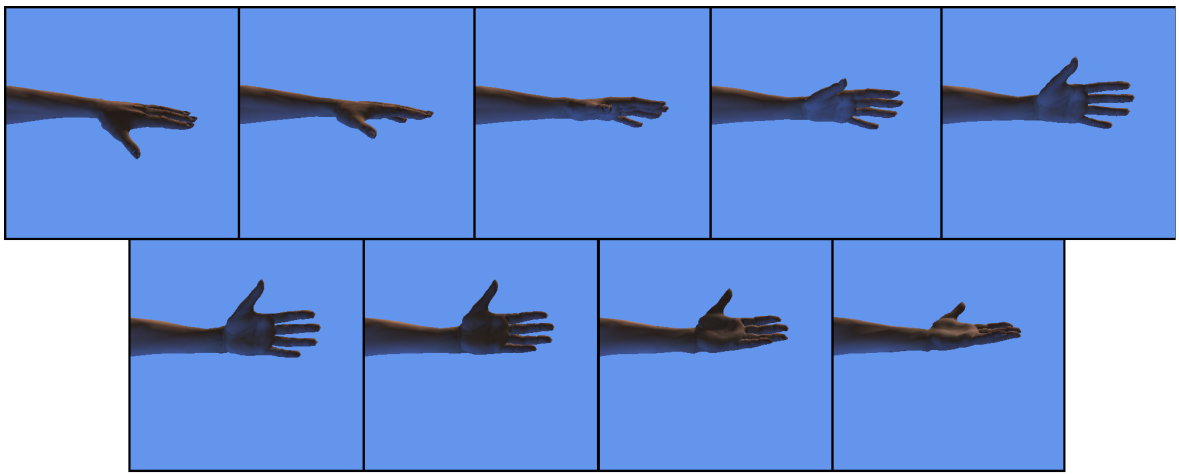

Figure 5.8: The 9 poses used to train the animation.

Chapter 6

Conclusion and Further Work

6.1 Conclusion

Our method shows an increase in correctness over the current PSD on top of SSD example based methods such as 'Shape by Example'[23] and 'Eigenskin'[14], which means that fewer poses need to be sculpted to achieve the same level of realism. We believe that this represents a step forward in terms of research into example base skinning. Compared to the rotational regression model[25] our method is shown to solve the same problem in a manner which is easier to understand and implement.

6.2 Further Work

While we have shown the benefits of our method by implementing it on the CPU, for it to be useful in a real world application a GPU implementation would probably be necessary. Since dual-quaternion skinning is GPU friendly, porting this part of the system to the GPU would not be difficult. Porting the PSD component over is a little trickier since an RBF weight for each pose would need to be passed to the vertex shader as per-vertex data. There is a fixed amount per-vertex data which can be passed to the GPU, but since our method reduces the number of poses needed this may now be a viable option. Another method used in [24] is to pack the RBF weight data into a texture and apply it in the fragment shader, storing the result to a texture which is then used in a subsequent rendering pass. In this manner PSD is applied through multiple rendering passes. This method could be used to implement our algorithm on the GPU.

A problem not addressed in this research is how to handle the case of flipping due to quaternion antipody. That is when a rotation reaches 180 degrees a flipping artifact occurs because the use of dual-quaternion skinning always chooses the shortest path of blending. Here the PSD corrections become incorrect because the underlying deformation is not continuous across the flip line, while the interpolation function is. One possible solution might be to make a special case of the comparison of pose locations which lie on either side of the flip line and handle the problem when distances in the

pose space are being calculated. Using the dual-quaternion representation to formulate pose space distances might be of some help here.

Weighted PSD provides an improvement to PSD by localizing the influence of individual pose controls on the resulting corrections, but at a cost in terms of execution time. As a result fewer poses are necessary when using weighted PSD, as is also the case with our method. It would be interesting to investigate if the two methods are complimentary to one another.

Dimensionality reduction of the weight data provides improved efficiency and reduced memory consumption in 'Eigenskin'[14]. It would be worth investigating dimensionality reduction with our method to see if the same benefits are realized.

A. Appendix

A1. Usage of the PSDEngine Class

The following code example illustrates how the PSDEngine class is initialized. It also illustrates how the AnimationPlayer class is used.

```
TimeSpan frameSpeed = new TimeSpan(0, 0, 0, 0, 30);
Model baseMesh = Content.Load<Model>("Wave\\masha_output");
Model [] poses = new Model[4];
poses[0] = Content.Load<Model>("Wave\\masha_pose130");
poses[1] = Content.Load<Model>("Wave\\masha_pose200");
poses[2] = Content.Load<Model>("Wave\\masha_pose230");
poses[3] = Content.Load<Model>("Wave\\masha_pose260");

// Look up our custom skinning information.
SkinningData skinningData = baseMesh.Tag as SkinningData;

if (skinningData == null)
    throw new InvalidOperationException
        ("This model does not contain a SkinningData tag.");

// Create an animation player, and start decoding an animation clip.
AnimationPlayer animationPlayer = new AnimationPlayer(skinningData);
AnimationClip clip = skinningData.AnimationClips["Anim-1"];
animationPlayer.StartClip(clip);

// Create the PSD Engine
PSDEngine psdEngine = new PSDEngine(baseMesh);

animationPlayer.Update(new TimeSpan(0, 0, 0, 5, 200), false, Matrix.Identity);
psdEngine.AddPose(poses[0], animationPlayer.GetSkinTransforms(), 0.9f);
```

```
animationPlayer.Update(new TimeSpan(0, 0, 0, 8, 0), false, Matrix.Identity);
psdEngine.AddPose(poses[1], animationPlayer.GetSkinTransforms(), 0.9f);

animationPlayer.Update(new TimeSpan(0, 0, 0, 9, 200), false, Matrix.Identity);
psdEngine.AddPose(poses[2], animationPlayer.GetSkinTransforms(), 0.9f);

animationPlayer.Update(new TimeSpan(0,0,0,10,400), false, Matrix.Identity);
psdEngine.AddPose(poses[3], animationPlayer.GetSkinTransforms(), 0.9f);

psdEngine.CPUSkeleton = true;
psdEngine.SolveWeights();
```

