A LINEAR FRAMEWORK FOR CHARACTER SKINNING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
OF THE UNIVERSITY OF CAPE TOWN
IN FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Bruce Merry
November 2007

Supervised by
Patrick Marais and James Gain

Abstract

Character animation is the process of modelling and rendering a mobile character in a virtual world. It has numerous applications both off-line, such as virtual actors in films, and real-time, such as in games and other virtual environments. There are a number of algorithms for determining the appearance of an animated character, with different trade-offs between quality, ease of control, and computational cost. We introduce a new method, *animation space*, which provides a good balance between the ease-of-use of very simple schemes and the quality of more complex schemes, together with excellent performance. It can also be integrated into a range of existing computer graphics algorithms.

Animation space is described by a simple and elegant linear equation. Apart from making it fast and easy to implement, linearity facilitates mathematical analysis. We derive two metrics on the space of vertices (the "animation space"), which indicate the mean and maximum distances between two points on an animated character. We demonstrate the value of these metrics by applying them to the problems of parametrisation, level-of-detail (LOD) and frustum culling. These metrics provide information about the entire range of poses of an animated character, so they are able to produce better results than considering only a single pose of the character, as is commonly done.

In order to compute parametrisations, it is necessary to segment the mesh into charts. We apply an existing algorithm based on greedy merging, but use a metric better suited to the problem than the one suggested by the original authors. To combine the parametrisations with level-of-detail, we require the charts to have straight edges. We explored a heuristic approach to straightening the edges produced by the automatic algorithm, but found that manual segmentation produced better results. Animation space is nevertheless beneficial in flattening the segmented charts; we use least squares conformal maps (LSCM), with the Euclidean distance metric replaced by one of our animation-space metrics. The resulting parametrisations have significantly less overall stretch than those computed based on a single pose.

Similarly, we adapt appearance preserving simplification (APS), a progressive mesh-based LOD algorithm, to apply to animated characters by replacing the Euclidean metric with an animation-space metric. When using the memoryless form of APS (in which local rather than global error is considered), the use of animation space for computations reduces the geometric errors introduced by LOD decomposition, compared to simplification based on a single pose. User tests, in which users compared video clips of the two, demonstrated a statistically significant preference for the animation-space simplifications, indicating that the visual quality is better as well. While other methods exist to take multiple poses into account, they are based on a sampling of the pose space, and the computational cost scales with the number of samples used. In contrast, our method is analytic and uses samples only to gather statistics.

The quality of LOD approximations by improved further by introducing a novel approach to LOD, *influence simplification*, in which we remove the influences of bones on vertices, and adjust the remaining influences to approximate the original vertex as closely as possible. Once again, we

use an animation-space metric to determine the approximation error. By combining influence simplification with the progressive mesh structure, we can obtain further improvements in quality: for some models and at some detail levels, the error is reduced by an order of magnitude relative to a pure progressive mesh. User tests showed that for some models this significantly improves quality, while for others it makes no significant difference.

Animation space is a generalisation of skeletal subspace deformation (SSD), a popular method for real-time character animation. This means that there is a large existing base of models that can immediately benefit from the modified algorithms mentioned above. Furthermore, animation space almost entirely eliminates the well-known shortcomings of SSD (the so-called "candy-wrapper" and "collapsing elbow" effects). We show that given a set of sample poses, we can fit an animation-space model to these poses by solving a linear least-squares problem.

Finally, we demonstrate that animation space is suitable for real-time rendering, by implementing it, along with level-of-detail rendering, on a PC with a commodity video card. We show that although the extra degrees of freedom make the straightforward approach infeasible for complex models, it is still possible to obtain high performance; in fact, animation space requires fewer basic operations to transform a vertex position than SSD. We also consider two methods of lighting LOD-simplified models using the original normals: tangent-space normal maps, an existing method that is fast to render but does not capture dynamic structures such as wrinkles; and tangent maps, a novel approach that encodes animation-space tangent vectors into textures, and which captures dynamic structures. We compare the methods both for performance and quality, and find that tangent-space normal maps are at least an order of magnitude faster, while user tests failed to show any perceived difference in quality between them.

Acknowledgements

My supervisors, Patrick Marais and James Gain, were invaluable in offering direction and critical comment in my thesis. Many times they returned drafts covered in comments saying "you need to expand this," and usually they were right. Dave Jacka and Ashley Reid took on a final-year project based on my research, and their skills made it a success that strengthened my results. The Geometry Interest Group, particularly Carl Hultquist and Chris de Kadt, also provided valuable insights.

Finding usable data proved to be a problem, and I am indebted to those who made theirs available. Robert Sumner posted the horse and cat models on his web page, and J. P. Lewis did likewise with the arm model. The Blender Foundation included "mancandy" in the regression test suite for Blender 2.43, and Id Software kindly gave permission for me to use the models from their bestselling game Doom III for research purposes.

I would also like to thank all the people in the lab who for the last year have insisted on only playing games that I did not like, meaning that I could get some work done and finish my thesis on time.

The National Research Foundation provided funding for my research, as did the UCT Postgraduate Funding Office through the KW Johnston, Myer Levinson, Nelli Brown Spilhaus and Harry Crossley scholarships.

And finally, my thanks go to my family and friends who have been there for me over the last four years.

Notation

Except where otherwise noted, we will use the following notation:

- scalars will be written in lowercase, normal font e.g., x;
- vectors will be written in lowercase bold e.g., $\mathbf{x}$;
- matrices will be written in uppercase, normal font e.g., M;
- standard sets will be written like this: $\mathbb{R}, \mathbb{Z}$;
- sets that we define will be written like this: $\mathcal{A}, \mathcal{P}$; and
- the space of $m \times n$ matrices over the field $\mathcal{S}$ will be written as $\mathcal{S}^{m \times n}$, e.g., $\mathbb{R}^{m \times n}$.

Vector and matrix notation

Throughout the thesis, we will work in a homogeneous space. We will also be doing some complex manipulations of 4-vectors and 4×4 transformation matrices in this space. In this section we will define a few of pieces of notation to assist with this, and also list some basic identities to aid in later proofs.

At times it will be useful to extract just the weight (4th) coordinate from a homogeneous vector, which will be denoted $\underline{\mathbf{v}}$. The part of the vector excluding the weight is denoted $\overline{\mathbf{v}}$.

Let M be a 4×4 matrix. We will deal only with affine transformations, that is, matrices whose last row is $\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$. We define the *linear* part of M to be the top-left 3×3 submatrix, denoted $\overline{M}$. The *translation* part of M is the vector consisting of the first three elements of the last column, denoted $\underline{M}$.

The following identities are used frequently, and may easily be verified by expansion.

1. The four operators defined here are all linear operators.
2. Applying a matrix M to a vector $\mathbf{s}$ applies the linear part to $\overline{\mathbf{s}}$, then translates by $\underline{\mathbf{s}}$ times

the translation part. The weight of $\mathbf{s}$ is not affected:

$$\overline{M\mathbf{s}} = \overline{M}\overline{\mathbf{s}} + \underline{M}\ \underline{\mathbf{s}} \tag{1}$$

$$\underline{M\mathbf{s}} = \underline{\mathbf{s}}. \tag{2}$$

3. The "linear selector" operator distributes over multiplication:

$$\overline{AB} = \overline{A}\ \overline{B}, \tag{3}$$

4. Translations are transformed by matrices multiplied on the left:

$$\underline{AB} = \underline{A} + \overline{A}\underline{B} \tag{4}$$

Contents

List of Tables

List of Figures

List of Algorithms

Listings

Chapter 1

Introduction

Characters, be they animals, humans, or alien creatures, bring life to computer animation. There are many examples of characters in films that are created with computer graphics. Alien creatures, such as the giant bugs in *Starship Troopers*, are a common application, as the only reasonable alternative is expensive animatronics. Character animation is also used for humanoid characters: the character Gollum in the film adaptation of the *Lord of the Rings* trilogy is played by an actor (Andy Serkis) wearing a motion capture suit [Serkis, 2003]; the on-screen character is generated by computer graphics and mostly matched to the motions of the actor. In some cases, characters may be produced both with real live actors and computer graphics: in *The Matrix Reloaded*, the climatic battle between Neo and Agent Smith was created entirely with computer graphics, using detailed scans of the actors [Borshukov et al., 2003], their environment, and even the reflectance properties of their clothes [Borshukov, 2003]. Animated characters are also used to simulate large crowds; huge battles were staged in the *Lord of the Rings* movies (some shots having over 200 000 characters), using software that simulated a simple "brain" for each character [Aitken et al., 2004].

Another major application area for character animation is in computer games. In the first-person shooter genre, these are usually enemies or monsters that need to be killed, and this genre has generally set the standard for detail and realism in the industry. However, other gaming genres are progressing rapidly; for example, many real-time strategy games now use detailed three-dimensional characters, making it possible to zoom in to the scene of a battle. Although the purpose is different, many of the same problems and solutions are found in the creation of virtual environments, which may be used for simulation, training or education.

The traditional work-flow for creating a three-dimensional animated character consists of four steps: modelling, rigging, skinning and animating. A modeller creates the geometry of the character in a single pose, called the rest pose. This might be done entirely using a software package, but it is also possible to capture a model created in the real world — either an actual character, such as a person, or a physical model, such as a clay sculpture.

Rigging consists of creating controls for an animator to manipulate. Typically, this is done by

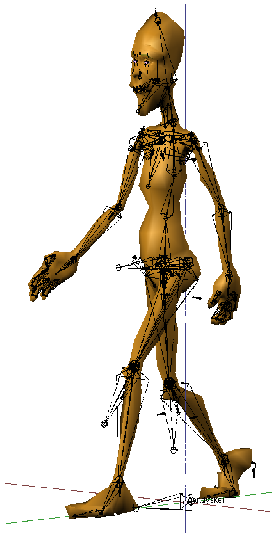


Figure 1.1: A complex rig, or skeleton, created in Blender. The black outlines show the bones that the animator positions to control the character.

inserting a skeleton into the model. There may also be non-skeletal controls, such as controls for facial actions (winking, smiling, etc.) and controls to determine the point at which the eyes will look. Figure 1.1 shows an example of a complex rig.

Skinning is the process of connecting the "skin", namely the geometry created by the modeller, to the rig. It is possible to at least partially automate this by attaching every vertex to nearby bones in the rig, but this usually does not give adequate results. In order for the skin to look natural, it must deform smoothly around joints, and extensive manual adjustment may be required. While the artist controls per-vertex parameters, an underlying algorithm is required to turn the settings of the controls into a transformation of the skin. Many such algorithms exist, from complex anatomically-based systems that model bone, muscle and fat (most applicable to high-quality off-line rendering, such as film) to algorithms that simply blend transformations together.

Finally, to bring the character to life it must be animated — something must move the rig. In film, and in many cases for games as well, the motion is determined by the artist in advance. Inverse kinematics (IK) can simplify this process by allowing the artist to place the extremities (for example, placing a hand on a button or a foot on the floor), and algorithmically determining the joint angles necessary to achieve this. More recently, games have begun to use real-time physics simulation to control the skeleton in some cases: a standard example is so-called "rag-doll" physics, in which a corpse may be pushed down a flight of stairs, with limbs bending and twisting in response to gravity and collisions.

The focus of this thesis is on skinning, and in particular on real-time skinning. There are two main reasons that a simple, lower-quality framework might be preferred over a more complex and realistic one. The first is speed: while anatomically-based physical simulations are ideal for the off-line rendering used in film, and also necessary for the level of realism that is required of the

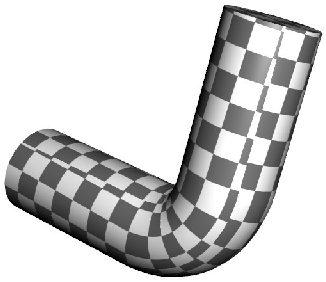


Figure 1.2: An example of a parametrisation. A chequerboard pattern is mapped back to the surface of the cylinder using the parametrisation function.

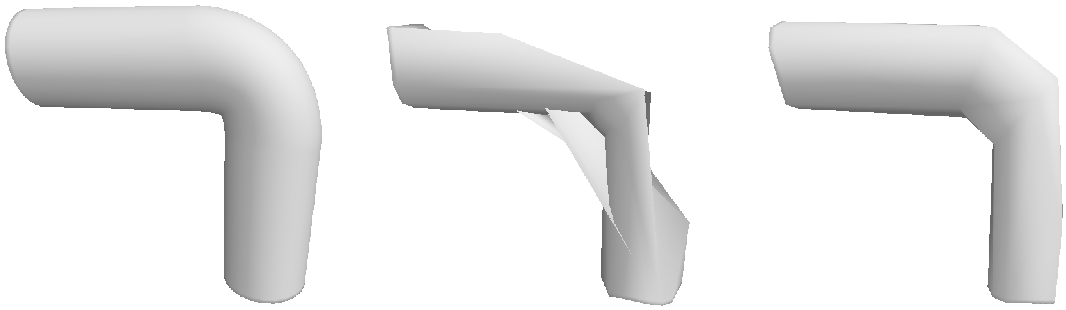


Figure 1.3: The effect of computing level-of-detail from the rest pose. Left: a full-detail model under animation, with the rest pose being a straight cylinder. Middle: a simplification computed from only the rest pose. Right: a simplification computed taking all the poses into account.

major characters, they are generally far slower than the algorithms used for games and other real-time applications. Secondly, the price of more generality is that more work is required to create a model; it is necessary to find a balance between generality and ease of modelling that suits the needs of the application in question.

Animation of any kind also introduces new challenges when combined with other areas of computer graphics; particularly areas that have traditionally considered only static scenes. For example, parametrisation maps the surface of an object to a simple 2D domain, aiming to minimise distortion, usually for the purposes of mapping a 2D function onto the surface. (Figure 1.2 shows an example). In the past, algorithms have considered only the rest pose of an object. However, the distortion depends on geometric properties and so the optimal parametrisation for the rest pose may not be optimal for the whole range of poses for the model. Similarly, level-of-detail techniques create a sequence of lower-detail approximations to a mesh. Traditionally, they have only been applied to the rest pose, but as can be seen in Figure 1.3, this may lead to very poor results during animation. More recently there have been some techniques that take multiple poses into account [Mohr and Gleicher, 2003a, Huang et al., 2006, DeCoro and Rusinkiewicz, 2005], but these are based on sampling the space of poses rather than examining the underlying structure of the animated character.

Our contribution is *animation space*, a character skinning framework described by a simple and elegant linear equation. Vertices are defined at fixed locations in a high-dimensional space (the

"animation space"), and projected to 3D during rendering. We show that despite this simple formulation, animation space has many attributes that are desirable in practice:

- Animation space is a strict generalisation of skeletal subspace deformation (SSD), a simple technique popular for real-time rendering. It follows that the algorithms developed in this thesis may be immediately applied to a large existing base of models.
- SSD is an extremely limited framework with well-known flaws. Like most subsequent skinning frameworks, animation space is able to address these flaws, while adding a relatively small number of degrees of freedom.
- We develop distance metrics on animation space that are independent of any particular pose. This allows the parametrisation and level-of-detail problems mentioned above to be tackled directly in animation space, rather than on a sample of poses in 3D space. The metrics also have other applications, such as visibility culling.
- It is possible to apply subdivision processes directly within animation space, and the linearity guarantees that the result is identical to applying the subdivision to a 3D pose.

The remaining chapters are organised as follows:

- Chapter 2 provides background on character animation techniques relevant to the thesis as a whole. Each chapter also begins with some background relevant to that chapter.
- In Chapter 3 we introduce the fundamentals of animation space, and derive the metrics referred to above.
- Chapter 4 demonstrates several methods of creating an animation-space model. This includes fitting a model to a set of example poses, which allows a modeller to easily generate animation-space models without any knowledge of the animation-space algorithm. We also show that the linear nature of animation space allows a natural application of subdivision.
- In Chapters 5 and 6 we show the practical benefits of the distance metrics developed in Chapter 3. Chapter 5 considers parametrisations that minimise stretch, while Chapter 6 considers level-of-detail approximations that minimise error. In both cases, animation space makes it possible to make measurements in a pose-independent manner to obtain solutions that work well across a range of poses.
- The use of animation space adds some complications to real-time rendering. We detail our implementation in Chapter 7, including the problem of reproducing full-detail shading on simplified models, and a method for determining a bounding volume that is guaranteed to contain any pose of the model.
- We finish with results in Chapter 8 and conclusions in Chapter 9.

Chapter 2

Background

There are a number of ways to animate characters [Collins and Hilton, 2001]. This thesis is concerned solely with skeletal animation, but we briefly describe other methods here for comparison.

Direct methods

The simplest method (algorithmically speaking) is to have the animator position each vertex or control point manually. However, while this is the most general method, it is also the most painstaking, as there is no high-level control.

Skeletal animation

Animation can be simplified by modelling a character in layers. More complex systems use multiple layers (such as bone, muscle, fat and skin), but simpler systems simply use bones and skin. The bones do not have geometry; they are a simple "stick-figure" representation of the character. There are two aspects to skeletal animation:

- The skin must be attached to the bones in some non-rigid way, so that it deforms smoothly as joints are bent. This process is called *skinning*, and is the primary focus of this thesis.
- The animator controls the skeleton in some way. This can be done directly, but for some applications it is preferable to position only certain key bones (usually extremities), and solve for the joint angles automatically. This latter process is known as *inverse kinematics* and is an active area of research; refer to Buss [2004] for a survey.

Shape interpolation

Shape interpolation is a method popular for facial animation. During modelling, a face is placed in a number of poses, such as smiling, frowning, neutral and so on. The animator controls the

expression of the face by interpolating between these poses. With simple linear blending, shape interpolation can be written as

$$\mathbf{v} = \sum_{i=1}^{n} a_i \mathbf{v}_i, \tag{2.1}$$

where i runs over the modelled poses, $\mathbf{v}_i$ is the position of a vertex in pose i, a_i is the weight given to pose i, and $\mathbf{v}$ is the final position.

Shape interpolation on its own is not suitable for body animation, because it works in a global coordinate system that does not consider rotations of joints. With linear interpolation, this would result in vertices moving along straight lines, rather than arcs, when interpolating between poses.

Spatial deformation

Spatial deformations use a control structure such as a lattice [Sederberg and Parry, 1986] or curve [Singh and Fiume, 1998] to define a transformation of the 3D space, along with objects embedded in it. Spatial deformations are not specific to character animation, but have been adapted to it, for example, by Chadwick et al. [1989] and Forstmann and Ohya [2006].

Physical simulation

The most accurate, but also the slowest, animations are produced by modelling the laws of motion in a body. A typical system models multiple layers (bone, muscle, fat and skin) with a mass-spring system, using Newton's laws to simulate the motion of each layer; another approach is to minimise an elastic energy of the skin surface [Collins and Hilton, 2001]. Because these approaches require the solution of large systems of differential equations, they are generally not used for real-time applications.

2.1 Notation for skeletal animation

In order to discuss skeletal animation in detail, we will require some notation. For modelling, bones are represented as line segments, but for analysis and implementation purposes it is more useful to consider the *frames* (coordinates systems) that they define. Each bone defines a local coordinate system with its own origin (one end-point of the bone) and axes (relative to the direction of the bone). The joints define the transformations between parent and child frames.

The number of bones will be denoted by b, and the bones (and their frames) are numbered from 0 to $b-1$. The parent of bone j is denoted $\phi(j)$. We also make the following assumptions:

- There is a single root bone. Some modelling systems allow multiple root bones, but this can be resolved by adding a new bone to act as a "super-root".

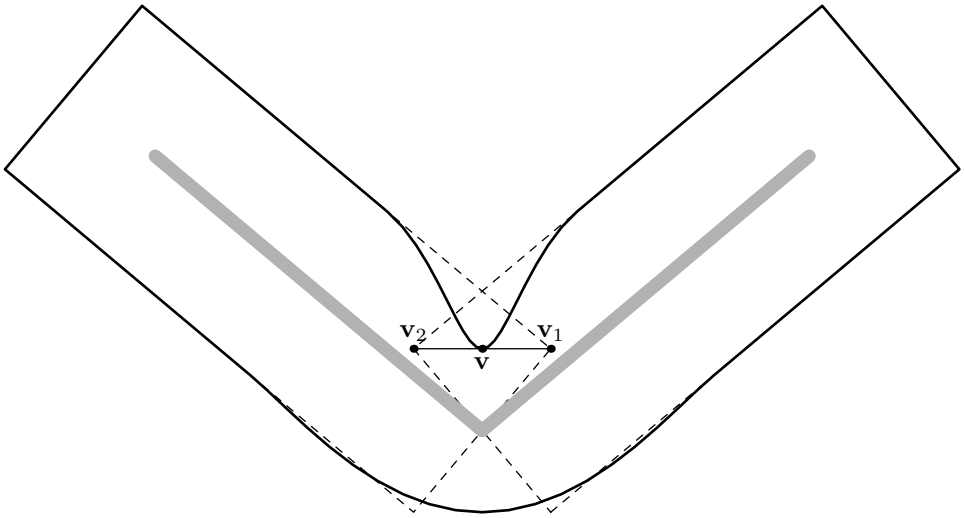


Figure 2.1: Illustration of skeletal subspace deformation. An originally straight limb is bent using SSD. Bones are shown in grey, and the rigid transformation of each half is shown by dashed lines. This produces $\mathbf{v}_1$ from the left bone and $\mathbf{v}_2$ from the right bone. The final position $\mathbf{v}$ is a linear combination of the two.

- The root bone does not undergo character animation, and the root frame is thus equivalent to model space. This does not restrict the animator, as transformation of the root bone is equivalent to a rigid transformation of the model as a whole.

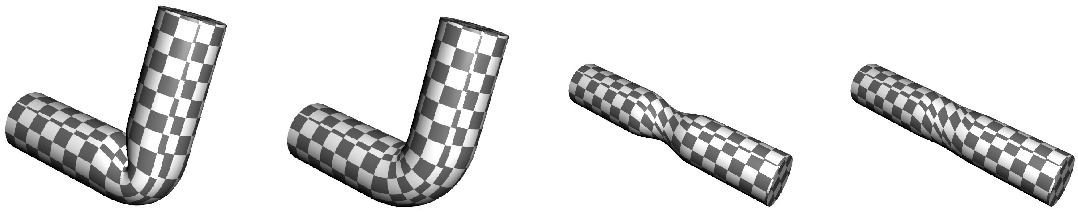
We will always label the root bone as bone 0. For each frame j, G_j is the matrix that transforms coordinates in that frame to model space (so $G_0 = I$, for example).

Modelling is done in a single pose, known as the rest pose (or dress pose). Values in this pose will be indicated with a hat (i.e., $\hat{\mathbf{v}}$, $\hat{G}$), while dynamic values will have no hat.

2.2 Skeletal subspace deformation

Skeletal subspace deformation (SSD), also known as enveloping, tweening, and linear blend skinning, is the most popular form of character skinning for computer games and other real-time environments. Its popularity is largely due to its simplicity and ease of hardware acceleration. The standard form in which it is used was not originally published in the literature, although the idea is generally credited to Magnenat-Thalmann et al. [1988].

In a model made of rigid pieces, it is sufficient to associate each vertex with a single bone in the skeleton; this corresponds to the dashed rectangles in Figure 2.1. However, this is insufficient for organic characters, whose skin stretches over joints. To accommodate this, SSD allows a vertex to be associated with (or *influenced by*) multiple bones. The vertex is transformed separately by each influencing bone, after which these transformed positions are linearly combined.



(a) (b) (c) (d)

Figure 2.2: Collapsing elbow and candy wrapper effects. (a), (c) Skeletal subspace deformation, exhibiting volume loss around the joint. (b), (d) Animation space, essentially free of these effects.

2.2.1 The SSD equation

We now define SSD more formally. The SSD equation is applied to each vertex independently, so we will consider only a single vertex $\mathbf{v}$ in homogeneous space. Let its position in the rest pose be $\hat{\mathbf{v}}$ and its influence from bone j be w_j. First, $\hat{\mathbf{v}}$ is transformed into the various local frames:

$$\hat{\mathbf{v}}_j = \hat{G}_j^{-1}\hat{\mathbf{v}}.$$

These local coordinates are then converted back into model-space coordinates, but this time using the dynamic positions of the bones (i.e., their positions in the current frame):

$$\mathbf{v}_j = G_j\hat{\mathbf{v}}_j = G_j\hat{G}_j^{-1}\hat{\mathbf{v}}.$$

Finally, these model-space positions are blended using the bone weights, as illustrated in Figure 2.1:

$$\mathbf{v} = \sum_{j=0}^{b-1} w_j G_j \hat{G}_j^{-1}\hat{\mathbf{v}} \quad \text{where} \sum_{j=0}^{b-1} w_j = 1. \tag{2.2}$$

2.2.2 Limitations of SSD

An alternative interpretation of SSD is found by rewriting (2.2) with different grouping, giving

$$\mathbf{v} = \left(\sum_{j=0}^{b-1} w_j G_j \hat{G}_j^{-1}\right)\hat{\mathbf{v}}. \tag{2.3}$$

In this form, the weights are seen as blending factors between transformation matrices. However, a linear combination of rigid transformations need not be rigid, and as a result, SSD has some well-known problems [Kavan et al., 2007]. The "collapsing elbow" and the "candy wrapper" effects are shown in Figures 2.2(a) and 2.2(c). They are caused by the linear blending of rotation matrices that have a large angle between them.

A less visible problem, but an important one in our approach, is that one cannot synthesise new

vertices as affine combinations of old ones. For example, suppose one wished to create a vertex that was the midpoint of two existing vertices. Averaging the rest positions and weights will not work, because they combine non-linearly in equation (2.2). This implies that a modeller needing extra vertices cannot simply subdivide a mesh; doing so will alter the animation, even if the new vertices are not displaced. As will be seen in Chapter 5, it is also useful to be able to take the difference between two points; however, the behaviour of the vector between two SSD vertices cannot be described within the SSD framework.

2.3 Improvements to SSD

Most methods for real-time character animation start with SSD as a basis and modify it to address the shortcomings, while aiming to preserve the simplicity and efficiency of SSD. The approaches can be grouped into several categories:

2.3.1 Example-based techniques

Example-based techniques use a database of sculpted examples, either manually produced by an artist, or generated from a physically-based modelling system (such as a finite element model, as used by Kry et al. [2002]). Each example has an associated pose, and the database should exercise the full range of each joint. The example meshes are compared to meshes generated with SSD in corresponding poses, and difference vectors are stored. During rendering, scattered data interpolation generates a new difference vector that adjusts the result of SSD, producing a higher quality model.

Lewis et al. [2000] were the first to propose such a method, pose space deformation (PSD). In their system, the underlying model is either rigid, or articulated but with rigid pieces (i.e., the skin is not smoothed around joints). Interpolation is done within a *pose space*; generally the axes of pose space correspond to joint angles in the model, but other types of control (such as "smile" for facial animation) can be included as well. Interpolation is handled with Gaussian radial basis functions, where the distance is simply the Euclidean distance within pose space.

Sloan et al. [2001] make several improvements to pose space deformation. They improve the data interpolation by starting with a best-fitting hyperplane, and using radial basis functions to adjust this hyperplane. The interpolation is also optimised by working in the space spanned by the provided examples, with those examples forming a basis. Finally, they use SSD as the underlying model; all interpolation is done in the rest pose, with the inverse of SSD applied to the examples to bring them into the rest pose, as shown in Figure 2.3.

Eigenskin [Kry et al., 2002] makes two further improvements. Firstly, it identifies the *support* of each joint — the region in which a change in the joint angle causes significant deviation. Secondly, within each joint's support, an eigenbasis is constructed, and only the dominant eigenvectors are retained. This significantly reduces the number of variables that must be interpolated. Limiting

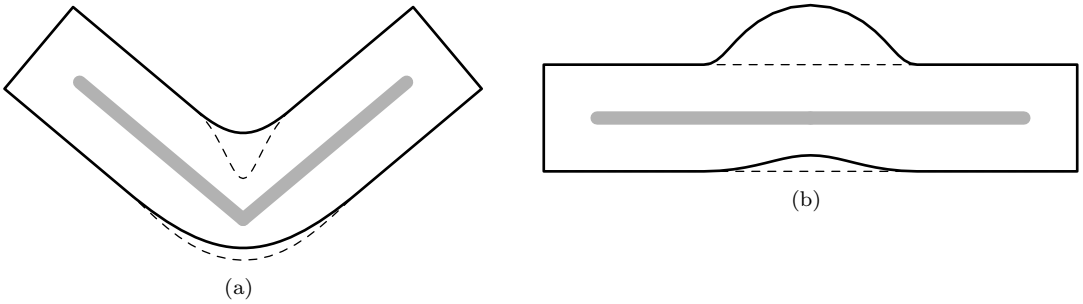


Figure 2.3: Shape-by-example. (a) An example provided by the artist. (b) The inverse of SSD is applied to bring the example into the rest pose. The dashed lines indicate the results with plain SSD.

the number of eigenvectors that are retained also makes it possible to pass all the necessary data to a vertex shader, thus offloading computation from the CPU.

Kurihara and Miyata [2004] introduce weighted pose space deformation (WPSD), in which the distance function in pose space becomes a per-vertex function. In PSD, the distance between two poses **a** and **b** is defined as

$$d(\mathbf{a}, \mathbf{b})^2 = \sum_{j=0}^{b-1} (a_j - b_j)^2, \tag{2.4}$$

while in WPSD it is defined for vertex i as

$$d_i(\mathbf{a}, \mathbf{b})^2 = \sum_{j=0}^{b-1} u_{i,j}(a_j - b_j)^2. \tag{2.5}$$

The weights $u_{i,j}$ determine the importance of each joint angle to vertex i; Kurihara and Miyata [2004] report good results from simply using the SSD weights. The effect of this non-uniform weighting is that the examples used to interpolate a vertex on an arm, for example, are chosen according to the similarity of their poses in the arm region rather than global similarity, making it possible to obtain good results with fewer poses. Unfortunately, evaluating the radial basis functions on a per-vertex basis adds considerable computational cost.

These example-based techniques can produce high-quality models given a training set that spans the range of motion, but conversely, the modeller must produce such a representative training set. Steps must also be taken to avoid over-fitting the model; for example, there may happen to be a correlation in the examples between a vertex on one leg and the pose of the other leg, but this is an artefact of the data and should not be generalised.

2.3.2 Extra bones

Mohr and Gleicher [2003b] introduce pseudo-bones into their models. Each joint is given a pseudo-bone that is rotated by half of the real joint angle. Vertices can be attached to this bone to receive

this half-way rotation; since this is a spherical rather than linear interpolation, it avoids the collapsing effect seen in Figures 2.1 and 2.2. They also introduce pseudo-bones that scale rather than rotate, to support muscle bulging effects. The advantage of adding bones is that only the model, and not the framework, is altered; however, the number of bone influences per vertex increases, which reduces rendering performance. It may also complicate hardware acceleration, as a straightforward implementation only allows a fixed number of attributes to be passed per vertex.

2.3.3 Non-linear transformation blending

While we have described SSD as a linear blending of vertices, it can also be seen as a linear combination of the bone matrices (Equation 2.3), and the defects that occur arise because linear interpolation of rotation matrices does not correspond to an interpolation of their rotations. Recently, several techniques have been proposed which use non-linear transformation blending techniques. Magnenat-Thalmann et al. [2004] use the matrix blending operator of Alexa [2002] to perform the interpolation; this improves the results, but is an expensive method (it is based on the matrix exponential and logarithm). Spherical blend skinning [Kavan and Žára, 2005] addresses the problem by computing linear interpolations of quaternions; the authors show that this is a good approximation to spherical interpolation. This does not allow for interpolation of scaling transformations, but a far more serious shortcoming is that blending translations between more than two transformations is done in an ad-hoc and expensive manner. The same authors subsequently solved this problem using dual quaternions [Kavan et al., 2007], which elegantly handle translations and rotations in a single framework.

2.3.4 Extra weights

Multi-weight enveloping (MWE) [Wang and Phillips, 2002] is the method most similar to our approach. In order to explain MWE, we need to expand Equation (2.2). For each bone j, let $N_j = G_j\hat{G}_j^{-1}$. This matrix defines the motion of the bone relative to its rest position. Substituting N_j into Equation (2.2) gives

$$\mathbf{v} = \sum_{j=0}^{b-1} w_j N_j \hat{\mathbf{v}} = \sum_{j=0}^{b-1} w_j \begin{pmatrix} n_{j,11} & n_{j,12} & n_{j,13} & n_{j,14} \\ n_{j,21} & n_{j,22} & n_{j,23} & n_{j,24} \\ n_{j,31} & n_{j,32} & n_{j,33} & n_{j,34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \hat{\mathbf{v}}. \tag{2.6}$$

Multi-weight enveloping assigns a weight to each entry of the matrix N_j [Wang and Phillips, 2002], giving the equation[1]

$$\mathbf{v} = \sum_{j=0}^{b-1} \begin{pmatrix} w_{j,11}n_{j,11} & w_{j,12}n_{j,12} & w_{j,13}n_{j,13} & w_{j,14}n_{j,14} \\ w_{j,21}n_{j,21} & w_{j,22}n_{j,22} & w_{j,23}n_{j,23} & w_{j,24}n_{j,24} \\ w_{j,31}n_{j,31} & w_{j,32}n_{j,32} & w_{j,33}n_{j,33} & w_{j,34}n_{j,34} \\ 0 & 0 & 0 & \frac{1}{b} \end{pmatrix} \hat{\mathbf{v}}. \tag{2.7}$$

These extra weights allow for non-linear effects, and Wang and Phillips [2002] show that the extra degrees of freedom make it possible to avoid the collapsing elbow and candy-wrapper effects. They use a modified least-squares method to fit the weights to a database of examples; unlike with the example-based techniques, the model approximates rather than interpolates the examples. However, the rest pose configuration and vertex positions are determined by the user, presumably selected from the database of examples.

MWE uses 12 weights per bone influence, while no joint has that many degrees of freedom. This leads to problems with correlation and over-fitting during the fitting process. These are handling by applying principal component analysis (PCA) [Jolliffe, 1986] and local ridge regression during the fitting process [Wang and Phillips, 2002]. Note that while PCA works best with linear correlations, the authors apply it to the entries of rotation matrices, which have non-linear relationships. The large number of weights also increases storage and bandwidth costs.

2.3.5 Comparison of methods

Generality

Example-based techniques are able to represent the widest range of potential animations, because they can interpolate any number of given poses exactly. Multi-weight enveloping has a fixed number of degrees of freedom, so it can only approximate example poses, but it nevertheless has significantly more freedom than SSD. The non-linear transformation blending methods have exactly the same degrees of freedom as SSD, making them the least general. Extra bones introduce more degrees of freedom; the amount of freedom will obviously depend on the number and type of the extra bones.

It should be noted that more degrees of freedom can be a disadvantage: with more degrees of freedom there is also more work in specifying them.

Animation space has a fixed number of degrees of freedom, between that of SSD and MWE.

[1] The original paper used 1 in place of $\frac{1}{b}$ in equation (2.7), but this gives $\mathbf{v}$ a weight of b rather than the more conventional 1. The difference is just a scale factor, so using this equation does not alter the method.

Rendering efficiency

It is difficult to quantify the efficiency of these methods, since no author has tested all the methods with a single data set and hardware configuration. We can only report isolated results from the literature:

- Rhee et al. [2006] implemented both PSD and WPSD with hardware acceleration and also incorporate elements of Eigenskin [Kry et al., 2002]. They found PSD and WPSD to run at about 85% and 5% as fast as SSD respectively.
- The dual quaternion implementation of Kavan et al. [2007] has performance of 70% of SSD, again using hardware acceleration.
- The authors of multi-weight enveloping did not report rendering speeds, but we created a basic hardware implementation (described in Section 8.6.1) where speeds ranged from 47% to 61% of that of SSD.

Surprisingly, our results show that animation-space models are faster to render than SSD models.

Ease of use

A major strength of the non-linear transformation interpolation methods is that they can be applied to existing SSD models with no changes. For creating new models, the limitations of SSD still apply: an indirect interface, and the inability to place vertices outside of some subspace. However, it is likely that the concepts used in direct manipulation interfaces [Mohr et al., 2003] and example-fitting [Mohr and Gleicher, 2003b, James and Twigg, 2005] could be adapted to these methods.

The other methods, as well as animation space, are all defined by a set of examples. While this is a more direct and intuitive interface, it is also time-consuming, as it is necessary to create the model in a variety of poses that exercises the full range of motion of all the joints.

2.4 Data acquisition

A skeletal animation framework is only useful if it is possible to obtain data to be used with the framework. Several types of data are required: the shape of the model (possibly only a single pose, but possibly a database of examples), a skeleton, and the skinning weights. We briefly review common methods here; for further references, consult the survey by Collins and Hilton [2001].

2.4.1 Geometry

A model can either be captured from the physical world (for example, a human subject, or a clay model of a more fanciful creature), designed entirely within a computer, or create by a hybrid of

the two. Common techniques for capturing real-world objects are laser range scanning [Curless, 2000], stereopsis [Faugeras, 1993] and shape-from-silhouette [Potmesil, 1987]. A problem that arises when multiple scans of the same subject are used with example-based techniques (discussed in Section 2.3.1), is that the models must be in *correspondence*; that is, they must have the same vertices and connectivity, so that it is meaningful to consider the position of a particular vertex in each of the poses. However, each scanned model is reconstructed independently, and may not even have the same number of vertices. The usual approach to bring meshes into correspondence is to use one of them as a template, then find deformations that transform this template to be as close as possible to the others. For rigid bodies, the classical method is the Iterated Closest Points algorithm [Rusinkiewicz and Levoy, 2001], but it is not suitable for character animation as characters undergo non-rigid animation. Allen et al. [2002] place markers on a body and use these to solve for the pose of a pre-defined skeleton, and use a pre-defined subdivision surface as the template. Anguelov et al. [2004a] obtain excellent results using expectation-maximisation with a probability function that aims to preserve both local and global geodesic distances, without requiring markers.

Computer modelling suffers from none of these issues. It is a mature field; numerous free and commercial modelling packages exist (Blender, 3D Studio MAX, Maya and Lightwave to name a few of the better-known ones), featuring a plethora of editing tools too numerous to review here.

2.4.2 Bones

Creating a skeleton for a model is relatively easy to do by hand, because it is simply a stick figure and does not require the same fine detail as the skin. Manually defining a skeleton also gives the animator control over the complexity of the skeleton: for example, areas like the shoulder and spine actually consist of multiple joints [Allen et al., 2002], but it may be desirable to use a simpler skeleton to ease animation.

Nevertheless, several automated techniques exist to extract skeletons from a set of example meshes. James and Twigg [2005] consider the rotations applied to move triangles from one pose to another. Triangles that are part of the same rigid section will undergo similar transformations, so bones are identified as clusters in the space of rotations. Note, however, that this only identifies the transformations associated with a bone, but it does not identify endpoints or a hierarchy. Anguelov et al. [2004b] use a statistical model that measures the probability of an assignment of vertices to rigid sections together with transformations of those sections, and perform an expectation-maximisation to optimise the assignments. Each rigid section corresponds to a bone, and the authors also solve for the positions of the joints.

2.4.3 Weights

The conventional method of assigning weights is to have the artist "paint" them onto the surface of a model. This is an indirect and often frustrating approach, as the artist cannot simply place

vertices where they need to go. Mohr et al. [2003] have developed a direct manipulation interface, in which the user may position a vertex in some pose, and the weights are then calculated to match the user's specification as closely as possible. Mohr and Gleicher [2003b] use an example-based technique, in which both the weights and the rest positions are determined using a bilinear solver to match a given set of examples. James and Twigg [2005] take the rest positions as given, and solve a non-negative least-squares problem to optimise the weights.

Chapter 3

Animation space

In the previous chapter, we described skeletal subspace deformation (SSD), a simple and efficient method for character animation. We also described a generalisation, multi-weight enveloping (MWE), that reduces the more glaring deficiencies in SSD. While these methods are linear in the bone matrices (and indeed, SSD is also known as linear blend skinning), they are non-linear in the vertex coordinates since the weights are multiplied with the rest-pose positions.

This chapter introduces *animation space*, an animation framework whose generality lies between that of SSD and MWE, and for which the animation is linear in both the matrices and the vertex coordinates. We start by introducing animation space, motivating it as an alternative way to express SSD which is also more general. The bulk of this chapter, however, is the derivation of a set of metrics on animation space with meaningful geometric properties. Subsequent chapters will identify applications for these metrics.

After deriving the metrics, we show that the probability density functions we used to compute them are not required in order to make global alterations to the model: the expectations we computed contain all the information necessary for the metrics to be updated.

We end the chapter by considering the problem of normals. Since animation space has a high dimension, there is no normal vector to a 2D surface. Instead, we use tangent planes to model derivatives.

3.1 Reformulating SSD

Let us re-examine equation (2.2) on page 8. We can combine the weight, the inverse rest-pose matrix and the vertex into a single vector, and write

$$\mathbf{v} = \sum_{j=0}^{b-1} G_j \mathbf{p}_j, \tag{3.1}$$

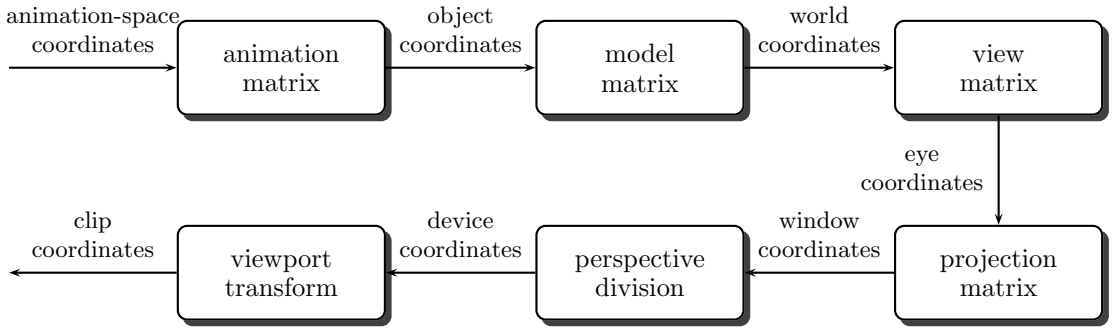


Figure 3.1: Coordinate systems and the transformations between them

where $\mathbf{p}_j = w_j \hat{G}_j^{-1} \hat{\mathbf{v}}$. We can rearrange this as a matrix-vector product as follows:

$$\mathbf{v} = \begin{pmatrix} G_0 & G_1 & \cdots & G_{b-1} \end{pmatrix} \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_{b-1} \end{pmatrix} = G\mathbf{p}. \tag{3.2}$$

One can view the vector $\mathbf{p}$ on the right as the coordinates of $\mathbf{v}$ in a multi-dimensional space, which we will call *animation space*. This space is $4b$-dimensional. The left-hand matrix converts from animation space to model space; we label it G and refer to it as the *animation matrix* and its operation as *animation projection*. Figure 3.1 illustrates the place of animation projection in a typical rendering pipeline, such as that used in the OpenGL API[1] [Segal and Akeley, 2006]. Animation projection does not produce final screen coordinates; rather, it produces coordinates in a local model space, which may be further manipulated by the model matrix. This allows rigid movement of the character (running, falling off a cliff, etc.) to be separated from control of the limbs. Also note that the first four stages are all matrix multiplications, so any subsequence of them may be collapsed into a single matrix if desired.

For a general element of the animation space, we find that

$$\underline{\mathbf{v}} = \underline{\mathbf{p}_0} + \cdots + \underline{\mathbf{p}_{b-1}}$$

(recall from page v that $\underline{\mathbf{v}}$ is the homogeneous coordinate of $\mathbf{v}$). We thus refer to the right-hand side as the *weight* of $\mathbf{p}$, and also denote it $\underline{\mathbf{p}}$. Just as we usually work with points in the 3D hyperplane $\underline{\mathbf{v}} = 1$, we will generally work with points in the animation-space hyperplane $\underline{\mathbf{p}} = 1$. In the context of equation (3.1), this simply says that the sum of the weights affecting a vertex is 1. The restriction to this hyperplane still gives us $4b - 1$ degrees of freedom, while standard SSD has only $b + 2$ degrees of freedom ($b - 1$ independent weights plus the coordinates of $\mathbf{v}$).

An enormous advantage of this formulation over SSD is that it is truly linear; that is, linear combinations of vertices distribute over animation projection, e.g., $G(\frac{1}{2}\mathbf{p} + \frac{1}{2}\mathbf{q}) = \frac{1}{2}G\mathbf{p} + \frac{1}{2}G\mathbf{q}$. In

[1] OpenGL combines the model and view matrices into a single *model-view* matrix.

particular, every point in a triangle mesh can be expressed as a convex combination of the three vertices forming one of its triangles, and hence has coordinates in animation space, not just the vertices. A further benefit is that subdivision surfaces can be computed once in animation space, then projected into model space (see Section 4.2).

3.2 Comparison to multi-weight enveloping

We can apply the same combining technique (that merged w_j and $\hat{\mathbf{v}}$) to multi-weight enveloping. Referring to equation (2.7), let $u_{j,rs} = w_{j,rs}\hat{v}_s$ where $1 \le r \le 3$ and $1 \le s \le 4$. Then

$$\mathbf{v} = \sum_{j=0}^{b-1} \begin{pmatrix} u_{j,11}n_{j,11} + u_{j,12}n_{j,12} + u_{j,13}n_{j,13} + u_{j,14}n_{j,14} \\ u_{j,21}n_{j,21} + u_{j,22}n_{j,22} + u_{j,23}n_{j,23} + u_{j,24}n_{j,24} \\ u_{j,31}n_{j,31} + u_{j,32}n_{j,32} + u_{j,33}n_{j,33} + u_{j,34}n_{j,34} \\ \frac{1}{b} \end{pmatrix} \tag{3.3}$$

Unlike SSD, MWE does not gain any generality from this transformation, as it can be reversed by setting $W_j = U_j$ and $\hat{\mathbf{v}} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}^T$. This also reveals that the so-called rest positions are in fact almost arbitrary, as weights can be found to match any rest position that does not have a zero component. From this formulation, we see that MWE is a superset of animation space: if $\mathbf{p}$ is a position in animation space and each row of U_j equals $(\hat{G}_j\mathbf{p}_j)^T$, then

$$\begin{aligned} \mathbf{v} &= \sum_{j=0}^{b-1} \begin{pmatrix} u_{j,11}n_{j,11} + u_{j,12}n_{j,12} + u_{j,13}n_{j,13} + u_{j,14}n_{j,14} \\ u_{j,21}n_{j,21} + u_{j,22}n_{j,22} + u_{j,23}n_{j,23} + u_{j,24}n_{j,24} \\ u_{j,31}n_{j,31} + u_{j,32}n_{j,32} + u_{j,33}n_{j,33} + u_{j,34}n_{j,34} \\ \frac{1}{b} \end{pmatrix} \\ &= \sum_{j=0}^{b-1} \begin{pmatrix} n_{j,11} & n_{j,12} & n_{j,13} & n_{j,14} \\ n_{j,21} & n_{j,22} & n_{j,23} & n_{j,24} \\ n_{j,31} & n_{j,32} & n_{j,33} & n_{j,34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \hat{G}_j\mathbf{p}_j \\ &= \sum_{j=0}^{b-1} N_j\hat{G}_j\mathbf{p}_j = \sum_{j=0}^{b-1} G_j\hat{G}_j^{-1}\hat{G}_j\mathbf{p}_j = \sum_{j=0}^{b-1} G_j\mathbf{p}_j = G\mathbf{p}. \end{aligned} \tag{3.4}$$

In practical terms, the extra generality of MWE allows a vertex to behave differently in different dimensions; for example, a vertex attached to a rotating joint may remain fixed in the x and y dimensions while moving sinusoidally in the z dimension. This has limited application, however, because these are global dimensions rather than the local dimensions of a bone frame, and hence any special effects obtained in this way will not be invariant under rotations of the model.

While MWE is more general than animation space, it is not necessarily better for all purposes. Most importantly, Equation (3.3) does not have the elegant form of the animation-space equation $\mathbf{v} = G\mathbf{p}$ which makes the analysis of the following sections possible. The extra weights also require

extra storage and processing, and do not necessarily contribute to the generality of the model: Wang and Phillips [2002] use Principal Component Analysis to reduce the dimension of the space.

3.3 Distance metrics

For several applications, such as level-of-detail and parametrisation, we need a measure of distance between two points in animation space. To be useful, such a metric should of course reflect some property of the distance in 3D model space, and should account for all poses rather than a single pose.