Bibliography

[1] Codeplex: Open source community, August 2008. http://www.codeplex.com/.

[2] Damien murtagh, research page, September 2008. https://www.cs.tcd.ie/~murtagda/research.

[3] Panda exporter, August 2008. http://www.andytather.co.uk/Panda/directxmax.aspx.

[4] Soiciety games, August 2008. http://www.societygames.com/dualquaternion/DualQuaternion.html.

[5] Turbo squid, 3d model resources, August 2008. http://www.turbosquid.com/.

[6] Xna creators club, August 2008. http://creators.xna.com/.

[7] Alexa, M. Linear combination of transformations. *ACM Trans. Graph. 21*, 3 (2002), 380–387.

[8] Bishop, C. M. *Neural Networks for Pattern Recognition*. Oxford University Press, November 1995.

[9] Blythe, D. The direct3d 10 system. *ACM Trans. Graph. 25*, 3 (2006), 724–734.

[10] Hejl, J. Hardware skinning with quaternions. 487–495.

[11] Kavan, L., Collins, S., Žára, J., and O'Sullivan, C. Skinning with dual quaternions. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 39–46.

[12] Kavan, L., Collins, S., Zara, J., and O'Sullivan, C. Geometric skinning with approximate dual quaternion blending. vol. 27, ACM Press.

[13] Kavan, L., and Žára, J. Spherical blend skinning: a real-time deformation of articulated models. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), ACM, pp. 9–16.

[14] Kry, P., James, D., and Pai, D. Eigenskin: Real time large deformation character skinning in hardware, 2002.

[15] Kurihara, T., and Miyata, N. Modeling deformable human hands from medical images. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2004), Eurographics Association, pp. 355–363.

[16] Lewis, J. P., Cordner, M., and Fong, N. Pose space deformations: A unified approach to shape interpolation and skeleton-driven deformation. In *Siggraph 2000, Computer Graphics Proceedings* (2000), K. Akeley, Ed., ACM Press / ACM SIGGRAPH / Addison Wesley Longman.

[17] Magnenat-Thalmann, N., Laperrière, R., and Thalmann, D. Joint-dependent local deformations for hand animation and object grasping. In *Proceedings on Graphics interface '88* (Toronto, Ont., Canada, Canada, 1988), Canadian Information Processing Society, pp. 26–33.

[18] Magnenat-Thalmann, N., Laperrière, R., and Thalmann, D. Joint-dependent local deformations for hand animation and object grasping. In *Proceedings on Graphics interface '88* (Toronto, Ont., Canada, Canada, 1988), Canadian Information Processing Society, pp. 26–33.

[19] Mohr, A., and Gleicher, M. Building efficient, accurate character skins from examples. *ACM Trans. Graph. 22*, 3 (2003), 562–568.

[20] Powell, M. J. D. Radial basis functions for multivariable interpolation: a review. 143–167.

[21] Scheepers, F., Parent, R. E., Carlson, W. E., and May, S. F. Anatomy-based modeling of the human musculature. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 163–172.

[22] Shoemake, K. Animating rotation with quaternion curves. *SIGGRAPH Comput. Graph. 19*, 3 (1985), 245–254.

[23] Sloan, P.-P. J., Charles F. Rose, I., and Cohen, M. F. Shape by example. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (New York, NY, USA, 2001), ACM, pp. 135–143.

[24] Taehyun Rhee, J.P. Lewis, U. N. Real-time weighted pose-space deformation on the gpu. *Computer Graphics Forum 25*, 3 (2006), 439–448.

[25] Wang, R. Y., Pulli, K., and Popović, J. Real-time enveloping with rotational regression. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers* (New York, NY, USA, 2007), ACM, p. 73.

[26] Wang, X. C., and Phillips, C. Multi-weight enveloping: least-squares approximation techniques for skin animation. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2002), ACM, pp. 129–138.

[27] Wilhelms, J., and Gelder, A. V. Anatomically based modeling. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 173–180.

[28] Xian, X., Lewis, J. P., Soon, S. H., Fong, N., and Feng, T. A powell optimization approach for example-based skinning in a production animation environment. In *CASA* (2006).