Instead of explicitly defining distances between $\mathbf{p}$ and $\mathbf{q}$, we define norms on $\mathbf{s} = \mathbf{p} - \mathbf{q}$. Since we are only interested in $\mathbf{p}$ and $\mathbf{q}$ with weight 1, we will define our norms only for animation vectors $\mathbf{s}$ with weight 0.

We describe two norms below, which we term the $L_{2,2}$ and the $L_{2,\infty}$ norms. The first index describes the spatial properties (both norms are related to the spatial Euclidean norm) and the second describes the aggregation over the space of poses: the former is a root-mean-squared distance, while the latter is an upper bound. In deriving algorithms to compute the $L_{2,2}$ norm, we will also develop some statistical models of animation space.

3.3.1 $L_{2,2}$ norm

Since the $L_{2,2}$ norm is a measure of *average* distance, we will need to make use of some statistics. In particular, we will use the expectation operator $E[\cdot]$, which gives the expected (mean) value of its argument. If x is a random variable with probability density function $p(x)$, then

$$E\big[f(x)\big] = \int p(x)f(x)dx. \tag{3.5}$$

Two important properties of $E[\cdot]$ are that it is linear and that $E[xy] = E[x]E[y]$ provided that x and y are independent variables.

We define the $L_{2,2}$ norm as

$$\|\mathbf{s}\|_{2,2} = \sqrt{E\left[\|G\mathbf{s}\|^2\right]}.$$

In this case, G is the random variable. Expanding the formula gives:

$$\begin{aligned}
\|\mathbf{s}\|_{2,2}^2 &= E\left[\mathbf{s}^T G^T G \mathbf{s}\right] \\
&= \mathbf{s}^T E\left[G^T G\right] \mathbf{s} \\
&= \mathbf{s}^T E\left[\begin{pmatrix} G_0^T G_0 & \cdots & G_0^T G_{b-1} \\ \vdots & \ddots & \vdots \\ G_{b-1}^T G_0 & \cdots & G_{b-1}^T G_{b-1} \end{pmatrix}\right] \mathbf{s}.
\end{aligned} \tag{3.6}$$

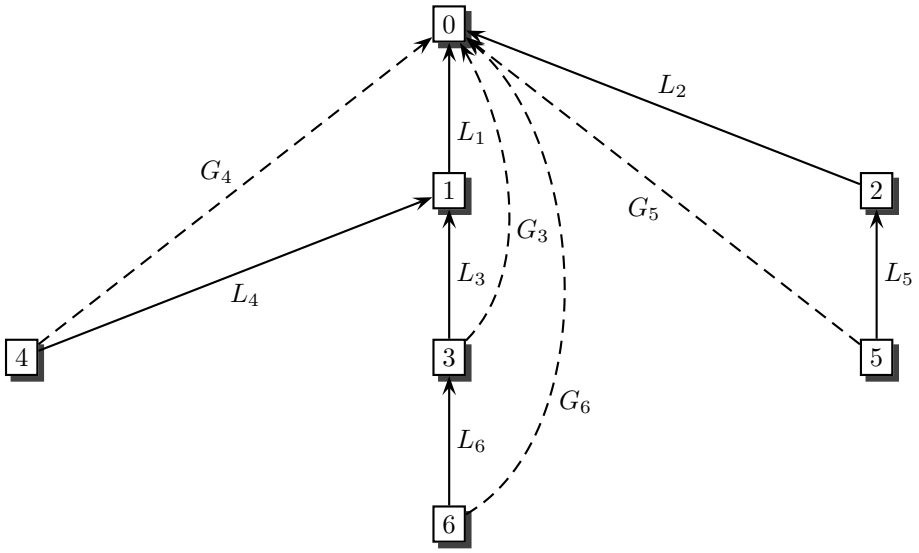


Figure 3.2: Relationships between frames. Solid arrows show child-to-parent relationships, modelled by the joint matrices L_i, while dashed arrows show frame-to-root relationships, modelled by the matrices G_i. G_1 and G_2 are the same as L_1 and L_2.

So far we have treated each frame as a separate entity, with no relationships between them, but for any meaningful statistical analysis we need to consider how the bones connect to each other. Each frame is animated either relative to a parent frame, or else is frame 0 (model space). Rather than work with the global transformations G_j (which transform directly to model space), we define L_j to be the local transformation which maps frame j to its parent (see Figure 3.2). In particular, if $j \neq 0$ then $G_j = G_{\phi(j)} L_j$ (recall that $\phi(j)$ is the parent of j). We will call these local matrices *joint* matrices, since they define the action of the joints connecting bones.

Let P be $E[G^T G]$; P is an expectation and hence independent of any particular pose of the model. To compute the expectation, we will need further information about how the skeleton will be animated. One option is to take samples of the pose space (such as from a recorded animation), and average the values of $G^T G$ together. However, in such a high-dimensional space, it is impractical to obtain a representative sampling. In order to make sampling practical, we make the simplifying assumption that different parts of the body are independent e.g., arms can move independently of legs. While not perfectly true (since body parts cannot interpenetrate), we believe that this is a sufficiently good approximation that it is justified by the benefits of being able to sample positions for each joint separately, rather than having to sample across all possible combinations of positions, and the results support this approach.

In the following subsections, we will examine some statistical models to derive P. We begin with a small number of assumptions about the statistical distributions, to obtain a fairly general model. More general models require more parameters that must be determined by other means, so we also show how we can add reasonable assumptions to reduce the amount of *a priori* knowledge required.

Independent joints

In this model we make the assumption that the joint matrices are independent. Rather than providing an explicit formula for $E[G_j^T G_k]$ (which is impractical, given that it depends on the tree structure), we provide an $O(b^2)$ algorithm to compute the values inductively. We begin with the assumption that we know $E[L_j]$ and $E[G_j^T G_j]$ for each joint j, since in this model they cannot be computed without further information. Later models will provides a means of calculating these.

In computing $E[G_j^T G_k]$, we start with the assumptions that $j \neq k$ (since otherwise we already have the value) and that k is not an ancestor frame of j (if it is, we can just swap the roles of j and k). These assumptions guarantee that L_k is independent of both $G_{\phi(k)}$ and of G_j, so that

$$E[G_j^T G_k] = E[G_j^T G_{\phi(k)} L_k] = E[G_j^T G_{\phi(k)}] E[L_k]. \tag{3.7}$$

We can apply this formula recursively to the first factor on the right, until we obtain a factor of the form $E[G_j^T G_j]$. For example, in the skeleton shown in Figure 3.2,

$$\begin{aligned} E[G_4^T G_6] &= E[G_4^T G_3] E[L_6] \\ &= E[G_4^T G_1] E[L_3] E[L_6] \\ &= E[G_1^T G_4]^T E[L_3] E[L_6] \\ &= \big(E[G_1^T G_1] E[L_4]\big)^T E[L_3] E[L_6] \\ &= E[L_4]^T E[G_1^T G_1] E[L_3] E[L_6]. \end{aligned} \tag{3.8}$$

From this point, we will consider statistical models on a per-joint basis. This allows several models to be used within a single skeleton. In each case, we need only show how to derive $E[L_j]$ and $E[G_j^T G_j]$ for the joint in question, as the algorithm above shows that this is sufficient to compute P.

Independent components

We can derive further results by assuming that the translation component of a joint matrix is independent from the rest of the matrix. It is not clear whether there is any use for this model in isolation, but the derivation is useful as a stepping stone for the next model (constant translations).

This time, we will start with the following known base information for each frame $j \neq 0$:

- $E[\overline{L_j}]$ and $E[\underline{L_j}]$;
- $E[\overline{G_j}^T \overline{G_j}]$, which is a measure of scale;
- $E[\underline{L_j}^T \overline{G_{\phi(j)}}^T \overline{G_{\phi(j)}} \underline{L_j}]$, which is a measure of the length of the bone in model space.

We now show how to compute $E[L_j]$ and $E[G_j^T G_j]$. Trivially,

$$E[L_j] = \begin{pmatrix} E[\overline{L_j}] & E[\underline{L_j}] \\ \mathbf{0}^T & 1 \end{pmatrix}. \tag{3.9}$$

To compute $E[G_j^T G_j]$, we first decompose it into its components.

$$\begin{aligned} E[G_j^T G_j] &= E\left[\begin{pmatrix} \overline{G_j}^T & \mathbf{0} \\ \underline{G_j}^T & 1 \end{pmatrix}\begin{pmatrix} \overline{G_j} & \underline{G_j} \\ \mathbf{0}^T & 1 \end{pmatrix}\right] \\ &= E\left[\begin{pmatrix} \overline{G_j}^T\overline{G_j} & \overline{G_j}^T\underline{G_j} \\ \underline{G_j}^T\overline{G_j} & \underline{G_j}^T\underline{G_j} + 1 \end{pmatrix}\right] \\ &= \begin{pmatrix} E[\overline{G_j}^T\overline{G_j}] & E[\overline{G_j}^T\underline{G_j}] \\ E[\underline{G_j}^T\overline{G_j}] & E[\underline{G_j}^T\underline{G_j}] + 1 \end{pmatrix}. \end{aligned} \tag{3.10}$$

Since G_0 is the identity (in which case the problem is trivial), we can ignore the case $j = 0$ and assume that j has a parent. From equation (4) on page vi, we have $\underline{G_j} = \underline{G_{\phi(j)} L_j} = \underline{G_{\phi(j)}} + \overline{G_{\phi(j)}}\underline{L_j}$. Substituting this into the bottom-right element of the matrix in (3.10) gives

$$\begin{aligned} E[\underline{G_j}^T\underline{G_j}] + 1 &= E[\underline{G_{\phi(j)}}^T\underline{G_{\phi(j)}}] + 2E[\underline{G_{\phi(j)}}^T\overline{G_{\phi(j)}}\underline{L_j}] + E[\underline{L_j}^T\overline{G_{\phi(j)}}^T\overline{G_{\phi(j)}}\underline{L_j}] + 1 \\ &= E[\underline{G_{\phi(j)}}^T\underline{G_{\phi(j)}}] + 2E[\underline{G_{\phi(j)}}^T\overline{G_{\phi(j)}}]E[\underline{L_j}] + E[\underline{L_j}^T\overline{G_{\phi(j)}}^T\overline{G_{\phi(j)}}\underline{L_j}] + 1. \end{aligned} \tag{3.11}$$

The expectations on the right-hand side are either assumed to be given, or else form part of $E[G_{\phi(j)}^T G_{\phi(j)}]$. Similarly,

$$\begin{aligned} E[\overline{G_j}^T\underline{G_j}] &= E[\overline{L_j}^T\overline{G_{\phi(j)}}^T(\underline{G_{\phi(j)}} + \overline{G_{\phi(j)}}\underline{L_j})] \\ &= E[\overline{L_j}^T\overline{G_{\phi(j)}}^T\underline{G_{\phi(j)}} + \overline{L_j}^T\overline{G_{\phi(j)}}^T\overline{G_{\phi(j)}}\underline{L_j}] \\ &= E[\overline{L_j}]^T E[\overline{G_{\phi(j)}}^T\underline{G_{\phi(j)}}] + E[\overline{L_j}]^T E[\overline{G_{\phi(j)}}^T\overline{G_{\phi(j)}}]E[\underline{L_j}]. \end{aligned} \tag{3.12}$$

The last step is the only time we have used the independence of $\overline{L_j}$ and $\underline{L_j}$. Once again, the expectations on the right-hand side are either given or part of $E[G_{\phi(j)}^T G_{\phi(j)}]$. Since the skeleton imposes a partial order on the joints, it is always possible to compute $E[G_{\phi(j)}^T G_{\phi(j)}]$ in advance.

Fixed translations

Up to now, our treatment has considered bones simply as coordinate frames, albeit with connections to each other. In practice, these connections are usually rigid bones joined at sockets, that constrain these coordinates frames to exist at fixed offsets relative to their parents. In this model, all translation components ($\underline{L_j}$) are fixed. This trivially makes them independent of each other and of the linear components. We can thus apply the results in the previous section. Since $\underline{L_j}$ will now factor out of expectations, we need only have the fixed value of $\underline{L_j}$ and the expectations $E[\overline{L_j}]$ and $E[\overline{G_j}^T\overline{G_j}]$.

Rotation and fixed translation only

Depending on the required sophistication of an animation, it may be sufficient to use only rotation matrices for the linear components ($\overline{L_j}$). This accounts for joint movements, but not effects like muscle bulging (which do not strictly fit into a "skeletal" animation model, but are often approximated with one). Provided that the ancestor frames never produce non-uniform scaling (i.e., stretch or shrink along some axis), then $E[\overline{G_j}^T \overline{G_j}]$ can be computed. Since $\overline{G_{\phi(j)}}$ represents only rotations and uniform scaling, $\overline{G_{\phi(j)}}^T \overline{G_{\phi(j)}}$ is of the form λI, where λ is a random variable. It follows that

$$\begin{aligned} E[\overline{G_j}^T \overline{G_j}] &= E[\overline{L_j}^T \overline{G_{\phi(j)}}^T \overline{G_{\phi(j)}}\, \overline{L_j}] \\ &= E[\lambda \overline{L_j}^T \overline{L_j}] \\ &= E[\lambda I] \\ &= E[\overline{G_{\phi(j)}}^T \overline{G_{\phi(j)}}]. \end{aligned} \tag{3.13}$$

We thus need only $\underline{L_j}$ and $E[\overline{L_j}]$. The former is simply the length and rest orientation of the bone, and the latter can be computed given information about the range of motion of each joint: either by sampling the range, or analytically from a distribution function. As an example, consider the case of a joint that can only rotate about the z axis by angles between ϕ_0 and ϕ_1, with a uniform distribution. Then

$$\begin{aligned} E[\overline{L_j}] &= \frac{1}{\phi_1 - \phi_0} \int_{\phi_0}^{\phi_1} \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} d\phi \\ &= \frac{1}{\phi_1 - \phi_0} \begin{pmatrix} \sin\phi_1 - \sin\phi_0 & \cos\phi_1 - \cos\phi_0 & 0 \\ \cos\phi_0 - \cos\phi_1 & \sin\phi_1 - \sin\phi_0 & 0 \\ 0 & 0 & \phi_1 - \phi_0 \end{pmatrix}. \end{aligned} \tag{3.14}$$

This can be generalised to arbitrary axes of rotation and to multiple degrees of freedom (with Euler angles), although the formulae become more cumbersome.

Uniform distribution of rotations, fixed translations

If a linear part $\overline{L_j}$ is equally likely to be any rotation matrix (as defined by the Haar measure [Shoemake, 1992]), then $E[\overline{L_j}] = 0$. This is quite an unlikely assumption (since it implies total freedom for all joints), but it does allow the norm to be computed in the absence of almost all information about the model: only $\underline{L_j}$ is required, which corresponds to the length of the bone. This information is required anyway in order to define the rest pose, so no further work is needed to obtain it.

In this case, we can also compute $E[\overline{G_j}^T \overline{G_j}]$ without the assumption that the ancestors perform

only uniform scaling (or no scaling). In Appendix B.2 we show that

$$E[\overline{G_j}^T \overline{G_j}] = \frac{\operatorname{tr} E[\overline{G_{\phi(j)}}^T \overline{G_{\phi(j)}}]}{3} \cdot I. \tag{3.15}$$

Fixed skeleton

The simplest possible model takes the rest pose and assumes that no animation occurs. The main use for this model is that it lets us measure the Euclidean norm on the rest pose within the same framework. In this model, $E[L_j] = L_j$ and $E[G_j^T G_j] = L_j^T E[G_{\phi(j)}^T G_{\phi(j)}] L_j$.

3.3.2 Inner products

In the previous section we showed how an average value for squared distance can be found. This can be extended to finding the average value for the dot product. We use this to establish the concept of orthogonality in animation space, but other applications will be developed in subsequent chapters.

We define $\langle \mathbf{s}, \mathbf{t} \rangle_{2,2}$ to be the expected dot product of model-space vectors $G\mathbf{s}$ and $G\mathbf{t}$:

$$\begin{aligned} \langle \mathbf{s}, \mathbf{t} \rangle_{2,2} &= E[G\mathbf{s} \cdot G\mathbf{t}] \\ &= E[\mathbf{s}^T G^T G \mathbf{t}] \\ &= \mathbf{s}^T E[G^T G] \mathbf{t} \\ &= \mathbf{s}^T P \mathbf{t}. \end{aligned} \tag{3.16}$$

Since $\langle \mathbf{s}, \mathbf{s} \rangle_{2,2} = \|\mathbf{s}\|_{2,2}^2$ is the expectation of a squared length, it cannot be negative. The other properties of inner products are easily verified (since $G^T G$ is symmetric for all G), with the exception of $\langle \mathbf{s}, \mathbf{s} \rangle_{2,2} = 0 \implies \mathbf{s} = \mathbf{0}$. This will fail for any vector $\mathbf{s}$ in the null space of P. Such a vector always exists in the *rotations and fixed translations* statistical model, because the skeleton has only three degrees of freedom per joint while animation space has four dimensions per joint (for a more rigorous proof, refer to Appendix B.3). In spite of this shortcoming of the $L_{2,2}$ norm and the inner product, we will continue to refer to them as such. Formally, they can be considered to be a norm and an inner product on the quotient space $\mathcal{M}/\mathcal{N}$, where

$$\mathcal{M} = \{\mathbf{x} \in \mathbb{R}^{4b} : \underline{\mathbf{x}} = 0\}$$

and $\mathcal{N}$ is the null space of P; in this space they satisfy all the requirements of a norm and an inner product.

In practice, the issue of rank deficiency in P only becomes important when optimising locations in animation space (for example, this makes the $L_{2,2}$ norm unsuitable for the edge constraints described in Section 4.1.2). We do not project onto $\mathcal{N}^\perp$, as that would destroy sparsity in animation-space coordinates.

We can define a linear transformation of animation space which maps the $L_{2,2}$ norm and inner product to the usual Euclidean norm and dot product. The matrix P is non-negative definite symmetric, so there exists a matrix C such that $C^TC = P$; we have computed $C = P^{\frac{1}{2}}$ from the singular value decomposition of P. If we let $\tilde{\mathbf{s}} = C\mathbf{s}$ and $\tilde{\mathbf{t}} = C\mathbf{t}$, then

$$\tilde{\mathbf{s}} \cdot \tilde{\mathbf{t}} = \mathbf{s}^T C^T C \mathbf{t} = \langle \mathbf{s}, \mathbf{t} \rangle_{2,2}. \tag{3.17}$$

We refer to this transformed space as *normalised animation space*. In general we try to avoid using it in implementations (because the transformation destroys sparsity), but it is a useful tool for analysis.

3.3.3 $L_{2,\infty}$ norm

The metrics derived above are useful when a good estimate of the distance is required, but cannot give hard bounds. Such bounds are sometimes required, for example with visibility culling, where an under-estimate of distance could lead to a visible object being culled. In this section we will derive an $L_{2,\infty}$ norm, such that

$$\|G\mathbf{s}\| \leq \|\mathbf{s}\|_{2,\infty} \text{ for all } G. \tag{3.18}$$

As usual with an L_∞ norm, it will be necessary to work with bounding volumes. Since a bounding "volume" is essentially just a set, we will use the term for sets of matrices and scalars, as well as sets of vectors. For an arbitrary set $\mathcal{S}$, we will use the notation $B[\mathcal{S}]$ to mean a bounding volume containing $\mathcal{S}$. Of course, we would like it to be as small a bounding volume as possible, but we do not absolutely require this as it may be too expensive to compute. We will also be performing algebraic operations on sets: the result is a set consisting of applying the operation to all possible combinations of elements from the input sets. For example, $\mathcal{A} + \mathcal{B}$ is the Minkowski sum $\{\mathbf{a} + \mathbf{b} : \mathbf{a} \in \mathcal{A} \text{ and } \mathbf{b} \in \mathcal{B}\}$, while for a matrix N, $N\mathcal{V} = \{N\mathbf{v} : \mathbf{v} \in \mathcal{V}\}$.

At a conceptual level, we can simply say that

$$\|\mathbf{s}\|_{2,\infty} = \max\{\|\mathbf{v}\| : \mathbf{v} \in B[G\mathbf{s}]\}, \tag{3.19}$$

As before, $\mathbf{s}$ must be a true vector (i.e., $\underline{\mathbf{s}} = 0$). Like the expectation operator $E[\cdot]$, $B[\cdot]$ operates on an expression and produces a bounding volume over all possible values of the expression (in this case, over all possible values of G).

Let us consider a particular G, and expand $G\mathbf{s}$ to $\sum_{j=0}^{b-1} G_j\mathbf{s}_j$. The geometric interpretation is that each $\mathbf{s}_j$ contains some part of the whole sum, but in a different coordinate frame (the various bone frames). Instead of directly converting each $\mathbf{s}_j$ to the root frame, we can propagate and combine the values up the tree. Referring to Figure 3.2 (page 21), we can propagate $\mathbf{s}_6$ up to frame 3 to obtain $L_6\mathbf{s}_6$, then combine it with $\mathbf{s}_3$ to obtain a value of $\mathbf{s}_3 + L_6\mathbf{s}_6$ in frame 3. When this value and $\mathbf{s}_4$ are propagated to frame 1 and combined with $\mathbf{s}_1$, we obtain a subtotal of $\mathbf{s}_1 + L_4\mathbf{s}_4 + L_3(\mathbf{s}_3 + L_6\mathbf{s}_6)$ in frame 1. Finally, all the contributions from the $\mathbf{s}_j$'s will propagate up to frame 0, and we will

have the value of $G\mathbf{s}$.

To compute $B[G\mathbf{s}]$, we can no longer consider just a single value of G and propagate vectors. Instead, we assume a bounding volume for each L_j, and propagate bounding volumes up the tree. Before any propagation occurs, we have the bounding volume $\{\mathbf{s}_j\}$ (a singleton set) in each frame j. If at some point in the algorithm, frame j has a bounding volume $\mathcal{V}_j$ and its parent $\phi(j)$ has a bounding volume $\mathcal{V}_{\phi(j)}$, then propagation makes the replacement

$$\mathcal{V}_{\phi(j)} \longleftarrow B\big[\mathcal{V}_{\phi(j)} + B[L_j]\mathcal{V}_j\big]. \tag{3.20}$$

This is the equivalent of the update $\mathbf{v}_{\phi(j)} \longleftarrow \mathbf{v}_{\phi(j)} + L_j\mathbf{v}_j$ we used for a fixed G in the example above. We replace L_j with $B[L_j]$ since we have to consider that L_j could take on any legal value; the outer $B[\cdot]$ is needed because its argument may not be suitably shaped to represent directly.

So far we have been careful to develop the mathematics without reference to the types of bounding volumes used, in order to produce a general model. We now consider the best choices for bounding volumes. Spheres make a good choice, since they are closed under translation, rotation and Minkowski summation. Translations are 3-vectors, so we bound them with straightforward spheres, denoted by

$$\mathcal{S}(\mathbf{v}, r) := \{\mathbf{u} : \mathbf{u} \in \mathbb{R}^3, \|\mathbf{u} - \mathbf{v}\| \leq r\}. \tag{3.21}$$

The positions we deal with occupy a 4D homogeneous space. We again limit the 3D part to a sphere, but it will be seen below that there is enough information to limit w to a fixed value. We thus use volumes of the form

$$\mathcal{S}_w(\mathbf{v}, r) := \{\mathbf{u} : \mathbf{u} \in \mathbb{R}^4, \|\overline{\mathbf{u}} - \mathbf{v}\| \leq r \text{ and } \underline{\mathbf{u}} = w\} \tag{3.22}$$

(where $\mathbf{v}$ is a 3-vector).

Finally, we must choose a bounding volume for matrices. We separate a matrix into its translation part (already covered) and linear part. We choose to limit the largest singular value[2]; for matrices that are known to only rotate and not scale, this is trivially 1. We use the notation

$$\mathcal{M}(s) := \{A : A \in \mathbb{R}^{3\times 3}, \|A\|_2 \leq s\}. \tag{3.23}$$

We denote the set of matrices whose linear part lies in $\mathcal{M}(s)$ and whose translation part lies in $\mathcal{S}(\mathbf{v}, r)$ by $\mathcal{M}(s, \mathbf{v}, r)$. The key equation (illustrated in Figure 3.3) that allows us to apply (3.20) is

$$\mathcal{M}(s)\mathcal{S}(\mathbf{v}, r) = \mathcal{S}\big(\mathbf{0}, s\|\mathbf{v}\| + sr\big). \tag{3.24}$$

Now suppose we have a volume $\mathcal{V} = \mathcal{S}_w(\mathbf{v}, r)$ in some frame j, and wish to transform it to the parent frame $\phi(j)$. The joint matrix will itself have a bounding volume $B[L_j]$ which must be

[2]The largest singular value of a matrix is the largest amount by which it can scale a vector, and is also known as the 2-norm.

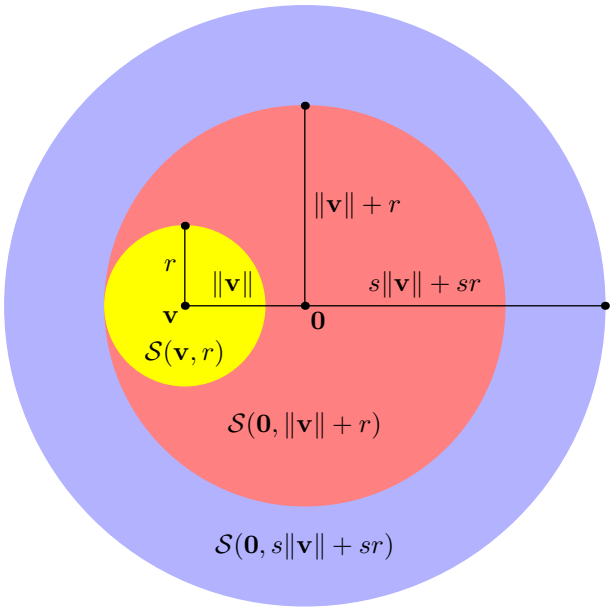


Figure 3.3: Visualisation of Equation (3.24). The yellow sphere is the original volume. The red sphere contains any rotation of the yellow sphere around the origin, while the blue sphere contains any scaling of the red sphere by up to s.

taken into account; let $B[L_j] = \mathcal{M}(s, \mathbf{v}_1, r_1)$. We compute the 3D and weight components of the resulting volume separately:

$$\begin{aligned} \overline{B[L_j]\mathcal{V}} &= \overline{B[L_j]}\,\overline{\mathcal{V}} + \underline{B[L_j]}\,\underline{\mathcal{V}} \\ &= \mathcal{M}(s)\mathcal{S}(\mathbf{v}, r) + w\mathcal{S}(\mathbf{v}_1, r_1) \\ &= \mathcal{S}(\mathbf{0}, s\|\mathbf{v}\| + sr) + \mathcal{S}(w\mathbf{v}_1, wr_1) \\ &= \mathcal{S}(w\mathbf{v}_1, s\|\mathbf{v}\| + sr + wr_1) \end{aligned} \tag{3.25}$$

$$\underline{B[L_j]\mathcal{V}} = \underline{\mathcal{V}} = w. \tag{3.26}$$

Therefore

$$B[L_j]\mathcal{V} = \mathcal{S}_w(w\mathbf{v}_1; s\|\mathbf{v}\| + sr + wr_1). \tag{3.27}$$

Equation (3.20) also requires the addition of two bounding volumes; but this is easy, since

$$\mathcal{S}_{w_0}(\mathbf{v}_0, r_0) + \mathcal{S}_{w_1}(\mathbf{v}_1, r_1) = \mathcal{S}_{w_0+w_1}(\mathbf{v}_0 + \mathbf{v}_1, r_0 + r_1). \tag{3.28}$$

Since we have equality in all the above equations, we can apply the propagation process repeatedly to obtain a tight bounding volume in the root space. In other words, there will exist some pose for which the $L_{2,\infty}$ norm is attained, although this pose might not be physically possible, because

we have not restricted the range of rotations.

3.4 Transforming a model

For some applications, it is useful to be able to transform an entire model within the model space, rather than just by modifying the model-to-world transformation. For example, it may be useful to transform a collection of models so that they all have the same scale, orientation and position relative to the model origin. This allows them to be combined in a scene without having to maintain extra matrices to encode these normalisations.

For this purpose, we consider how an animation-space model can be transformed by any transformation consisting of translations, rotations and uniform scaling. In particular, we show how the bounds and expectations can be transformed without requiring knowledge of the probability density functions originally used to compute them. Throughout the following sections, a prime is used to denote the updated value of a variable, e.g., G'_j is the new value of G_j.

3.4.1 Scaling a model

We first consider how a model can be scaled by a uniform scale factor s about the origin. Introducing scaling into the matrices is undesirable, since it limits the choice of file formats and rendering systems (e.g., Doom III's model format represents transformations compactly using quaternions [Henry, 2005]). Instead, we modify the coordinates of the vertices to achieve the same effect. Specifically, we set $\overline{\mathbf{p}_j}' = s\overline{\mathbf{p}_j}$ and $\underline{G_j}' = s\underline{G_j}$, and leave the remaining parts unchanged. Expanding the animation space equation gives

$$\begin{aligned}\overline{\mathbf{v}}' = \overline{G'\mathbf{p}'} &= \sum \overline{G'_j \mathbf{p}'_j} = \sum \left(\overline{G'_j}\,\overline{\mathbf{p}'_j} + \underline{G'_j}\,\underline{\mathbf{p}'_j}\right) \\ &= \sum \left(s\overline{G_j}\,\overline{\mathbf{p}_j} + s\underline{G_j}\,\underline{\mathbf{p}_j}\right) \\ &= s\overline{\mathbf{v}}.\end{aligned} \tag{3.29}$$

Note that since we do not modify $\underline{\mathbf{p}_j}$, we have $\underline{\mathbf{v}'} = \underline{\mathbf{v}}$.

Now observe that

$$\begin{aligned}s\underline{G_{\phi(j)}} + s\overline{G_{\phi(j)}}\underline{L_j} &= s\underline{G_j} \\ &= \underline{G'_j} \\ &= \underline{G'_{\phi(j)}} + \overline{G'_{\phi(j)}}\underline{L'_j} \\ &= s\underline{G_{\phi(j)}} + \overline{G_{\phi(j)}}\underline{L'_j}.\end{aligned} \tag{3.30}$$

In order to satisfy this we must have $\underline{L'_j} = s\underline{L_j}$; that is, the translation part is scaled by s. The linear parts of the G_j's do not change, so $\overline{L'_j} = \overline{L_j}$.

Finally, we must adjust the expectations and bounds. Since expectation is a linear map, $E[\overline{L'_j}] = E[\overline{L_j}]$ and $E[\underline{L'_j}] = sE[\underline{L_j}]$. Now let $E[G_i^T G_j] = \left(\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right)$, where A is 3×3, B is 3×1, C is 1×3

and D is a scalar. Then

$$E[G_i'^T G_j'] = E\left[\begin{pmatrix} \overline{G_i'}^T \overline{G_j'} & \overline{G_i'}^T \underline{G_j'} \\ \underline{G_i'}^T \overline{G_j'} & \underline{G_i'}^T \underline{G_j'} + 1 \end{pmatrix}\right] = \begin{pmatrix} A & sB \\ sC & s^2(D-1)+1 \end{pmatrix}. \tag{3.31}$$

The bounds are simple to modify. There are no changes to the linear parts, so the largest singular values are unchanged. The translation parts of each bone are scaled by s, and so the bounding spheres for the translations must be similarly scaled.

3.4.2 Rotating and translating a model

Next, we consider how to rotate and translate a model by a matrix M. Note that any transformation in which scaling is uniform can be decomposed into a pure scale followed by a rotate-and-translate. Since we wish to make the change in model-space, we cannot apply any change to the root. However, we can apply M to the bones immediately below the root, setting $L_j' = ML_j$ for all bones j that are children of the root. This has the effect of making $G_j' = MG_j$ for every non-root bone. We thus set $\mathbf{p}_0' = M\mathbf{p}_0$ to account for the transformations on vertices that are influenced by the root, giving

$$\mathbf{v}' = G_0'\mathbf{p}_0' + \sum_{j=1}^{b-1} G_j'\mathbf{p}_j' = M\mathbf{p}_0 + \sum_{j=1}^{b-1} MG_j\mathbf{p}_j = M\left[G_0\mathbf{p}_0 + \sum_{j=1}^{b-1} G_j\mathbf{p}_j\right] = MG\mathbf{p} = M\mathbf{v}. \tag{3.32}$$

Updating $E[L_j]$ and the bounds is simple: $E[L_j'] = ME[L_j]$ for sub-root bones, the 2-norms of $\overline{L_j}$ do not change, and the translation bounding sphere must be transformed by M for sub-root bones. Updating $E[G_i^T G_j]$ is less trivial. Let $M^T M = \left(\begin{smallmatrix} I & B \\ B^T & C \end{smallmatrix}\right)$ where B is a 3×1 matrix and C is a scalar; the top-left portion is the identity because M is a rigid transformation. Then for $i, j > 0$,

$$\begin{aligned} G_i'^T G_j' &= G_i^T M^T M G_j \\ &= \begin{pmatrix} \overline{G_i}^T & \mathbf{0} \\ \underline{G_i}^T & 1 \end{pmatrix} \begin{pmatrix} I & B \\ B^T & C \end{pmatrix} \begin{pmatrix} \overline{G_j} & \underline{G_j} \\ \mathbf{0}^T & 1 \end{pmatrix} \\ &= \begin{pmatrix} \overline{G_i}^T & \mathbf{0} \\ \underline{G_i}^T & 1 \end{pmatrix} \begin{pmatrix} \overline{G_j} & \underline{G_j} + B \\ B^T\overline{G_j} & B^T\underline{G_j} + C \end{pmatrix} \\ &= \begin{pmatrix} \overline{G_i}^T \overline{G_j} & \overline{G_i}^T \underline{G_j} + \overline{G_i}^T B \\ \underline{G_i}^T \overline{G_j} + B^T \overline{G_j} & \underline{G_i}^T \underline{G_j} + \underline{G_i}^T B + B^T \underline{G_j} + C \end{pmatrix} \\ &= G_i^T G_j + \begin{pmatrix} 0 & \overline{G_i}^T B \\ B^T\overline{G_j} & \underline{G_i}^T B + B^T \underline{G_j} + C - 1 \end{pmatrix}. \end{aligned} \tag{3.33}$$

If instead $i = 0, j > 0$ then $G_i'^T G_j' = MG_j$, and similarly for $i > 0, j = 0$.

In order to evaluate Equation (3.33), we need the expected values of G_j for each j. But $G_j = G_0^T G_j$, so $E[G_j] = E[G_0^T G_j]$, which we already have for the original model.

3.5 Transformation of normals

Our formulation of character animation poses a unique challenge in correctly computing normals. For a static model, it is well known that normals should be transformed by the inverse transpose of the matrix used for transforming vertices [Turkowski, 1990]. That is, if $\mathbf{v} = M\hat{\mathbf{v}}$ then

$$\mathbf{n} = \overline{M}^{-T}\hat{\mathbf{n}}. \tag{3.34}$$

For SSD a similar solution is often employed: the SSD equation is regrouped as $\mathbf{v} = (\sum w_j G_j \hat{G_j}^{-1})\hat{\mathbf{v}}$ and so the transformed normal is computed as

$$\mathbf{n} = \overline{\left(\sum w_j G_j \hat{G_j}^{-1}\right)}^{-1}\hat{\mathbf{n}}. \tag{3.35}$$

This equation is only correct in regions where the weights are constant, although it is possible to correct it using terms involving derivatives of the weight fields [Merry et al., 2006].

SSD defines skinning as a deformation of a 3D surface (the rest pose), so the idea of a normal is well-defined. Animation space, however, defines skinning as the projection of a surface in a higher-dimensional space, where a normal vector is not well-defined. We could choose one of the many vectors for which the $L_{2,2}$ inner product (Section 3.3.2) with tangent vectors is zero; however, this inner product gives only the *expected* dot product in 3D, so this vector would not be guaranteed to be normal to the surface in all poses, and furthermore, it would be difficult to control whether it pointed into or out of the surface.

A better approach is to note that the skin is a 2D surface in animation space, and hence has a tangent plane at every point at which it is smooth. The linear animation projection will transform this plane into a plane in 3D, which will be tangent to the projected surface. The desired normal is thus perpendicular to this plane.

We implement this approach by storing with each sample point (vertex for per-vertex lighting, or texture sample for per-pixel lighting) a pair of animation-space vectors $\mathbf{s}$ and $\mathbf{t}$, which are tangent to the surface at that point. For numerical stability, we choose them to be orthonormal with respect to the $L_{2,2}$ inner product. During animation, we apply animation projection to both vectors, and take the normalised cross product of the results as the normal. Note that since $\{\mathbf{s}, \mathbf{t}\}$ spans the animation-space tangent plane, $\{G\mathbf{s}, G\mathbf{t}\}$ will span the 3D tangent plane. It follows that the cross product will be non-zero, and the normal well-defined, except where animation projection collapses the tangent plane to a line or a point. This can only happen in pathological situations which do not correspond well with the real world, such as surface self-intersection.

Computing good animation-space tangent planes is surprisingly challenging. We defer a discussion of the approaches we have used until Section 4.3.

3.6 Summary

- We have defined animation space as a $4b$-dimensional homogeneous space, together with a pose-dependent linear projection that maps it to 4D homogeneous space. Animation-space vectors are the difference between two points in animation space, and the projection operator maps them to 3D vectors.
- Animation space is a generalisation of skeletal subspace deformation (SSD) and a specialisation of multi-weight enveloping (MWE).
- We have developed an $L_{2,2}$ norm which measures the root-mean-squared length of an animation-space vector after projection (across all poses), together with a compatible inner product, which depend on both the structure of the skeleton and the probability distribution of the joint configurations. However, the inner product space thus defined generally has fewer than $4b - 1$ true dimensions, due to the presence of null vectors in animation space that have no effect on skinning.
- We have also developed an $L_{2,\infty}$ norm which provides a guaranteed bound on distance in any pose. This norm also depends on the skeletal structure and range of motion, but not on the probability distribution.
- We have shown that any non-shearing global transformation (i.e., a rigid transformation together with uniform scaling) may be absorbed into the model, and it is not necessary to store the original probability distributions in order to update the metrics.
- We have described (at a high level) how to compute normals. While our method is expensive (requiring two animation projections), it is also accurate, unlike the majority of methods which make the hidden and incorrect assumption that the transformation matrix will be locally constant.

Chapter 4

Model creation

Chapter 3 covered the underlying theory of animation space, in which we showed that it has two primary advantages:

(a) It provides a unifying framework for analysing the large body of existing SSD-based models and tools.

(b) It is more general than SSD, allowing for models that are more accurate and realistic. There are many models that cannot be represented with SSD that can be represented with animation space.

The first aspect has immediate benefit, and in subsequent chapters we consider some applications of the framework. However, in order to realise the full potential of animation space, we must consider how new models may be created without the restrictions of SSD. As is generally the case with a more general framework, the extra degrees of freedom come at the price of ease of use. The following sections examine some methods for creating an animation-space model that are practical to use.

In Section 4.1, we consider the problem of generating an optimal animation-space model, given a set of example poses that we wish to match. Section 4.2 considers a less powerful but also less onerous method: a coarse SSD model is refined by subdivision in animation space. Finally, in Section 4.3 we describe several methods for computing tangent planes to the discrete mesh.

4.1 Fitting models to a database of examples

SSD models tend to be modelled by the user in a modelling package. The rest pose of the model is built, then weights are "painted" onto the surface. This is a tedious and frustrating process, as setting weights only indirectly controls vertex positions. Mohr et al. [2003] have developed an authoring tool that allows vertices to be directly manipulated in posed positions, but the modelling is still all done by hand. In animation space, it is not practical to directly assign a $\mathbf{p}$ vector to

each vertex, because there are too many degrees of freedom, and because the intuitive relationship between SSD weights and "amount of influence" is lost. Thus, some other modelling technique is necessary to take full advantage of the extra degrees of freedom that are available.

In the last few years there has been interest in fitting a model to a set of example poses. Example poses can come from a variety of sources, such as laser range scans, high-end animation (e.g., a physically-based model), or an artist. Wang and Phillips [2002] use a modified linear solver to fit their multi-weighted enveloping model. Mohr and Gleicher [2003b] use a bilinear solver to fit an SSD model, while James and Twigg [2005] take the rest positions as given and solve a non-negative least-squares problem to compute the weights.

All of these fitting schemes have two initial requirements: the example meshes must be in correspondence (i.e., each vertex in one mesh is associated with a particular vertex in the other meshes), and a skeleton, or at least a set of bone coordinate frames, must be known along with its associated pose for each of the examples. Establishing correspondence is a separate problem that we do not address; refer to Section 2.4.1 for existing methods.

The linearity of animation space makes fitting a relatively simple least-squares problem. Consider a single vertex, and a set of examples with poses $G^1, G^2, \ldots, G^n$ and corresponding vertex positions $\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^n$ (we use superscripts to distinguish G^1, an example of G, from G_1, a component of G). If the vertex has position $\mathbf{p}$ in animation space, then we aim to satisfy

$$\begin{pmatrix} G^1 \\ G^2 \\ \vdots \\ G^n \end{pmatrix} \mathbf{p} = \begin{pmatrix} \mathbf{v}^1 \\ \mathbf{v}^2 \\ \vdots \\ \mathbf{v}^n \end{pmatrix}. \tag{4.1}$$

In general this may be an over-constrained problem, so instead we minimise the objective function

$$E(\mathbf{p}) = \sum_{k=1}^{n} \left\| G^k \mathbf{p} - \mathbf{v}^k \right\|_2^2. \tag{4.2}$$

In practice, however, this will lead to over-fitting (the resulting model will fit well to the example poses, but will not generalise to new poses). Furthermore, it is likely that every bone will influence every vertex, which will make it unusable for real-time rendering. In the following subsections we address these issues.

4.1.1 Influence sets

Over-fitting tends to create spurious influences, where moving a bone in one part of the body has an effect in a completely separate part of the body, simply because this produces a better fit. This is usually avoided by limiting the bones influencing a vertex to a pre-defined set. Wang and Phillips [2002] require the user to paint an *influence map* onto the surface to establish where influences may exist for each bone; Mohr and Gleicher [2003b], Kry et al. [2002] and James and Twigg [2005]

automatically determine influences by looking for correlations between joint movement and vertex movement in the examples. Either of these approaches can be used for animation space; we have used manually designated influence sets in our implementation.

4.1.2 Regularisation

Regularisation is a common method for preventing over-fitting [Wang and Phillips, 2002]. Equation (4.2) is modified to take the form:

$$E'(\mathbf{p}) = \sum_{k=1}^{n} \left\| G^k \mathbf{p} - \mathbf{v}^k \right\|_2^2 + \lambda \left\| \mathbf{p} \right\|_2^2 . \tag{4.3}$$

The regularisation parameter λ is usually chosen to be much smaller than the other coefficients, so that it has little effect when $\mathbf{p}$ is properly determined, but prevents $\mathbf{p}$ from becoming unnaturally large when it is under-determined.

The use of influence sets introduces a characteristic artefact that is not fixed by this form of regularisation. Vertices at the boundary of an influence set may receive a large influence from the associated bone in order to obtain a good fit, while neighbouring vertices outside the influence set receive no influence at all. When the model is placed into new poses, this often manifests as a discontinuity, such as a shearing effect in a limb (see the neck in Figure 8.3(a) for an example of this shearing). Wang and Phillips [2002] address this by adjusting the relative weights of the fitting and regularisation terms to penalise large influences near the boundary (this requires a smooth user-specified influence map).

We address discontinuities by penalising large jumps between neighbouring vertices. Unfortunately, this means that we can no longer solve for each vertex individually, but must solve for them simultaneously. The objective function becomes

$$E''(\mathbf{p}_1, \ldots, \mathbf{p}_V) = \sum_{i=1}^{V} \sum_{k=1}^{n} \left\| G^k \mathbf{p}_i - \mathbf{v}_i^k \right\|_2^2 + \delta \sum_{j=1}^{E} \frac{1}{\ell_j} \left\| \mathbf{p}_{j,1} - \mathbf{p}_{j,2} \right\|_2^2 , \tag{4.4}$$

where i runs over the vertices and j runs over the edges of the model, and k runs over the examples, as before. $\mathbf{p}_{j,1}$ and $\mathbf{p}_{j,2}$ are the animation-space positions at the end-points of edge j, and ℓ_j is an estimate of the length of edge j (computed as the average length in the example meshes). The user specifies a value for δ, which weights the relative importance of continuity and closeness of fit.

Since we have used a Euclidean measure throughout, the function is not scale-invariant. In particular, w components in animation space are scale-invariant while the other components are not. To address this, we initially transform the models so that an axis-aligned bounding box around the reference model has a maximum side length of 1 and is centred at the origin. It is tempting to use the $L_{2,2}$ norm (which elegantly handles the issue of scale invariance), but, unfortunately, the directions which are under-determined are precisely those to which the $L_{2,2}$ norm assigns a

low weight, because they are unlikely to contribute to the animated position. As discussed before, positions in animation space may not be unique (see Appendix B.3) and so this is more than just a theoretical concern.

Optimising all vertices simultaneously rather than one at a time makes it essential to exploit sparsity in the matrix system. We have used LSQR [Paige and Saunders, 1982], a variant of conjugate gradients, as implemented by the Meschach library [Stewart and Leyk, 1994] to efficiently solve this large but sparse system. Meschach allows the matrix to be represented implicitly by providing callbacks to compute $A\mathbf{x}$ and $A^T\mathbf{x}$ given $\mathbf{x}$, and we have exploited this to reduce the memory required. The implementation uses $\mathrm{O}(Vn + I)$ memory to solve for V vertices, n example meshes and I variables; we show in the results that, in practice, the time and memory requirements are quite reasonable.

4.1.3 Homogeneous weight

The fitting process does not provide any guarantee that the resulting position $\mathbf{p}$ will satisfy $\underline{\mathbf{p}} = 1$; in fact, the regularisation terms work against this. To guarantee $\underline{\mathbf{p}} = 1$, we choose some bone k in the influence set and substitute

$$\underline{\mathbf{p}_k} = 1 - \sum_{j \neq k} \underline{\mathbf{p}_j}$$

into the linear equations before eliminating $\underline{\mathbf{p}_k}$. The value of $\underline{\mathbf{p}_k}$ is determined from this equation after solving for the rest of $\mathbf{p}$. Note that here $\mathbf{p}_j$ is the 4-element subvector corresponding to bone j in position $\mathbf{p}$, whereas in equation (4.4), $\mathbf{p}_i$ is the animation-space position of vertex i.

4.1.4 Skeleton fitting

So far we have assumed that the skeleton (i.e., G) is known for all the example poses, but this may not be the case. For ease of implementation, we have used a method that is partially user-guided to determine a skeleton and associated poses. There are also fully-automated techniques available; refer to Section 2.4.2.

In our system, the user creates a skeleton, and marks a set of core vertices associated with each bone. The user-drawn skeleton is used only to provide a hierarchy and bone-vertex associations — the geometry is almost entirely ignored. From here the approach is similar to that of James and Twigg [2005]. For each bone and each example, a least-squares optimisation is performed to find the affine matrix that best transforms the core vertices to fit the positions in this pose. Let the core vertices have homogeneous positions $\hat{\mathbf{v}}_1, \ldots, \hat{\mathbf{v}}_n$ in the reference pose, and $\mathbf{v}_1, \ldots, \mathbf{v}_n$ in the pose under consideration; and let M be the matrix for which we will solve. We

find the least-squares solution to the set of equations

$$\begin{pmatrix} \hat{\mathbf{v}}_1^T \\ \vdots \\ \hat{\mathbf{v}}_n^T \end{pmatrix} \begin{pmatrix} M_{11} & M_{21} & M_{31} \\ M_{12} & M_{22} & M_{32} \\ M_{13} & M_{23} & M_{33} \\ M_{14} & M_{24} & M_{34} \end{pmatrix} = \begin{pmatrix} \overline{\mathbf{v}_1}^T \\ \vdots \\ \overline{\mathbf{v}_n}^T \end{pmatrix}. \tag{4.5}$$

For the models we have dealt with, scaling and shearing effects appear to result from over-fitting rather than properties of the models, and we eliminate them. We can apply the *polar decomposition* [Golub and Van Loan, 1996] to write $\overline{M} = RS$, where R is orthogonal and S is symmetric. We replace $\overline{M}$ by R, which is the rotation part of the transformation [James and Twigg, 2005]. In some cases we find that R is a reflection rather than a rotation. Usually this occurs because the least-squares problem is ill-conditioned, for example, because all the core vertices lie close to a plane. In this case, we negate the smallest eigenvalue of S and recompute R. Having modified the linear part of M, the translation part may no longer be optimal. We perform another least-squares optimisation to update just the translation part of M.

We initially tried to preserve the bone lengths marked by the user and modified only the rotation; however, we found that it was difficult to manually determine the proper lengths, and incorrect lengths would over-constrain the problem and lead to poor fits.

This method solves for the matrix required to *move* a bone from the reference pose to the target pose, but it does not determine the animation projection matrix $\hat{G}$ for the rest pose. The rotations of the $\hat{G}_j$ matrices are irrelevant, but the translations determine the end-points of the bones. We initially used the user-provided skeleton to control this, but once again we found that the joints might be poorly placed and would lead to large translation components in the joint matrices. Apart from being anatomically infeasible, this would complicate calculation of the $L_{2,2}$ metric, as well as make the animator's task more difficult by making it necessary to control both rotations and translations.

To mitigate this problem, we compute a new matrix $\hat{G}$ that minimises the variance in the translation components of each joint matrix. Let M_j^p be the matrix that moves bone j from its reference position to its position in pose p (the matrix labelled M above). Since we are free to choose the linear part of $\hat{G}_j$, we set it to the identity. The free variables are thus only the translation components; let us denote them as $\mathbf{h}_j = \underline{\hat{G}_j}$. In pose p, we have $G_j^p = M_j^p \hat{G}_j$ and thus

$$\begin{aligned} \underline{L_j^p} &= \underline{(G_{\phi(j)}^p)^{-1} G_j^p} \\ &= \underline{(M_{\phi(j)}^p \hat{G}_{\phi(j)})^{-1} M_j^p \hat{G}_j} \\ &= \underline{\hat{G}_{\phi(j)}^{-1} [(M_{\phi(j)}^p)^{-1} M_j^p] \hat{G}_j} \\ &= \underline{\hat{G}_{\phi(j)}^{-1}} + \overline{\hat{G}_{\phi(j)}^{-1}} \underline{(M_{\phi(j)}^p)^{-1} M_j^p} + \overline{\hat{G}_{\phi(j)}^{-1} (M_{\phi(j)}^p)^{-1} M_j^p} \underline{\hat{G}_j} \\ &= -\mathbf{h}_{\phi(j)} + \underline{(M_{\phi(j)}^p)^{-1} M_j^p} + \overline{\underline{(M_{\phi(j)}^p)^{-1} M_j^p}} \mathbf{h}_j. \end{aligned} \tag{4.6}$$

We want to make the values of $\underline{L_j^p}$ as close to equal as possible, where j is held fixed and p ranges over the example poses. This is achieved by minimising

$$\sum_{p=1}^{n}(\underline{L_j^p} - \mathbf{m}_j)^2,$$

where $\mathbf{m}_j$ is a free variable that will equal the mean of the target set when the function above is minimised. We solve for each $\mathbf{h}_j$ in turn, working down the tree; for each j we solve the linear least-squares system with the free variables $\mathbf{h}_j$ and $\mathbf{m}_j$. Note that this gives the same result as a simultaneous optimisation over all the variables, because any change to $\mathbf{h}_{\phi(j)}$ is cancelled by a corresponding change to $\mathbf{m}_j$ without affecting $\mathbf{h}_j$.

Unfortunately, the solution may not always be well-determined. This is particularly true for joints that rotate around only one axis, in which case the centre of rotation may be placed anywhere along this axis. For this reason, we introduce an extra regularisation equation, $\lambda(\mathbf{h}_j - \mathbf{u}_j) = \mathbf{0}$, where λ is a regularisation weight (10^{-3} in all our results) and $\mathbf{u}_j$ is the original joint position specified by the user. This is the only place in which the geometry of the user-specified skeleton is used.

4.2 Subdivision surfaces

Subdivision surfaces are an extension of parametric surfaces to domains of arbitrary topology. They are widely used in character animation, having been popularised by DeRose et al. [1998]. There are many subdivision schemes (Catmull-Clark [Catmull and Clark, 1978], Loop [1987] and $\sqrt{3}$ [Kobbelt, 2000] being amongst the more popular), but they all have the same general approach: a smooth surface is defined by a coarse base mesh and a subdivision rule that defines how to refine the mesh. Each subdivision step splits the faces into multiple smaller faces, after which the positions are smoothed. The subdivision surface is the limit that is approached as this subdivision process is repeated over and over. We will not consider any particular subdivision scheme here, but rather the general properties of the subdivision process.

Depending on the application, subdivision may be done as a pre-process or on-the-fly. Shiue et al. [2005] have shown how a programmable GPU can be used to do hardware-accelerated subdivision on the fly, but the method is quite complex. On-the-fly subdivision with geometry shaders is relatively straightforward on fourth-generation programmable hardware such as the GeForce 8 series [Blythe, 2006], but at present this hardware has a very small installed base. An additional advantage of pre-processed subdivision is that subdivision then becomes a potential modelling tool: an artist may design a coarse approximation, apply subdivision to smooth edges, and then create the fine detail.

We now show that subdivision can be applied as a pre-process within animation space. This produces the same results as run-time subdivision of the animated 3D mesh, in contrast to SSD, where subdividing the rest-pose control mesh does not produce the same results as run-time

subdivision. It is possible to do this in animation space because subdivision schemes define the position of new vertices as affine combinations of the original ones. Let

$$V = \begin{pmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_n \end{pmatrix}$$

be the matrix containing all the positions in the control mesh, and let V' be the corresponding matrix after a subdivision step (for some arbitrary numbering of the vertices). Then there is a *subdivision matrix* S such that $V' = VS$ [Umlauf, 2005]. Critically, the matrix S depends only on the connectivity of the mesh, and not the positions. Now let

$$P = \begin{pmatrix} \mathbf{p}_1 & \cdots & \mathbf{p}_n \end{pmatrix} \tag{4.7}$$

be the equivalent matrix of animation space positions, and let $P' = PS$ — that is, the result of applying a subdivision step directly in animation space. If we apply animation projection to P' we get $GP' = GPS = VS = V'$, which is the same as applying subdivision to V in three dimensions.

4.3 Tangent planes

In Section 3.5 (page 31), we explained how we use tangent planes to model the normals of the animated model. In this section we will present several techniques to generate tangent planes.

The underlying difficulty is that there is no obvious method to take linear combinations of planes. Geometric algebra [Dorst and Mann, 2002] provides a well-defined linear combination, but in more than three dimensions the result is not necessarily a plane, but a more general element of the algebra. Another possibility is to take linear combinations of the basis elements, but this is unsatisfactory as the result depends on the choice of basis. In practice we have found that this leads to unsightly lighting artefacts, unless the bases are roughly aligned with each other. We use this for run-time interpolation across faces or between texture samples for efficiency (where we have taken care to ensure approximate alignment), but for off-line pre-processing we prefer basis-invariant techniques.

4.3.1 Fitting tangent planes to geometry

With static models, it is common practice to estimate the normal at a vertex by averaging the normals of the incident faces. We can extend this idea to estimate a tangent plane at a vertex from the planes of the incident faces. We use geometric algebra to define a distance metric between planes, and solve for the tangent plane that minimises the total distance to the given planes. The details of the algorithm may be found in Appendix A.1.

4.3.2 Fitting tangent planes to SSD normals

If our animated model was created using SSD, then it may have normals in the rest pose. Furthermore, these normals may contain more accurate information than can be determined from the geometry alone (they may, for example, control the apparent sharpness of a crease). It would be useful to create tangent planes that mimic the behaviour of the SSD normal, so that SSD models can be used with our system and produce the same results.

Fortunately, it is possible to choose a tangent plane that produces identical results to Equation (3.35) on page 31. Let us pick some vertex with rest-pose position $\hat{\mathbf{v}}$. SSD transforms $\hat{\mathbf{v}}$ by a 4×4 matrix. Referring to equation (2.2), this is the matrix $\sum_{j=0}^{b-1} w_j G_j \hat{G}_j^{-1}$, which we will refer to as M. Now let $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$ be orthogonal rest-pose tangents to the surface at $\hat{\mathbf{v}}$ (i.e., both tangents are orthogonal to the normal stored in the model). SSD transforms the normal by M^{-T}, which is equivalent to transforming the tangents by M and then taking their cross product[1]. Let $\mathbf{s} = M\hat{\mathbf{s}}$ and $\mathbf{t} = M\hat{\mathbf{t}}$ be these dynamic model-space tangents.

When we first introduced our animation scheme, we did it by separating M into its dynamic portion (G_j) and static portion, and merging the static portion with the vertex. We can apply the same method to the tangents, defining $\mathbf{q}$ in terms of $\hat{\mathbf{s}}$ as

$$\mathbf{q} = \begin{pmatrix} w_0 \hat{G}_0^{-1} \hat{\mathbf{s}} \\ \vdots \\ w_{b-1} \hat{G}_{b-1}^{-1} \hat{\mathbf{s}} \end{pmatrix}, \tag{4.8}$$

with $\mathbf{r}$ defined similarly in terms of $\hat{\mathbf{t}}$. Then

$$G\mathbf{q} = \sum_{j=0}^{b-1} G_j \mathbf{q}_j = \sum_{j=0}^{b-1} G_j w_j \hat{G}_j^{-1} \hat{\mathbf{s}} = \sum_{j=0}^{b-1} w_j G_j \hat{G}_j^{-1} \hat{\mathbf{s}} = \mathbf{s}, \tag{4.9}$$

and similarly $G\mathbf{r} = \mathbf{t}$. It follows that the tangent plane spanned by $\mathbf{q}$ and $\mathbf{r}$ has the desired properties.

Merry et al. [2006] have shown that for SSD, tangents should more accurately be transformed by the linear component of the matrix

$$N = \sum_{j=0}^{b-1} w_j G_j \hat{G}_j^{-1} + G_j \hat{G}_j^{-1} \hat{\mathbf{v}} \frac{\partial w_j}{\partial \hat{\mathbf{v}}}. \tag{4.10}$$

We can adapt Equation (4.8) by setting

$$\mathbf{q}_j = w_j \hat{G}_j^{-1} \hat{\mathbf{s}} + \hat{G}_j^{-1} \hat{\mathbf{v}} \frac{\partial w_j}{\partial \hat{\mathbf{v}}} \hat{\mathbf{s}}, \tag{4.11}$$

[1] The length is not the same, but only the direction of a normal is relevant.

so that

$$G\mathbf{q} = \sum_{j=0}^{b-1} G_j \mathbf{q}_j = \sum_{j=0}^{b-1} w_j G_j \hat{G}_j^{-1} \hat{\mathbf{s}} + G_j \hat{G}_j^{-1} \hat{\mathbf{v}} \frac{\partial w_j}{\partial \hat{\mathbf{v}}} \hat{\mathbf{s}} = N\hat{\mathbf{s}}. \tag{4.12}$$

4.4 Summary

While animation space provides a linear framework for analysing SSD models, this in itself does not overcome the flaws in SSD. We have demonstrated two practical methods for generating models that are more general than those possible with SSD: fitting to a set of examples, an approach that is already well-established with other animation frameworks; and subdivision of an existing SSD model, a common run-time method that animation space allows during modelling. We have also shown that although computing normals with animation space is relatively expensive, it is also more general and can emulate correct transformation of SSD normals [Merry et al., 2006] with no extra cost.

Chapter 5

Parametrisation

In this chapter, we will introduce the concept of a parametrisation, review some existing methods, and then describe our implementation. Specifically, we adapt an existing method to use the $L_{2,2}$ metric of animation space. This makes it possible to produce parametrisations that balance distortion across the probability distribution of poses.

5.1 Background

For many applications, it is convenient to have a 2D parametrisation of a mesh. A parametrisation is a function that maps the surface of a mesh to a subset of $\mathbb{R}^2$, and provides a convenient way to sample and store a function defined on the mesh. The most common application is texture-mapping: sampling mesh colour and storing it in an image, which can then be mapped back to the surface in hardware. More recently, parametrisations have been used to store other values such as position [Gu et al., 2002] and normals [Cohen et al., 1998]. Storing geometric properties in a two-dimensional domain simplifies various algorithms; for example, Briceño et al. [2003] use video-compression methods to compress an animation.

Since the topologies of most meshes are not directly embeddable in the plane (even a sphere cannot be embedded), parametrisation schemes divide the mesh into a number of charts. These charts are usually topological discs, and thus easily embedded in the plane. There has also been recent work in parametrising over domains with spherical topology (e.g., Praun and Hoppe [2003]), which can be done without cuts for genus-zero meshes. Mappings to a cube are particularly useful, as hardware-based cube-mapping is now widely available. A weakness of these techniques is their limited control over texture stretch. Tarini et al. [2004] address this with the polycube, a generalisation of cube mapping. However, polycube-maps are expensive to render (the authors use over 200 pixel shader instructions per pixel). On the whole, non-planar parametrisations are significantly more complex, and we restrict our survey below to plane mappings.

Some desirable properties of a parametrisation are:

Few charts. This is particularly important for level-of-detail schemes, which are unable to simplify features that cross chart boundaries [Sander et al., 2001].

Low stretch/length preservation. When a texture map is used to sample a function over a mesh, it is desirable to have a more or less uniform sampling rate in all directions and over all regions of the mesh (otherwise some regions will be either over- or under-sampled).

Conformality/angle preservation. For some applications, particularly texturing with a predefined pattern, such as wood, it is important that shear and anisotropic stretch are minimised, which is the case for an angle-preserving map.

One-to-one. Without a one-to-one mapping, several points on the mesh can map to the same point in the parametric domain, and so the inverse mapping becomes ambiguous.

In our implementation, the goal is to produce parametrisations that can be used to sample colour and normal data from a character model, to be used in conjunction with the level-of-detail method discussed in Chapter 6. This requires a one-to-one parametrisation, uniform stretch and few charts. Conformality is not a requirement, other than to the extent that non-conformal maps will have non-uniform stretch.

Parametrisation is normally separated into three steps: segmentation of the mesh into charts, flattening of those charts onto the plane, and packing of the flattened charts into a rectangular domain. Each step is discussed separately below. Global methods, such as those of Gu and Yau [2003] and Khodakovsky et al. [2003], avoid the seams introduced by segmentation, but can have very high stretch. Tapering elements (such as limbs) tend to be badly parametrised, making such methods inappropriate for character animation. Kharevych et al. [2006] address this by manually inserting conical singularities into the parametrisation domain. These singularities are later flattened by introducing cuts, but unlike the cuts used in other global schemes, these are determined only after the mapping is computed and hence have no effect on properties such as stretch and conformality.

5.1.1 Segmentation into charts

The first step is to cut the surface into disc-like regions, referred to as charts. The goal is to create roughly planar regions, so that the charts can be flattened without too much distortion. Although it is sufficient for a region to be developable[1] rather than planar, developable regions are difficult to identify and so the schemes discussed below search for planar regions.

Eck et al. [1995] grow their charts from a number of seed points: each face is assigned to the chart corresponding to the nearest seed. If this produces too few seeds, then more seeds are added and the process is restarted. Lévy et al. [2002] use a similar approach, but modified to place chart boundaries in areas of high curvature.

[1] A developable surface is one that has zero Gaussian curvature and hence can be flattened without stretching, e.g., a cylinder.

Sander et al. [2001] grow charts using a bottom-up merging approach. Initially, every face of the mesh is a chart, and charts are iteratively merged. Potential merges are ranked by compactness (measured by the length of the perimeter), and by planarity (measured by the distance to an approximating plane). The approach is based on earlier work by Garland et al. [2001]. This earlier work uses a discrete metric, summing the squared distance of each vertex from the plane, while Sander et al. [2001] integrate the squared distance of every point on the surface from the approximating plane. Since the surface is piecewise linear, this can be computed without resorting to numerical integration. The advantage of the continuous scheme is that it is independent of the resolution of the mesh, while the discrete scheme will over-weight densely triangulated regions.

Although used for a different application (shape approximation), Cohen-Steiner et al. [2004] use k-means clustering to identify a fixed number of charts. They employ both the mean-squared distance metric discussed above, as well as a metric that sums the squared error of the normals. The latter produces better results, as it penalises charts in which some triangles are "flipped" relative to the others.

5.1.2 Flattening charts

Somewhat confusingly, the process of flattening a single chart onto the plane, by assigning parametric coordinates to each vertex, is also known as "parametrisation"; in the interests of clarity, we will refer to it as "flattening" instead. There is a large body of literature on flattening, which is reviewed by Floater and Hormann [2004]. Rather than giving an exhaustive list of the algorithms proposed, we will identify several categories and reference a few examples in each.

A common property of all the algorithms we consider is that they depend only on intrinsic properties of the surface, i.e., those that would be apparent to a two-dimensional "inhabitant" of the surface. Since they are independent of the embedding, they can be adapted to animation space simply by replacing the Euclidean metric with our $L_{2,2}$ metric.

Linear methods

The majority of methods describe the parameters of each vertex in terms of a linear combination of its one-ring neighbours. The algorithms vary by the choice of linear weights and the handling of boundary conditions. Least squares conformal maps [Lévy et al., 2002] derives weights from an energy function that measures deviation from conformality. Desbrun et al. [2002] generalise this to a family of methods which includes a method which is locally area-preserving. Floater [2003] uses a slightly different set of weights based on the Mean Value Theorem for harmonic functions.

Angle-based methods

Some recent work uses non-linear methods that minimise energies that directly measure the change in angles. Angle-based flattening [Sheffer and de Sturler, 2000] solves for the angles in each flat-

tened triangle using constraints to ensure that the triangles do not flip and that it will be possible to consistently choose sizes for the triangles. Kharevych et al. [2006] instead solve for the angles between the circumcircles of triangles; the motivation is that this property is preserved by Möbius transformations, which are conformal.

Length-based methods

Some earlier work on parametrisation used spring energy [Eck et al., 1995] or multi-dimensional scaling [Zigelman et al., 2002] to minimise changes in edge length. This avoids the large range of scales that can typically be produced by conformal methods, but unfortunately the parametrisations are not guaranteed to be one-to-one.

Sander et al. [2001] consider uniformity directly, with their *texture stretch* metric. They define L_2 and L_∞ measures of texture stretch in terms of the Jacobian of the parametrisation. They also adjust the parametrisation once a progressive mesh has been created, to optimise both texture stretch and texture deviation (deviation of the texture between levels of detail). The optimisation is non-linear, so they use an iterative approach of moving one vertex at a time to a better position. Because the texture stretch of a flipped triangle is defined to be infinite, all the triangles in an optimal solution will be correctly oriented. However, if the initial estimate has too many flipped triangles, the optimisation may fail to converge.

Later work by Sander et al. [2002] defines a metric that is more appropriate when the function to be encoded is known, giving more parameter space to portions of the surface where the function changes more rapidly. In the case of a uniform function, the metric degenerates to the stretch metric discussed above. The non-linear optimisation is improved using a multi-resolution approach.

Boundary handling

Many earlier techniques require that the boundary of the patch be pre-specified, usually as a convex region. This can introduce areas of high distortion. Several later papers [Lévy et al., 2002, Sheffer and de Sturler, 2000, Sander et al., 2002] determine the boundary as part of the algorithm, usually requiring only two points to be fixed to determine position and orientation. Kharevych et al. [2006] go further by providing optional control of the exterior angle at each boundary vertex.

5.1.3 Packing charts into an atlas

The flattening step maps each surface chart into a polygon in 2D parameter space. Texture mapping hardware expects data in a regularly sampled rectangle, so it is necessary to place all the polygons into a rectangular region with minimum wasted space. This problem is known to be NP-hard [Sander et al., 2001]. Note that although the amount of available texture memory is

rapidly increasing, a tighter packing allows more detail to be represented with the same memory footprint, and may also have benefits for memory bandwidth and caching.

Sander et al. [2001] use a simple rectangle-packing heuristic. Each polygon is first placed inside a bounding rectangle, and the bounding rectangles are dropped in decreasing order of height, alternately left-to-right and right-to-left. In their later work [Sander et al., 2002], this is exploited in the parametrisation step by encouraging charts to grow into rectangular regions[2]. Lévy et al. [2002] improve upon this with their "tetris" heuristic: the texture space is filled bottom-up, with each polygon inserted so as to waste the minimum space between its bottom and the "skyline" of the already placed textures.

5.2 Segmentation

Our segmentation algorithm is similar to that of Sander et al. [2001] and Garland et al. [2001], but uses a metric more suitable for characters. These hierarchical schemes start by making every triangle into a separate chart, and adjacent charts are then merged to create larger charts. The choice of which charts to merge is driven by two factors:

1. topological constraints;
2. a cost metric.

In our case (as in that of Sander et al. [2001]), it is preferable that charts should be topologically equivalent (homeomorphic) to discs. We thus disallow any merges that create charts that are not discs. It is also possible to allow slightly more general charts if one only requires a continuous function mapping it to a disc; this corresponds to making a cut on the original mesh. For example, an uncapped cylinder could be made into a single chart, by cutting it down its length. We do not support these more general charts as they would have significantly complicated the implementation.

The cost metric determines the order of merges. The algorithm is greedy, always applying the legal merge with the lowest cost. In this it is very similar to edge collapses in a progressive mesh, which apply the lowest-cost legal collapse — in fact, Garland et al. [2001] demonstrate that a merge is very similar to an edge collapse on the dual graph.

5.2.1 Metric

Sander et al. [2001] use the following metric to determine the cost of a chart (the cost of a merge is defined as the cost of the chart it produces):

- For any plane, the mean-squared distance of the chart to the plane can be evaluated.
- The chart cost is the minimum such distance over all possible planes.

[2]While this reduces conformality, it improves sampling rate.

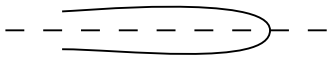


Figure 5.1: A sock. The plane indicated by the dashed line is close to the sock at every point, yet the sock is far from planar.

Figure 5.2: Our segmentation metric. (a) A planar chart, where the surface area is the same as the area of the boundary. (b) A non-planar chart, where the surface area is much greater than that of the boundary.

Their motivation is that a completely planar chart will have zero cost (because the best-fitting plane will be the plane through the chart), while a highly curved chart will have a large mean-squared distance from any plane. Garland et al. [2001] proposed essentially the same algorithm earlier; the difference is that Sander et al. evaluate mean-squared distance as an integral over the surface, while Garland et al. only evaluate it at the vertices.

Unfortunately, this metric (which we will refer to as the "mean-squared distance" metric) has a flaw that is particularly damaging when applied to character models. Consider a model with limbs, and consider a chart that has the shape of a sock, i.e., a cylinder capped at one end (see Figure 5.1). If the sock is long and narrow (as it would be for a limb), then it is reasonably well approximated by a plane containing its axis. In the extreme case of a completely flattened sock, it will be planar and have a cost of zero. However, it is impossible to flatten a sock without introducing significant stretch. The problem is exacerbated by systems that aim for compact charts by rewarding those with a short boundary.

There are two ways to address the "sock" problem: detection and prevention. Lévy et al. [2002] try to detect socks after segmentation, and insert a cut to improve the parametrisation. This raises new questions, such as the threshold at which to make the cut, and how the cut should be made. We take a preventative approach, by using an alternate metric that heavily penalises sock-shaped charts. Rather than adding an extra "sock-penalising" term to the existing metric (which would then require a weight to be tuned by the user), we use a metric that better measures how unfolding will stretch a mesh. It is equivalent to the $L^{2,1}$ metric of Cohen-Steiner et al. [2004], but we derive it in a different manner.

Consider first the special case of a chart with a planar boundary. The boundary can be considered to be a polygon in this plane, with a well-defined area. If the chart itself coincides with this polygon (Figure 5.2(a)), then the chart and the boundary polygon will have the same area. However, if the chart is far from planar (Figure 5.2(b)), then the chart's surface area will greatly exceed that

of the boundary. Thus, we can measure the planarity of the chart as

$$E_{\text{area}} = \frac{\text{surface area}}{\text{boundary area}}. \tag{5.1}$$

If the chart boundary is not planar, then the area of the "boundary polygon" ceases to be well-defined. Instead we define it as the largest possible area of the projection of the boundary onto any plane. In fact, the area of the projected boundary is just the area of the projected mesh itself (with flipped triangles having negative area). So, our generalised error metric is

$$E_{\text{area}} = \frac{\text{surface area}}{\text{maximum projected surface area}}. \tag{5.2}$$

In three dimensions, this is quite simple to calculate. For each triangle i, let $\mathbf{n}_i$ be the normal to the triangle, scaled by its area. The area is then trivially $\|\mathbf{n}_i\|$, while the area when projected onto a plane with unit normal $\mathbf{n}$ is $\mathbf{n}_i \cdot \mathbf{n}$. The projected area of the whole surface is simply

$$\sum \mathbf{n}_i \cdot \mathbf{n} = \left(\sum \mathbf{n}_i\right) \cdot \mathbf{n} \leq \left\|\sum \mathbf{n}_i\right\|,$$

and the maximum is achieved when $\mathbf{n}$ is parallel to $\sum \mathbf{n}_i$. Thus,

$$E_{\text{area}} = \frac{\sum \|\mathbf{n}_i\|}{\|\sum \mathbf{n}_i\|}. \tag{5.3}$$

It is also possible to perform the calculation in animation space, as described in Appendix A.1. Since sock-like regions tend to remain sock-like under animation, working in animation space does not usually improve the quality of the parametrisation, and in some cases can degrade it[3]. However, it causes chart boundaries to align better to the joints, which is beneficial for simplification as chart boundaries cannot be adjusted dynamically.

Sander et al. [2001] also aim to minimise a compactness term, which they define as

$$\frac{(\text{perimeter})^2}{4\pi(\text{area})},$$

and which has a value of 1 for a circle. This rewards charts that have roughly circular boundaries. Unfortunately, it also rewards sock-like charts, as they have a large area but a small boundary. A smooth boundary is beneficial for a number of reasons, so we use a similar compactness term. To avoid sock-like regions, we define ours as

$$\frac{(\text{perimeter})^2}{4\pi(\text{maximum projected area})}.$$

[3]We have not determined why degradation occurs, but it is most likely to be due to the chaotic effects of making a choice between two merges with very similar costs.

This metric depends only on the shape of the border, not the interior, and it cannot be less than 1. Our combined metric is defined as

$$E_{\text{combined}} = \frac{\text{surface area} + \rho(\text{perimeter})^2/4\pi}{\text{maximum projected area}}, \tag{5.4}$$

with ρ being the weighting factor for the compactness term.

The $L^{2,1}$ metric of Cohen-Steiner et al. [2004] can be shown to be equivalent to ours (without the compactness term), but is expressed differently (as the integral of squared errors in the normal). We feel that our expression of the metric is more intuitive, and the modified compactness term is novel.

5.2.2 Optimisation process

The metric E_{combined}, as defined in equation (5.4), is scale-invariant. However, we would clearly prefer bad charts to be smaller and good charts to be larger, so we define a metric for the entire mesh as

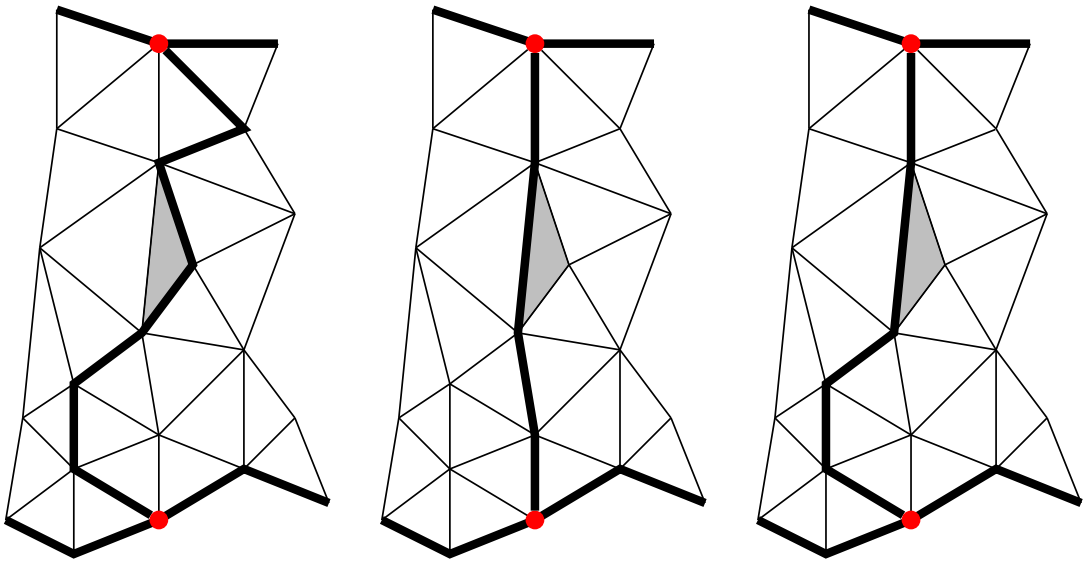
$$E_{\text{mesh}} = \frac{\sum E_i A_i}{\sum A_i}, \tag{5.5}$$

where E_i is E_{combined} for chart i and A_i is the surface area of chart i. Our optimisation process greedily merges charts so as to minimise E_{mesh}, considering only merges that leave all charts with disc-like topology. Note that the denominator is constant and hence has no effect on the optimisation itself; we have included it to make the metric scale-invariant and hence comparable across models. The theoretical lower bound for the metric is $1 + \rho$.

5.2.3 Edge straightening

The greedy merging of charts often leaves rather ragged boundaries. As explained in the next section, it is desirable that the boundaries be straight. We follow Sander et al. [2001] in replacing each boundary with the shortest possible route between the original corners, that does not intersect other boundaries (see Figure 5.3).

There are some cases where straightening an edge leads to extremely poor results. This is usually due to sock-like regions of the mesh, as shown in Figure 5.4. After each straightening, we recompute the segmentation metric for the two charts adjoining the edge. If either new metric exceeds the old by more than some user-specified ratio, the straightening is re-computed but with an additional constraint: the straightened edge may only use those mesh vertices present on the original edge, but it may "short-cut" some vertices (Figure 5.3(c)). This keeps the edge in essentially the same place, but prevents any mesh triangle from sharing more than one edge with the boundary, as occurs for the grey triangle in Figure 5.3.

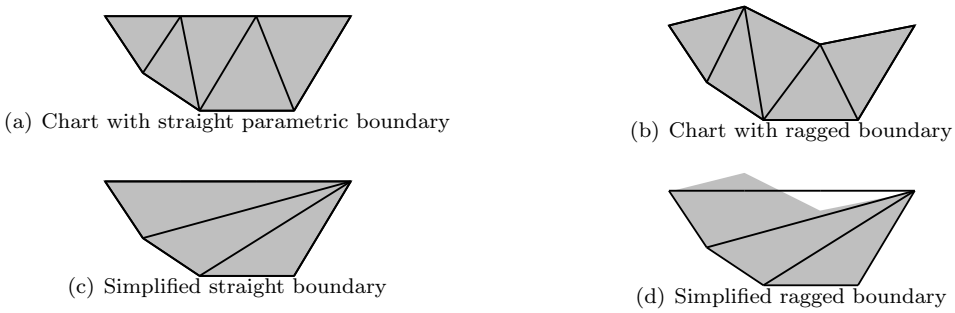


(a) Original ragged boundary (b) Straightened boundary (c) Restricted straightening

Figure 5.3: Boundary straightening. The ragged boundary (a) is improved by replacing it with the shortest path between the marked corner points (b). Note that the grey triangle in (a) touches the boundary on two sides; if this boundary were mapped to a straight line in parameter space, the triangle would become degenerate. If (b) causes either chart to become too curved, we use the straightest path that uses only a subset of the original vertices, as shown in (c).

(a) Original boundary (b) Shortened boundary

Figure 5.4: Edge straightening and socks. The original segmentation (left) cuts the sock neatly in half. After edge straightening (right) the cut is much shorter, but one chart covers most of the sock.



(a) Chart with straight parametric boundary

(b) Chart with ragged boundary

(c) Simplified straight boundary

(d) Simplified ragged boundary

Figure 5.5: Simplification and chart boundaries. On the left, a simplified version of the mesh covers the same area in parametric space. On the right, the ragged boundary cannot be simplified because it alters the coverage in parametric space.

5.3 Flattening

Once the charts have been created, we proceed to flatten them onto the plane. The conditions listed in Section 5.1 depend only on the intrinsic properties of the mesh[4], and not on any embedding. It follows that any parametrisation scheme that depends only on these intrinsic properties can be trivially adapted to work in animation space, using the $L_{2,2}$ metric. Fortunately, all the flattening methods discussed so far have this property.

Our parametrisation scheme is an implementation of least squares conformal maps [Lévy et al., 2002], with some additional processing to make the result suitable for simplification. Least squares conformal maps (LSCMs) have an advantage over several other schemes in allowing the boundary to be solved, rather than pre-specified, and in particular it allows for non-convex boundaries. When the charts themselves are far from circular, this may enable a better parametrisation (refer to Section 5.1 for other schemes with this property).

Unfortunately, an arbitrary boundary in parameter space is undesirable when used with simplification schemes. Simplification should not allow parts of parameter space to appear or disappear from the model. Hence, boundary vertices can only be removed if they lie on a straight section of boundary, on all charts on which the vertex lies (Figure 5.5). It is thus necessary to force the parameter-space boundaries to lie straight whenever possible.

The LSCM algorithm allows for an arbitrary number of boundary vertices to be fixed in advance. We use a first pass in which the minimum number (two) is fixed to determine the general layout, and then pin all the boundary vertices in a second pass. The corners are left as is, while the edge vertices are placed along straight lines between the corners, distributed by the lengths of the mesh edges between them. This scheme can, however, fail for several reasons:

1. A chart may have only two corners with other charts. In this case, we introduce an artificial corner, midway between the existing corners, as shown in Figure 5.6.

2. In the first pass, Lévy et al. [2002] have shown that the parametrisation is guaranteed to

[4]Those properties that could be measured by a Flatlander "inhabiting" the mesh

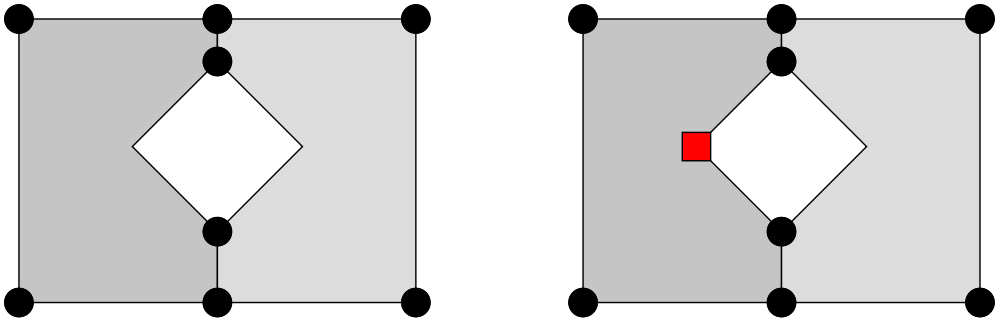


Figure 5.6: Introduction of a third corner. (a) The central chart initially has only two corners (black dots). (b) We introduce a third corner (red square) to facilitate boundary straightening.

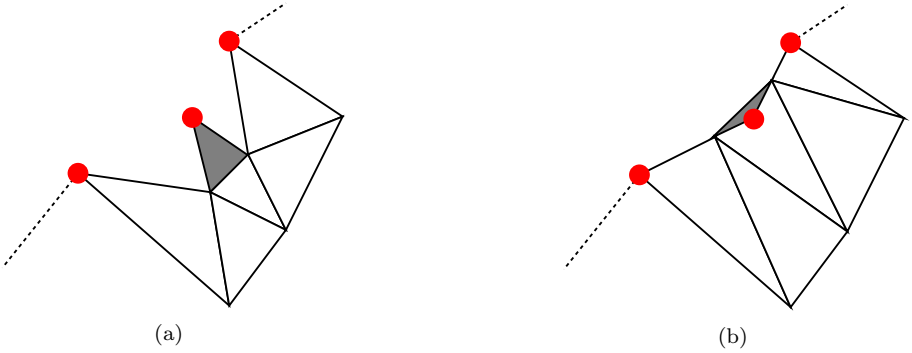


Figure 5.7: Forced triangle flips. (a) A chart that has been laid out with an unconstrained boundary. The red dots mark points where three charts meet. (b) The boundary edges are pinned to straight lines. This forces the grey triangle to flip over, and also to lie outside the chart.

be free of flipped or degenerate triangles. We have found that numerical instability may lead to degenerate or marginally flipped triangles. To counter this, we follow all passes with Sander et al.'s non-linear optimisation, which can also improve the overall quality of the parametrisation [Lévy et al., 2002]. The implementation is discussed in Section 5.3.1.

3. In the second pass, there may be triangles that are forced to flip due to the boundary pinning (especially triangles with all three vertices on boundaries — see Figure 5.7). If this happens, then a third pass is done where the corners are instead placed around a unit circle (distributed by the arc length), and the edge vertices are again placed along straight lines.

4. The third pass may also fail, particularly if there is a triangle whose vertices all lie along the *same* boundary edge (since the pinning then forces this triangle to be degenerate). This will not usually happen, because the chart edge straightening (Section 5.2.3) eliminates such cases. However, the boundaries of holes in the mesh cannot be altered, and may cause this type of failure. If the third pass fails, we introduce an extra corner, as shown in Figure 5.8. We select the vertex whose sum of distances (in animation space) from the neighbouring

Figure 5.8: Insertion of an extra corner. (a) The original parametrisation (dashed) and the straightened form (solid). (b) An extra corner is added to allow the straightened form to more closely approximate the original.

corners exceeds the distance between those corners by the greatest amount. After adding the corner, the algorithm restarts from the beginning.

We note that the last failure case is a possible area for future research. Currently, either all edges are straightened, or none are. It should be possible to obtain partial straightening in the presence of holes.

The steps above ensure that we will always have a parametrisation that is free from flipped or degenerate triangles. However, this is not quite sufficient to guarantee a one-to-one parametrisation, because the boundary may self-intersect. After flattening, the boundary is searched for self-intersections. If an intersection between two edges is found, we partition the chart into two subcharts, and recursively apply the flattening algorithm to each. We use a crude heuristic to perform the split: we perform a two-source breadth-first search to assign every face to one subchart or the other, based on which of the two edges involved in the intersection is closer to the face in question. Distance is measured as the minimum number of edges that must be crossed in a path. Since each subchart will be connected and the original chart had disc-like topology, so will the subcharts. We have found that this often fails to eliminate all self-intersections in the first split, and so more splits than necessary are made. We have not attempted to improve the splitting algorithm because these self-intersections are quite rare, and become even more so if the initial segmentation produces good results.

5.3.1 Stretch optimisation

As discussed above, we follow LSCM with a non-linear optimisation to reduce the stretch. The search is implemented essentially as described by Sander et al. [2001]: individual vertices are adjusted by performing a line search along a randomly chosen direction. However, we found that with suitable pre-computation it was possible to improve performance by 1–2 orders of magnitude over a naïve implementation.

Let f be the function that maps parametric coordinates to geometric coordinates. The L_2 texture

stretch of a triangle in a static model is defined as

$$\sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}},$$

where σ_1 and σ_2 are the singular values of the Jacobian of f. Sander et al. [2001] show that

$$\frac{\partial f}{\partial s} = \frac{1}{2A}\big[\mathbf{p}_1(t_2 - t_3) + \mathbf{p}_2(t_3 - t_1) + \mathbf{p}_3(t_1 - t_2)\big], \tag{5.6}$$

and similarly for $\frac{\partial f}{\partial t}$, where the vertices have positions $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ and texture coordinates (s_1, t_1), (s_2, t_2), (s_3, t_3), and A is the area of the triangle in parameter space. We can rewrite this as

$$J = \begin{pmatrix} \frac{\partial f}{\partial s} & \frac{\partial f}{\partial t} \end{pmatrix} = \begin{pmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 \end{pmatrix} S, \tag{5.7}$$

where S is a 3×2 matrix that depends only on the parametric coordinates of the vertices. However, the correct space to measure the stretch is not animation space, but rather normalised animation space (see Section 3.3.2), where the Euclidean norm is geometrically meaningful. This means that we need the singular values of CJ, which can be determined from

$$(CJ)^T CJ = J^T P J = S^T \begin{pmatrix} \langle \mathbf{p}_1, \mathbf{p}_1 \rangle_{2,2} & \langle \mathbf{p}_1, \mathbf{p}_2 \rangle_{2,2} & \langle \mathbf{p}_1, \mathbf{p}_3 \rangle_{2,2} \\ \langle \mathbf{p}_2, \mathbf{p}_1 \rangle_{2,2} & \langle \mathbf{p}_2, \mathbf{p}_2 \rangle_{2,2} & \langle \mathbf{p}_2, \mathbf{p}_3 \rangle_{2,2} \\ \langle \mathbf{p}_3, \mathbf{p}_1 \rangle_{2,2} & \langle \mathbf{p}_3, \mathbf{p}_2 \rangle_{2,2} & \langle \mathbf{p}_3, \mathbf{p}_3 \rangle_{2,2} \end{pmatrix} S. \tag{5.8}$$

The observation that allows us to significantly accelerate the optimisation process is that the middle factor above is invariant for each triangle and may be computed in advance. This makes the speed of the optimisation independent of the number of bones, since this is a 3×3 matrix regardless of the dimension of animation space.

5.4 Packing

We use the heuristic of Sander et al. [2001] of packing charts into bounding rectangles, then filling in the rectangles alternately left-to-right and right-to-left, as shown in Figure 5.9. The rectangles are first sorted by height, and the optimal scaling factor (i.e., such that all the charts fit without being smaller than necessary) is found by binary search. While there are more sophisticated heuristics, this approach is very simple to implement and generally makes reasonable use of the available space.

We also follow Sander et al. [2001] in attempting to orient the charts to minimise the area used by the bounding rectangle, with the long axis aligned vertically. However, we approximate this by sampling the range of orientations rather than optimising it exactly.

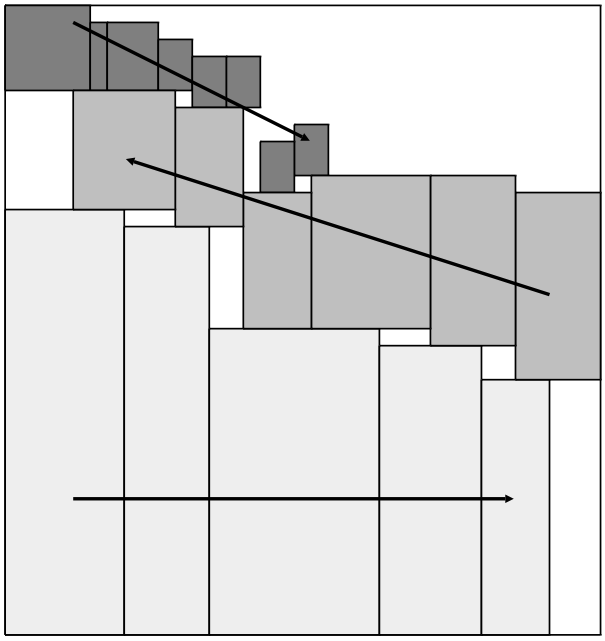


Figure 5.9: Packing of charts into an atlas. The charts are packed alternately left-to-right and right-to-left. The colours in the figure indicate three passes.

5.5 Summary

Parametrisation conventionally occurs in three phases: segmentation, flattening, and packing. Our segmentation implementation was necessary for the development of level-of-detail (discussed in Chapter 6), but the results show that further work is required to produce a fully-automated solution that can be relied upon in all cases. For packing, we simply used an existing algorithm; our real contribution is in the area of flattening. We showed how one algorithm could be adapted to animation space simply by using the $L_{2,2}$ metric in place of the 3D Euclidean metric. This is a very general approach, and so more sophisticated parametrisation algorithms could be adapted just as easily. The results will show that this makes an improvement to stretch in practice, without it becoming prohibitively slow.

Chapter 6

Level-of-detail

Level-of-detail (LOD) techniques aim to display an object with only the amount of detail that is necessary at the time. High-detail representations can be used in situations where quality is important, while low-detail representations can be used to improve frame rate where the loss of quality will not be noticed. Typically, low-detail representations are used when the object is far away, although there are approaches that consider the attention of the user, e.g., Brown et al. [2003]). Using low-detail approximations for unimportant objects allows them to be drawn faster, enabling real-time rendering of scenes that nominally contain millions of polygons.

In this chapter, we make two contributions. The first is to adapt an existing LOD method, namely appearance preserving simplification, to operate directly in animation space. This is done in a similar manner to the adaptation of LSCM in the previous chapter, namely by substituting the $L_{2,2}$ metric for the Euclidean metric. The second contribution is *influence simplification*, which simplifies models not by removing geometric detail (edges, faces and vertices), but by removing influences of bones on vertices. We show how the $L_{2,2}$ metric can be used to do this optimally, which can produce significant savings for models with superfluous influences. We also demonstrate that influence simplification can be integrated into the geometric LOD structures to produce a unified LOD system.

6.1 Background

LOD is a well-explored field, and providing a full survey here would be impractical. We will cover only a few techniques that have application to our research. For the interested reader, Garland [1999] provides a survey of the available methods.

One of the most popular LOD data structures is the progressive mesh [Hoppe, 1996]. The fundamental operation of the progressive mesh is the edge collapse, shown in Figure 6.1. The collapse removes one edge and up to two faces from the mesh. By applying the edge collapse repeatedly one can produce a sequence of progressively simpler meshes. Algorithms that use the progressive

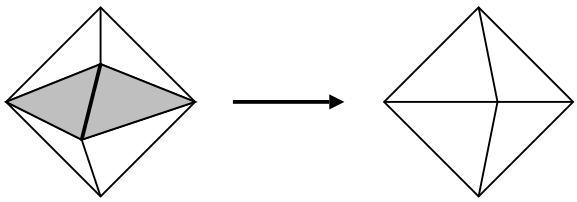


Figure 6.1: The edge collapse operation. The thick edge is collapsed, causing the shaded faces to vanish.

mesh structure differ in the way they choose which edge to collapse at each stage. Most employ an error metric that measures the deviations between the original and the new mesh. A popular and efficient error metric is the quadric error metric (QEM) of Garland and Heckbert [1997], which has also been extended to handle surface attributes such as colour [Garland and Heckbert, 1998, Hoppe, 1999]. However, many other metrics exist, trading off mesh quality against speed of evaluation.

A common problem with purely geometric error metrics is that they do not take texture coordinates into consideration. If geometry is preserved but texture coordinates are not, then textures may be displaced along the surface, a phenomenon known as *texture sliding*. Cohen et al. [1998] and Sander et al. [2001] use an error metric that bounds the maximum distance between points on the original and simplified surfaces that have corresponding texture coordinates. Because the technique is based directly on texture coordinates, both geometric error and texture sliding are addressed within a single framework. Both sets of authors also sample normal maps that allow the original normal field to be used on the simplified mesh, thus maintaining high-fidelity lighting. These approaches yield high quality object display even with low triangle counts, although they require a parametrisation of the mesh (as discussed in Chapter 5).

Progressive meshes have level-of-detail applications beyond simply pre-computing a few static representations [Hoppe, 1996]. They provide an almost continuous sequence of presentations, allowing a triangle budget to be met to within one triangle. Furthermore, it is possible to do *selective refinement* [Hoppe, 1997, Xia and Varshney, 1996]: that is, applying a variable amount of simplification across the model. This is most commonly used with terrains, so that the ground that one stands on has high resolution, while distant hills are at a lower resolution. Selective refinement can also be used to create high-quality silhouettes. For other applications, the extra overhead usually outweighs the gains on modern hardware [Dietrich, 2000], although there has been some work to optimise the rendering of selective refinements [El-Sana et al., 1999].

6.1.1 LOD and animation

In this section we will briefly review previous attempts to combine level-of-detail techniques with animation. The work that exists mainly focuses on using a simpler model of the high-level animation for unimportant objects, rather than on geometric simplification.

The Cal3D [Cal3d] library for 3D character animation provides a level-of-detail mechanism, based

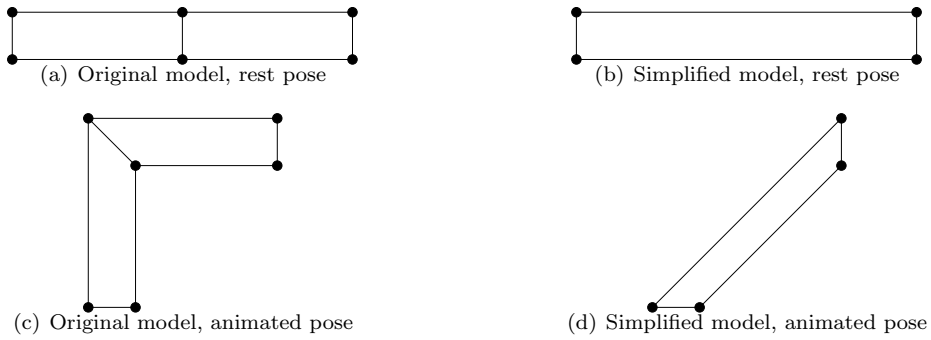


Figure 6.2: "Lost elbow" problem. A naïve LOD algorithm may simplify (a) to (b) (an apparently good approximation), ignoring the fact that this causes (c) to reduce to (d) (a poor approximation).

on progressive meshes with quadric error metrics [Garland and Heckbert, 1997]. The scheme considers only the rest-pose positions of vertices, ignoring the structure of the skeleton. This has the capacity to create progressive meshes which animate very poorly. For example, consider simplifying an arm with an elbow joint. Clearly, high detail is required around the elbow to produce the appearance of a curved surface. However, if the rest pose has the arm extended then the elbow may be completely eliminated, as illustrated in Figure 6.2, and more graphically in Figure 1.3.

Kähler et al. [2001] consider the inverse problem of adding detail to a mesh to smooth over joints. They use a variation of subdivision surfaces to add triangles only in high-curvature parts of the mesh, and the curvature is evaluated at run-time. However, the technique can only smooth over sharp creases; it cannot recover detail that was lost in simplifying the mesh in the first place. The animation is also a mass-spring system, rather than skeletally-based character animation.

Adaptive level of detail for human animation (ALOHA) [Giang et al., 2000] applies level-of-detail ideas to animated characters. However, it does not use either progressive meshes or SSD. Instead, it uses an anatomically-based approach to skinning, using bones, muscle, fat and skin. The muscles are modelled as super-ellipsoids which distort as the bones move. The fat and skin are attached with springs, with spring end-points corresponding to skin vertices. Level of detail is controlled by the number of springs used, and hence the number of vertices on the skin. The complex animation process makes this approach unsuitable for real-time rendering (certainly for real-time rendering of crowds). The subdivision process used to create the springs does not take into account the curvature of the skin, so simple models will be of much lower quality than similar models created as progressive meshes.

Shamir and Pascucci [2001] consider LOD for very general animated meshes with a pre-specified animation path (such as for movies or cut-scenes in games). Like Cal3D, it uses progressive meshes with a quadric error metric, and computes a decimation from a single pose. However, it only decides which edges to collapse at run-time. This is made possible by organising the collapses into a dependency graph, and adjusting the choice of edges to collapse based on the graph and the cost of each collapse in the *current* frame. This method produces good results, but

- requires a pre-specified animation path, making it unsuitable for dynamic animation in games;
- requires more CPU processing due to the view-dependent LOD[1] algorithm [Dietrich, 2000].

Geometry videos [Briceño et al., 2003] offer a different approach, based on the geometry images of Gu et al. [2002]. The mesh is first parametrised (see Chapter 5), then re-sampled over a regular grid in parameter space. The result is a 2D image with 3 channels, for x, y and z coordinates. The mesh geometry is stored or transmitted simply as an image stream (video), which can be compressed using existing video compression methods. Level-of-detail is obtained by sub-sampling the images, while being careful to preserve cut boundaries. While geometry images have their own advantages (particularly the implicit connectivity), they suffer the same disadvantages as all re-meshing strategies i.e., it is impossible to recover the original mesh exactly.

Deformation sensitive decimation [Mohr and Gleicher, 2003a] is more similar to our approach. Instead of applying simplification in a single rest pose, a sampling of poses is used. The quadric error metric [Garland and Heckbert, 1997] for each potential edge collapse is summed over the samples, and the collapse with the lowest total penalty is chosen. This can eliminate the "lost elbow" problem, provided that a good sampling is chosen. DeCoro and Rusinkiewicz [2005] improve this technique by using stratified random sampling to automatically sample the pose space, given a description of its probability density function; this frees the user from having to select a representative set of samples. Huang et al. [2006] use a similar metric for a pre-specified animation (with every frame forming a sample), and augment it with a term that promotes detail-preservation in areas that undergo large frame-to-frame deformations. Nevertheless, these methods all depend on a sampling of the pose space, and the computation time scales with the product of the number of edges and the number of sample poses. In contrast, our method is analytic and does not require any sampling; if samples are available they can be used to extract statistics, but the running time is independent of the number of samples.

6.2 Appearance-preserving simplification

Our level-of-detail scheme is an extension of appearance-preserving simplification (APS) to animation space. APS suits our needs well because it uses a parametrisation to match up corresponding points between the simplified model and the original, which would otherwise be difficult to do in such a high-dimensional space. We also considered adapting a quadric error metric [Garland and Heckbert, 1997], but since it relies on the solution of a least-squares problem for every potential collapse, it would scale very poorly to the high dimensions used in animation space.

[1] Here the "viewpoint" is determined by position in time rather than in space, but the underlying approach is the same.

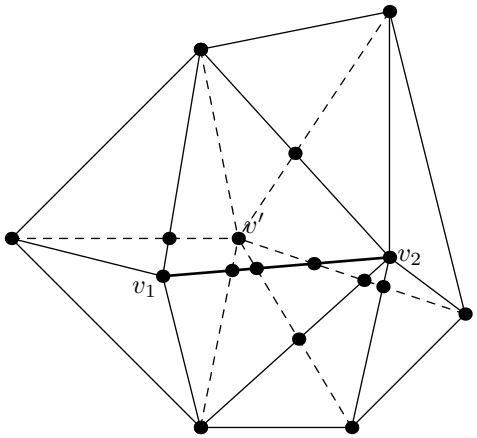


Figure 6.3: An edge collapse in parameter space, where v_1v_2 is collapsed to v'. Dashed lines represent the new triangles and dots represent cell corners.

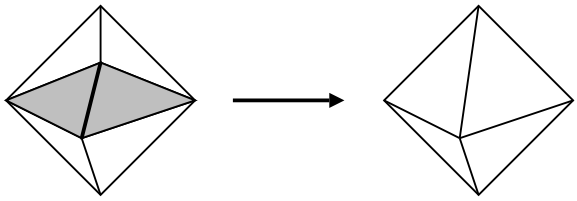


Figure 6.4: The half-edge collapse. Compare to Figure 6.1, where a new vertex is created.

6.2.1 Review of APS

APS scores potential collapses by the maximum deviation they could introduce between the original and the simplified surface, in terms of the correspondence implied by the parametrisation. Computing the maximum deviation precisely is prohibitively expensive, so some conservative approximations are used. For each face, an axis-aligned bounding box is maintained that bounds the offset vector between any point on that face and the corresponding point in the original mesh. Figure 6.3 shows how this information is used to score a potential edge collapse. In parameter space, the old and new triangles are overlaid, which partitions the local neighbourhood into cells. Within each cell, the deviation between old and new is linear, and the prior bounding box is constant. It follows that the maximum deviations must lie at the corners of cells. For every cell corner in the diagram, the new deviation is added to the bounding box for each of the original faces on which it is incident, and the greatest length amongst these deviations is deemed to be the cost of the collapse.

When a collapse is actually performed, the bounding box information must be propagated to the new faces. For each new face, the same calculation is done for each incident cell corner, and the resulting bounding boxes are merged to produce the new bounding box.

We have followed Sander et al. [2001] in using the half-edge collapse. This is a restricted form of the edge collapse, in which the edge is collapsed to one of the existing end-points rather than to a new point (Figure 6.4). This is computationally cheaper (in that no optimisation is needed

to determine the new vertex position), and also leads to more compact representations as there are no new vertices to encode. The disadvantage is that the quality of the simplified meshes is worse for the same number of triangles. In particular, volume loss effects are increased because the simplified model cannot protrude beyond the convex hull of the original. We leave computing the optimal location for a new vertex in animation space as a possible area for future research.

Sander et al. [2001] also use a simpler "memoryless" form of the error metric. They do not maintain the bounding box information that is used by standard APS, but instead define the cost to be the largest deviation between the immediately previous and the new neighbourhood. This is just the largest deviation at any cell corner in Figure 6.3. Because only half-edge collapses are used, there are also generally far fewer cell corners to consider.

6.2.2 APS in animation space

We have implemented both the standard and the memoryless forms of APS, and adapted them to animation space. For the memoryless form, the adaptation is simply a matter of replacing the Euclidean metric with our $L_{2,2}$ metric.

For the standard form, some changes need to be made to accommodate the bounding-box propagation. Our initial implementation used a bounding box aligned to the axes of animation space, but this is in fact a poor choice because the axes are not orthogonal in terms of the corresponding inner product. Appendix B.3 shows that in some circumstances, certain displacements have no cost in terms of the $L_{2,2}$ norm, but an axis-aligned bounding box that contains them will be unnecessarily large. Instead, we compute the bounding boxes in normalised animation space (see Section 3.3.2), in which the Euclidean metric corresponds to the $L_{2,2}$ metric in animation space. Deviations with zero (or almost zero) $L_{2,2}$ norm will become zero (or very small) when transformed to normalised animation space, ensuring a suitably sized bounding box.

A disadvantage of this approach is that the transformation to normalised animation space destroys the sparsity of the vectors concerned. This does not have a major impact on performance (since the bounding box updates constitute only a small part of the running time), but it does greatly increase the memory requirements for models with many bones.

6.3 Influence simplification

The progressive mesh structure discussed so far reduces complexity by removing edges and faces from the model. However, the complexity of a skinned model arises not only from the number of vertices and faces, but also from the number of bones that influence each vertex. Reducing the number of influences on a vertex reduces the cost of rendering it, without decreasing the amount of detail in the model. In the extreme case, it is possible to reduce all vertices to only one bone influence, giving totally rigid limbs with no smooth transition from one to the next. From

a suitable distance, such rigidity may not be noticeable to a viewer. We refer to the process of reducing the number of influences on a vertex as *influence simplification*.

A naïve approach to influence simplification would be to simply discard influences that are deemed to be minor. We have developed a more sophisticated strategy that makes two improvements:

1. It is able to quantify the error that is introduced, in a manner that allows influence simplification to be combined with a progressive mesh.
2. The error is minimised by adjusting the remaining influences to match the effect of the lost influence as closely as possible.

The $L_{2,2}$ metric provides a natural error metric for influence simplification, as it indicates how much geometric error the simplification is expected to introduce. We consider each influence on an animation-space position $\mathbf{p}$ in turn, determine how to optimally eliminate it, and then choose the elimination that introduces the smallest error. To pose this problem formally, we introduce some additional notation: let $\mathcal{I}(\mathbf{p})$ be the set of indices of the bones that influence $\mathbf{p}$. In order to eliminate a particular influence, say that of bone i, we replace $\mathbf{p}$ with $\mathbf{p} + \mathbf{s}$, where $\mathbf{s}$ is chosen subject to the following:

1. $\mathbf{s}$ has no influences that are not part of $\mathbf{p}$, since otherwise we would be introducing new influences: $\mathbf{s}_j = \mathbf{0}$ for all $j \notin \mathcal{I}(\mathbf{p})$.
2. $\mathbf{p} + \mathbf{s}$ is not influenced by bone i: $\mathbf{s}_i = -\mathbf{p}_i$.
3. $\mathbf{s}$ is a vector, since we wish $\mathbf{p} + \mathbf{s}$ to be a point on the projection hyperplane: $\underline{\mathbf{s}} = 0$.
4. The expected error $\|\mathbf{s}\|_{2,2}$ is minimal subject to the above constraints.
5. This minimum may be not be well-defined, in which case we choose the minimum that also minimises $\|\mathbf{s}\|_2$ (the Euclidean norm).

The constraints are all linear equations, and $\mathbf{s}_{2,2}^2 = \mathbf{s}^T P \mathbf{s}$, so this is just a matter of minimising a quadric subject to linear constraints. While the optimisation can be done directly in this form, it will not exploit any sparsity in $\mathbf{p}$. Instead, we note that since bones not in $\mathcal{I}(\mathbf{p})$ play no role, we can entirely eliminate them from the problem. Let $\mathbf{p}_{\mathcal{I}(\mathbf{p})}$ and $\mathbf{s}_{\mathcal{I}(\mathbf{p})}$ be the restrictions of $\mathbf{p}$ and $\mathbf{s}$ to the dimensions corresponding to the influences on $\mathbf{p}$, and $P_{\mathcal{I}(\mathbf{p})}$ similarly be the submatrix of P consisting of those rows and columns. Then since we have only eliminated zero values from $\mathbf{s}$, we find that

$$\|\mathbf{s}\|_{2,2}^2 = \mathbf{s}^T P \mathbf{s} = \mathbf{s}_{\mathcal{I}(\mathbf{p})}^T P_{\mathcal{I}(\mathbf{p})} \mathbf{s}_{\mathcal{I}(\mathbf{p})}. \tag{6.1}$$

The constraints on $\mathbf{s}$ translate directly to $\mathbf{s}_{\mathcal{I}(\mathbf{p})}$, except that the first is no longer necessary as it is implicit in the dimension reduction. The problem of optimising $\mathbf{s}_{\mathcal{I}(\mathbf{p})}$ now has the same form as the original, but with exactly five linear constraints (items 2 and 3 above) and only $4|\mathcal{I}(\mathbf{p})|$ dimensions.

This type of constrained least-squares problems essentially amounts to transformation to a suitable space (in this case, normalised animation space) followed by a projection onto the subspace defined by the constraints. The details are slightly complicated by the fact that P may be non-singular

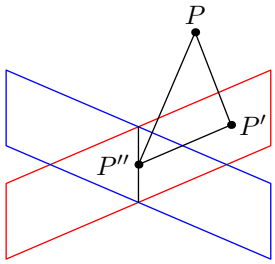


Figure 6.5: Visualisation of two iterations of influence simplification. Projecting P onto the red plane and then onto the line of intersection gives the same result as projecting P directly onto the line of intersection.

and hence the minimum is not well-defined. A description of our implementation may be found in Appendix A.2. The implementation does some computation on $P_{\mathcal{I}(\mathbf{p})}$ as well as on the matrix $A_{\mathcal{I}(\mathbf{p}),i}$ which holds the coefficients for the linear constraints. Since these matrices do not depend on the actual value of $\mathbf{p}$, it is frequently possible to reuse the results of these computations. We exploit this by maintaining the pre-processed results for the last 128 $(\mathcal{I}(\mathbf{p}), i)$ pairs encountered.

In each step, we optimise the new position to fit the previous (possibly already simplified) position as closely as possible, rather than the original position as closely as possible. This is more efficient since we are then working in a lower-dimensional space and we do not need the memory to hold the original position. A formal proof that these are equivalent is given in Appendix B.1, but it may be seen intuitively from the 3D thought experiment conducted in Figure 6.5: consider two planes in space, representing two different sets of linear constraints, and a point, representing the original value of $\mathbf{p}$. Let $\mathbf{p}'$ be the closest point to $\mathbf{p}$ on the first plane, and $\mathbf{p}''$ be the closest point to $\mathbf{p}'$ on the line of intersections of the planes (which represents the points satisfying both sets of constraints). Then clearly $\mathbf{p}''$ is also the closest point on this line to $\mathbf{p}$.

Depending on which lighting method is used (see Section 7.5), we may also need to compute per-vertex tangent planes during rendering. In order to reap the full rewards of the simplification, it is thus necessary to simplify the tangents as well. Since the tangents are not relevant to all uses of the simplified model, we decide which influence to eliminate based only on the position. Whichever influence is removed from the position is also removed from the tangents. After removing the influences, we adjust the tangents to again form an orthonormal basis for the tangent plane using the Gram-Schmidt process [Golub and Van Loan, 1996].

6.3.1 Combination with mesh simplification

Since our implementations of both influence simplification and mesh simplification use the same metric (distance between corresponding points), we can combine them in a hybrid simplification scheme with a single sequence of simplifications, some of which are edge collapses and some of which are influence simplifications. Conceptually, potential edge collapses and influence simplifications

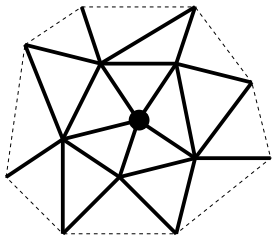


Figure 6.6: Edge updates from an influence simplification. When the central vertex has its position simplified, the edge collapses corresponding to the solid edges must be updated.

share a single priority queue, but for implementation reasons[2] we have a separate queue for each and always process the head-of-queue with the least cost. Apart from the usual maintenance that needs to be done when constructing a progressive mesh, after applying an edge collapse we need to remove the two old positions from the influence simplification queue and insert the new position. After applying an influence collapse, any edge incident on a neighbour of this position must be re-evaluated as a potential collapse (see Figure 6.6).

With memoryless APS, the cost of a simplification is simply the $L_{2,2}$ distance between the original and the simplified vertex. For standard APS, it is necessary to consider the bounding box information. For each incident triangle, there is an associated bounding box that provides a bound on the existing deviation of the vertex; we can thus take their intersection and add the new offset ($\mathbf{s}$) to get a bound on the total deviation after simplification. After applying a simplification, we take this bounding box for the vertex's total deviation and propagate it back to the incident faces, by making the new face bounding box the union of the old face bounding box and the new vertex bounding box.

6.4 Summary

As with parametrisation, we have shown that an existing LOD framework can be readily adapted to animation space, in order to produce approximations that work well, on average, across poses. Memoryless APS works particularly well for this purpose, as it does not require high-dimensional bounding boxes to be maintained. We confirm the improvements experimentally in Chapter 8.

We have also introduced the novel concept of influence simplification — removing influences of bones on vertices to produce a sparser model. The results indicate that for some models, this can make a substantial improvement, although in other cases it makes little difference and can even be slightly worse. We have discussed influence simplification in the context of run-time level-of-detail, but it could potentially also be applied as a once-off process to eliminate unnecessary influences in a model, or to control the maximum number of influences on any vertex. Our rendering implementation (Chapter 7.4) allows for an arbitrary number of influences, but in the absence of run-time LOD or on legacy hardware, it may be more efficient to use a vertex program with a fixed number of influences per vertex.

[2]We use a C++ templatised priority queue class, which makes it inconvenient to place two different types in the same queue.

Chapter 7

Rendering

To render a large crowd of animated characters at interactive rates, we must take advantage of rendering hardware — even with level-of-detail optimisations, current general purpose CPUs are still too slow. In this chapter we will consider exclusively the task of interactive rendering on standard consumer video cards. The off-line rendering used in the film industry is less performance critical, and will benefit less from the low-level optimisations described in this chapter.

We start by considering frustum culling (eliminating objects that lie entirely off-screen) using the $L_{2,\infty}$ metric, and our implementation of level-of-detail with a pixel error tolerance. These concepts are generic and could be applied to any rendering system. We then consider how to efficiently perform animation projection, taking into account the variable number of influences per vertex and the limitations of hardware. Finally, we describe two methods for encapsulating normal information in textures, for high-fidelity lighting on simplified models.

Our implementation is based on OpenGL [Segal and Akeley, 2006], and the following explanations are in terms of the terminology and features of OpenGL.

7.1 Target environment

In 2001, the GeForce 3 video card introduced programmable pipelines to consumer video cards [Lindholm et al., 2001]. Since then, the level of programmability has steadily increased, but in order to get good performance, the programming model is still restricted in a number of ways. We have chosen to target the feature-set available in the GeForce 6 and 7 video cards (the so-called Shader Model 3.0). While this model has a rich feature-set, it has already been superseded by the GeForce 8 series which implements Shader Model 4.0 and the features in Direct3D 10 [Blythe, 2006].

Figure 7.1 shows the stages of a programmable graphics pipeline that are relevant to this chapter. Some of the important features and limitations of the implementation in the GeForce 6 and 7 series are listed below:

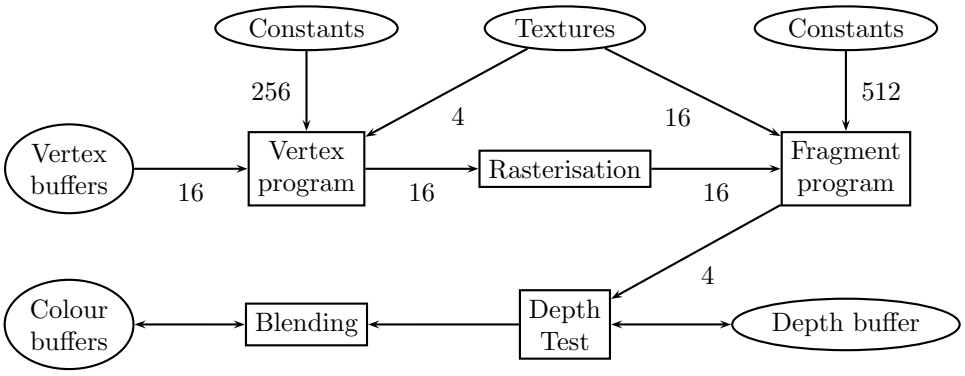


Figure 7.1: A programmable graphics pipeline. Ovals represent data stores while boxes represent processes. Numbers represent the maximum number of inputs/outputs supported by a GeForce 6600.

- The primary data type is a vector of four floating-point values, referred to here just as a "vector".
- A fixed number of vectors (up to 16) may be associated with each vertex. These are typically used for positions, normals and texture coordinates, but also for other per-vertex data. Vertex programs also have access to a small bank of constants (256 vectors), which can be addressed with a variable expression, and textures. Texture lookups are not optimised, as they are intended primarily for fragment programs, and hence introduce a large latency. Vertex programs support flow control, including branching and looping. The "constants" are fixed from the perspective of the vertex program, but the CPU program may alter them between draw calls through the OpenGL API.
- Fragment programs receive up to 16 vectors from interpolation of the vertex program outputs. Three levels of precision are supported: 12-bit fixed point (a range of -2 to 2), 16-bit floating point (half-precision) and 32-bit floating point (full-precision). The lower-precision instructions are significantly faster. Again, a small bank of constants is available, but they can only be addressed with constant indices. The fragment pipelines are designed for efficient texture lookups, and it is common practice to encode complex functions in texture maps.

  Fragment programs support both predicated execution[1] and dynamic branching. Dynamic branching is relatively expensive, particularly when fragments that are being processed in parallel follow different code-paths.

  Fragment programs may modify the depth of a fragment, and may output up to four separate colours.
- Floating point textures and colour buffers are supported at both 16- and 32-bit precision, but certain state combinations are not supported. In particular, texture filtering and framebuffer

[1] *Predicated execution* means that instructions may be labelled as conditionally executed depending on a flag register. This avoids the overheads of true branching, but the instructions have a cost regardless of whether they are executed or not. Predicated execution is best suited to conditionals with a small amount of calculation in each branch.

blending are not supported at 32-bit precision.

We have written vertex and fragment programs in Cg, a high-level shading language developed by NVIDIA [2005a]. It generates low-level assembler code that is loaded into OpenGL using the `GL_ARB_vertex_program` and `GL_ARB_fragment_program` extensions. Apart from the usual advantages of using a higher-level language, we found that Cg had several benefits over coding with either the low-level assembly language or the high-level OpenGL shading language:

- It exposes the lower-precision types (`half` and `fixed`). These are not exposed by the OpenGL shading language, and can only be used in the assembly language by invoking NVIDIA-specific extensions. In contrast, Cg has a number of backends, and can either output assembler code that uses these extensions, or produce portable code by mapping the low-precision types to a generic floating-point type.
- In contrast to the OpenGL shading language, it is possible to examine the assembler code that is produced. In a number of cases, this has allowed us to identify cases where instructions could be saved by doing some pre-processing.
- While experimenting with different methods for lighting and shading, it is convenient to have a single program with `if` statements to control the execution flow. Cg allows these switches to be labelled as seldom changing; when the user switches lighting models, the program is automatically recompiled and the `if` statement is optimised away. In contrast, doing the same thing with the low-level language requires the program to be constructed at run-time, which complicates debugging of compilation errors, as well as requiring manual register allocation.

7.2 Frustum culling

A simple but effective technique to speed up rendering is to entirely skip objects that lie completely outside the view frustum. To be useful, the test must be much cheaper than simply rendering the object regardless, and it is standard practice to test against bounding volumes.

In Section 3.3.3 on page 26, we developed an L_∞ metric to measure the maximum possible distance between two points in the animation space. By choosing some fixed point $\mathbf{c}$ in animation space and finding the maximum of $\|\mathbf{c} - \mathbf{p}\|_{2,\infty}$ over all vertices $\mathbf{p}$ of the model, we can establish a bounding sphere. This sphere is guaranteed to contain the object in any pose, provided that the skeleton does not violate the stored bounds on bone translation or scaling.

In order to minimise the cost of transforming the centre $\mathbf{c}$ to object space (i.e., computing $G\mathbf{c}$), we have chosen values for $\mathbf{c}$ that are only affected by the root bone, and hence are already in object space. Specifically, we have taken the expected object-space position of the centroid of the object, computed using the uniform rotations and fixed translations model (Section 3.3, page 24). We use this model even if we have more accurate expectations, because the actual distribution is irrelevant and can introduce unwanted bias into the location of the centre.

At run-time, we need only determine whether each bounding sphere intersects the frustum. We reject the object if it lies entirely on the outside of any of the planes defining the frustum; this is easily computed by comparing the distance of the centre to the plane against the radius. This is a conservative test, because the sphere may intersect several of the planes without intersecting the frustum itself.

7.3 Level-of-detail

Since we are rendering characters that can be expected to have a roughly uniform distance from the viewer (over the surface of an individual character), we do not need selective refinement methods; these methods are more suitable for terrain modelling where the terrain exists at a wide range of distances. Instead, we sample the progressive mesh to create a discrete set of representations, each with half as many influences as the previous one. This allows us to compute texture coordinates, triangle indices and so on, in advance, and prevents the CPU from becoming a bottleneck during rendering.

We use *geomorphing* [Hoppe, 1996] to smooth the discontinuities between these discrete levels of detail, which otherwise manifest as "popping" as one representation is replaced with another when the viewing distance changes. Geomorphing linearly interpolates vertices between the levels of detail. Based on the distance of the object from the viewer, a real-valued level of detail l is chosen, and this is used to interpolate between the discrete levels $\lfloor l \rfloor$ and $\lfloor l \rfloor + 1$. Let $\mathbf{p}^{\lfloor l \rfloor}$ be a position in the more refined representation (level 0 being the original model); by tracking $\mathbf{p}$ through edge and influence collapses, one may find a corresponding point $\mathbf{p}^{\lfloor l \rfloor + 1}$ in the coarser representation. Geomorphing places the vertex at

$$\mathbf{p}^l = \mathbf{p}^{\lfloor l \rfloor} + (l - \lfloor l \rfloor)(\mathbf{p}^{\lfloor l \rfloor + 1} - \mathbf{p}^{\lfloor l \rfloor}). \tag{7.1}$$

As long as l varies continuously, so will the chosen representation. We use a user-specified pixel error tolerance p to choose the largest value for l (lowest detail) for which we can guarantee that this error bound is not exceeded. Because we wish to use a conservative bound, we use the minimum depth of any point on the bounding sphere. We determine a lower bound ϵ on the geometric error that could give rise to this on-screen error. Let w be the width of the viewport (in pixels) and 2θ be the horizontal field of view (we use the horizontal dimensions since they are typically larger than the vertical dimensions). The worst case occurs when the object is at the edge of the viewport, as shown in Figure 7.2. Referring to the notation of the figure, the ratio $\epsilon : q = d : \frac{w}{2} \cot\theta$ and $q = p\cos\theta$, giving

$$\epsilon = \frac{2pd\sin\theta}{w}. \tag{7.2}$$

Each discrete level of detail has an associated ϵ^l value, which is simply the largest cost of any simplification encountered in reducing the original model to that representation. Since our sim-

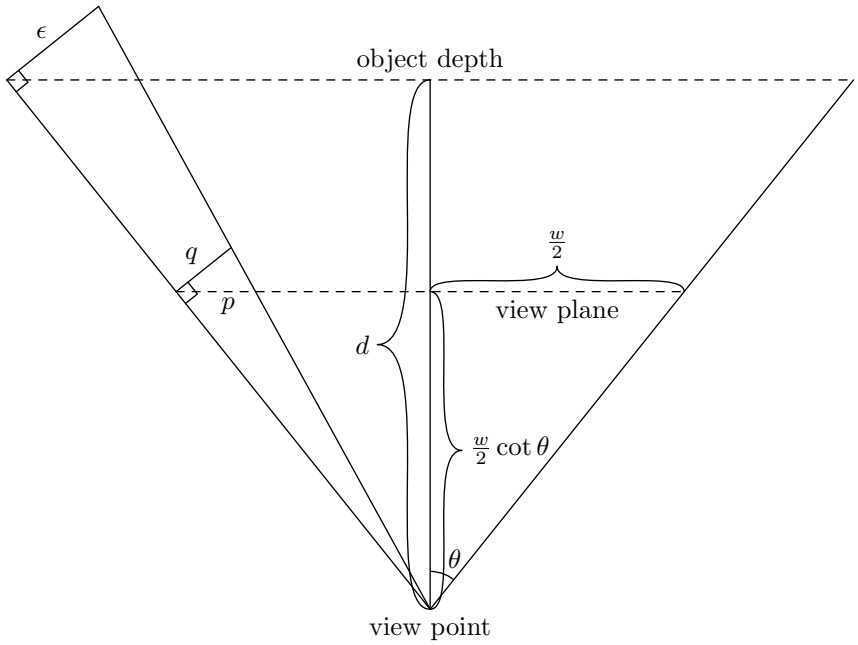


Figure 7.2: LOD selection

plification schemes measure cost in terms of geometric deviation, we can simply select the level whose ϵ^l equals ϵ. Of course, there will generally be no exact match, so we choose l by linear interpolation between the two levels of detail that bracket ϵ.

7.4 Vertex transformation

With a static character, vertex transformation can be handled entirely by OpenGL: we simply specify the transformation matrix using standard OpenGL API calls. For our animated characters, however, we have to compute the matrix-vector product $G\mathbf{p}$, where G is a $4 \times 4b$ matrix and $\mathbf{p}$ a $4b$-vector. The hardware limit of 16 vertex attributes is a more severe limitation than for SSD, because we must pass a 4-vector per influence rather than just a scalar weight. Geomorphing doubles these requirements, and even worse, one of our lighting methods uses per-vertex tangents, which further triples the requirements (a six-fold increase in total). For these reasons, it is not practical to perform vertex transformations simply by using a vertex program that operates on vertex attributes.

For comparison, we have implemented four methods of vertex transformation: two forms of purely CPU-based transformation, a two-pass GPU approach that uses the fragment pipeline for vertex transformation, and a single-pass GPU approach that uses vertex texture lookups; each is described in one of the following subsections. All of these approaches implement geomorphing in a similar manner. Referring to Equation (7.1), denote $l - \lfloor l \rfloor$ by Δl and $\mathbf{p}^{\lfloor l \rfloor + 1} - \mathbf{p}^{\lfloor l \rfloor}$ by $\Delta\mathbf{p}^{\lfloor l \rfloor}$. We precompute $\mathbf{p}^{\lfloor l \rfloor}$ and $\Delta\mathbf{p}^{\lfloor l \rfloor}$ to make the evaluation of Equation (7.1) more efficient. Furthermore, we ensure that $\mathbf{p}^{\lfloor l \rfloor}$ and $\Delta\mathbf{p}^{\lfloor l \rfloor}$ are influenced by the same set of bones, potentially introducing zero

influences to align them. This allows the interpolation to be done on an influence-by-influence basis, without burdening the rendering process with checking for missing influences.

7.4.1 CPU transformation

We transform the vertices on the CPU, using SSE[2] to accelerate the computations. We use one array to hold all the elements of the $\mathbf{p}^{\lfloor l \rfloor}$ and $\Delta\mathbf{p}^{\lfloor l \rfloor}$ vectors (interleaved), a second to hold all the bone indices, and a third to hold the number of influences per vertex. Each vertex is transformed in turn, as shown in Algorithm 1.

Algorithm 1 Vertex transformation on the CPU

```
for all attributes ∈ {position, tangent_1, tangent_2} do
  for all vertices do
    v ← 0      //3D position/tangent
    for each influencing bone j do
      v ← v + G_j (p_j^⌊l⌋ + Δl Δp_j^⌊l⌋)
    end for
    write v to appropriate buffer
  end for
end for
```

We have also implemented a variant of this technique in which the outer loop iterates over bones, rather than over vertices (Algorithm 2). For each bone, we store a list of all vertices influenced by that bone and the influences themselves. The disadvantage of the former method is that the inner loop iterations are short and highly variable in length, which is bad for branch prediction. In contrast, this alternative method has relatively long-running inner loops (the number of vertices influenced by each bone). It also uses one transformation matrix at a time, rather than randomly accessing them. The disadvantage is that updates to the vertex buffer are not sequential (although the inner loop iterates over the vertices in order, so accesses have some coherence).

Algorithm 2 Alternative vertex transformation on the CPU

```
for all attributes ∈ {position, tangent_1, tangent_2} do
  zero the buffer v for this attribute
  for all bones j do
    load G_j
    for all vertices p with index k influenced by bone j do
      v[k] ← v[k] + G_j (p_j^⌊l⌋ + Δl Δp_j^⌊l⌋)
    end for
  end for
end for
```

Depending on the lighting method used (Section 7.5), we may also need to transform per-vertex tangents. These are handled similarly, using a completely separate set of arrays, a separate transformation pass and separate output buffers. A potential area of future work is to determine

[2]Streaming SIMD Extensions, a set of single-instruction multiple-data instructions supporting four single-precision operations concurrently.

whether it is more efficient to build shared bone and count arrays and perform only a single pass, as is done in the GPU-based rendering methods. This has the potential to improve cache hit rates, since each bone index and transformation matrix will be used three times in a row.

7.4.2 Fragment program

Modern graphics cards have significantly more computational power than CPUs [NVIDIA, 2007]. We can take advantage of this by using two passes to transform the vertices. In the first pass, we map the vertices to pixels in an off-screen floating-point buffer. The animation-space positions are encoded into a set of textures and the transformations into another texture, and a fragment program is used to do the transformations. At the end of the first pass, we copy the data from the output image to a vertex buffer. The second pass sources the positions from this buffer. This improves throughput compared to CPU-based transformation, but there is a significant overhead in switching between the passes.

Packing the data into textures is complicated by the fact that different vertices may have different numbers of influences. In order to get optimal performance, we must consider the following goals:

(a) The amount of wasted space in the texture should be kept small. In particular, allocating the same amount of texture area per vertex (which would correspond to the maximum number of influences on any vertex) can easily double the required texture size. This will increase bandwidth requirements, degrade texture cache performance and reduce the number of vertices that can be stored in device memory.

(b) Dynamic branching on the GeForce 6 series is most effective when branches are spatially coherent, that is, when nearby fragments take the same branches [NVIDIA, 2005b].

We handle (b) by re-ordering the vertices in decreasing order of number of influences. The influences for each vertex are arranged in a vertical block within the texture. These blocks are packed into the texture sequentially, left-to-right then bottom-to-top. For reasons explained later, it is necessary for all the blocks in a row to have the same height, so a small number of vertices may be augmented with a zero influence into order to satisfy this. Figure 7.3 shows an example of this texture layout.

The wider the texture is, the more wasted space there is likely to be. We start with a width of 4, and successively double it until the packing produces a height of no more than 512. Since the half-precision floating-point type has only a 10-bit mantissa, this limit on the height is necessary to ensure that we can exactly address the centres of texels (which have half-integer coordinates).

This *influence texture* holds only the actual influences themselves (i.e., the elements of $\mathbf{p}$, not the information about which influence goes with which bone). This allows us to store the four real values of each influence in the RGBA channels of a floating-point texture. The corresponding bone data, indicating which bone applies to each value in the influence texture, is stored in a separate *bone texture*. This is a single-channel fixed-point texture, with the same organisation as

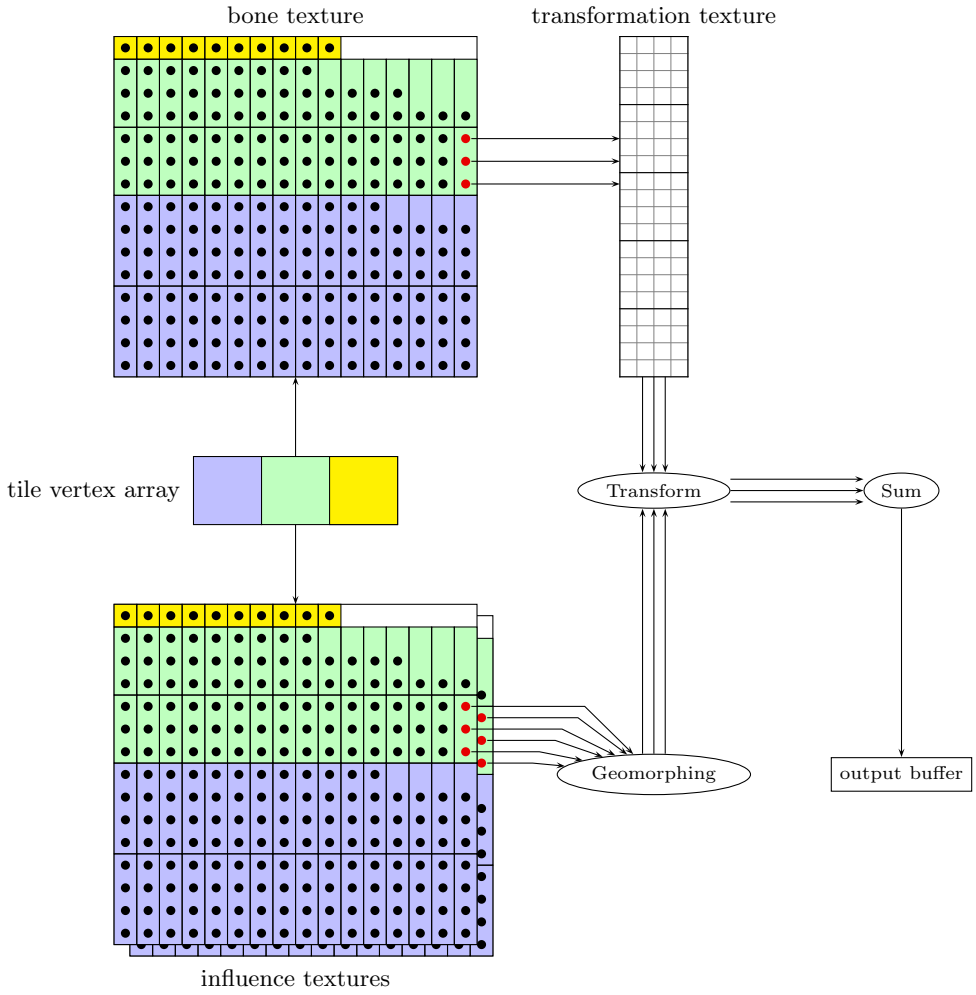


Figure 7.3: Data flow for fragment-program vertex processing. Each small rectangle in the bone and influence textures holds information about one vertex, with individual influences represented by dots. The red dots indicate a vertex being processed. Note that some zero influences (empty spaces) are introduced to keep the structure regular. The blue, green and yellow regions each correspond to a particular influence count, and to a single quadrilateral submitted to the OpenGL pipeline.

the influence texture. This texture is looked up in the fragment program, and the value is then used to address the *transformation* texture. Figure 7.3 illustrates the data flow in this algorithm.

We implement geomorphing by using two influence textures: the first holds data for $\mathbf{p}^{\lfloor l \rfloor}$ and the second holds data for $\Delta\mathbf{p}^{\lfloor l \rfloor}$. Since we have represented these values for each vertex on a common set of bones, we can share the bone texture as well as the texture layout. Similarly, when tangents are required for lighting, we use a total of six influence textures (two for position and two for each tangent) with a common bone texture and texture layout, adding zero influences where necessary to produce a common influence set. In this case the fragment program writes three outputs per fragment (position and two tangents) to separate buffers.

Finally, we must pass the relevant parameters to the fragment program. Here we take advantage of the regular structure of the textures. All rows with a particular height are rendered using a single quadrilateral (each colour in Figure 7.3 corresponds to one quadrilateral). The texture coordinates are set so that the fragment program receives the coordinates of the first and last influence of the vertex, making it possible to iterate over the influences.

7.4.3 Vertex texturing

The GeForce 6 series supports texture lookups from a vertex program, although there are significant restrictions and the performance is poor compared to the performance in a fragment program [Gerasimov et al., 2004]. We can use vertex texture lookups to perform the vertex transformations in a similar way to the previous technique (Section 7.4.2), but using only a single pass. This has the advantage of eliminating the state switching, but throughput suffers from the high latency of the texture lookups. A second disadvantage is that a vertex will be referenced multiple times during a single frame, and even with a hardware vertex cache, this may result in this expensive process being repeated many times.

While we do not expect this method to be optimal on a GeForce 6 series card, it has the most potential for future hardware such as the GeForce 8 series. For this newer hardware, a unified shader architecture makes vertex texturing just as capable as fragment texturing, and transform feedback [Blythe, 2006] makes it possible to perform the transformations as a pre-process.

The setup is similar to that of Section 7.4.2, but slightly simplified. We made the following modifications, illustrated in Figure 7.4:

- The vertices are not sorted by influence count — instead, they are placed in the order they are first encountered in the triangle list, to improve locality of access (this is also done for the CPU-based transformation routines).
- The influences on a vertex are arranged horizontally rather than vertically, and the texture is made as wide as possible rather than as narrow as possible. This is done because we no longer need the rectangular structure seen in Figure 7.3.
- The two values for geomorphing ($\mathbf{p}^{\lfloor l \rfloor}$ and $\Delta\mathbf{p}^{\lfloor l \rfloor}$) are stored one above the other in the

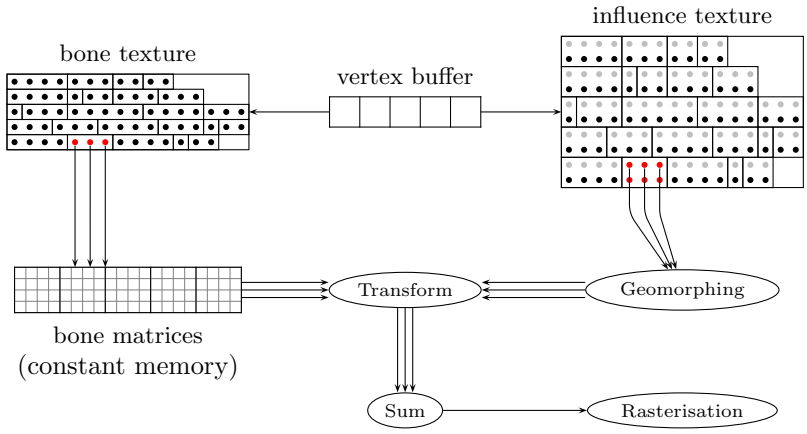


Figure 7.4: Texture layout for vertex-texturing vertex transformation. The rectangles represent positions and the dots represent individual influences. The red dots represent a vertex undergoing processing; the grey and black dots in the influence texture represent $\mathbf{p}^{\lfloor l \rfloor}$ and $\Delta\mathbf{p}^{\lfloor l \rfloor}$, respectively.

same texture, rather than in separate textures. The original intention was to use the texture filtering hardware to perform geomorphing with only one texture instruction. Unfortunately, GeForce 6 series hardware does not support texture filtering in a vertex program. The texture layout code was kept, but it would be equivalent to use pairs of textures as in Figure 7.3 (the performance may differ, but further investigation would be required to determine this). The grey and black dots in Figure 7.4 correspond to $\mathbf{p}^{\lfloor l \rfloor}$ and $\Delta\mathbf{p}^{\lfloor l \rfloor}$, respectively.

- The transformation matrices are placed into vertex program constants rather than a texture; despite the smaller constant bank (which limits this method to about 80–85 bones), this is more suitable than using a texture (as was done for the fragment program method) due to the existence of an address register as well as the poor performance of vertex texturing on GeForce 6 hardware. Newer cards (GeForce 8 series) have a larger constant bank, and also allow a buffer to be used for constants, making it possible to upload all the matrices with a single API call.

The indexing data (i.e., the texture coordinates for the bone and influence textures) are passed as vertex attributes.

7.5 Normals and lighting

Cohen et al. [1998] use high-fidelity lighting to disguise the loss of detail in simplified models, and today it is common practice to encode shading information into textures at a higher resolution than is practical or necessary for the geometric detail. We have used a simple lighting model, consisting of two directional diffuse light sources, but arbitrarily complex lighting methods can be used once a normal is available.

Blinn [1978] first proposed that surface features could be simulated by altering the normals of an object through a lookup table, without adding true geometric detail. He proposed encoding

a displacement to the surface, in the direction of the normal, for each point on a parametric surface. The perturbed normal is computed from the geometric normal and partial derivatives of the displacement function, evaluated using central differences.

While this encoding is memory-efficient (requiring only one scalar per sample), it is inconvenient for hardware application because it requires multiple table lookups for each normal calculation. Subsequent representations include:

Object-space normal maps [Peercy et al., 1997] All the normals for the object are expressed in a single coordinate system. This works well for static meshes, but there is insufficient information to adjust the normals during animation.

Tangent-space normal maps [Peercy et al., 1997] Instead of using a global coordinate system, a separate frame is defined for each vertex, using the geometric normal, a tangent and a binormal. These frames are interpolated across faces, to define a frame at every point on the surface. Normals are then encoded relative to this frame. Since this frame is defined relative to the local neighbourhood, it will animate correctly and hence so will the normals.

A variation of this technique is to exclude the coordinate corresponding to the geometric normal, and encode only the other two coordinates. Since the normal is known to have unit length, this third coordinate can be recovered provided that the geometric and perturbed normals are within 90° of each other.

Tangent-space normal maps (TSNMs) are not ideal for LOD (particularly continuous LOD), because the encoding is dependent on the geometry and hence a different map is required for each level of detail.

Frame maps Frame maps [Kajiya, 1985] are a concept developed for anisotropic lighting models. In addition to a normal, a surface tangent is stored for each surface sample point. The direction of the tangent indicates the orientation of the anisotropic surface; it might, for example, indicate the direction of a wood grain.

None of these schemes can represent surface detail that varies under animation. For example, areas of skin may be wrinkled in some poses and smooth in others, but if the geometry does not capture the wrinkles then there is no way to encode this behaviour into the maps. Furthermore, each has disadvantages for either LOD or animation.

We have implemented two approaches. The first uses TSNMs, with a separate texture for each discrete level of detail. When geomorphing between levels of detail, we simultaneously morph between the two applicable textures. The normals to encode are selected from an arbitrary user-designated pose. TSNMs work best if the tangents of neighbouring vertices are roughly aligned, since the interpolation between bases is done with linear rather than spherical interpolation. We achieve this by computing animation-space tangents at a vertex that align as closely as possible to the orientation of the parameter space at that vertex.

The second method uses a *tangent map*, each sample of which is a pair of animation-space surface tangents. The normal is computed by applying animation projection to the tangents and taking

their cross product, as for per-vertex normals, but on a sample-by-sample basis. We discuss in the next section how the tangent map is stored in memory for efficient rendering.

7.5.1 Rendering tangent maps

In order for lighting to look realistic, it is important that normals sampled from a texture are continuous. If the underlying textures are simply point-sampled (taking the nearest texture element), then each sample will produce a constant normal over an area, with discontinuities between the coverage of neighbouring samples. This gives a faceted appearance with very poor visual quality. Furthermore, since we pack data into textures in a space-saving manner, we cannot use OpenGL's texture filtering capabilities (for example, some textures hold indices into arrays). Bilinear filtering may be implemented in a fragment program by doing the computations once for each of the four surrounding texels, but this reduces performance by a factor of four. Instead, we do a normal-map generation pass immediately before the main pass for each rendered model. This generation pass converts the pose-independent tangent map into a pose-dependent object-space normal map, which is then used with hardware-accelerated filtering for the main rendering pass.

In Section 7.4.2 we encoded elements of animation space into textures for transformation in a fragment program. Unfortunately, that method relies upon reordering the elements to group those with the same number of influences together. Since the ordering of the tangent map samples is not arbitrary, this technique is not applicable to tangent maps, and a more complex packing scheme is required. We trade off a regular layout against maximum density by splitting the parameter space into equally-sized square *tiles*. Within each tile, we force every tangent-map sample to be influenced by the same number of bones, adding zero influences as before. Note that although we force the two tangents of a sample to have the same set of influences (allowing one set of bone indices to cover both tangents), we allow different samples within the same tile to have different influence sets as long as they are of the same size. An $n \times n$ tile with b influences is represented as a $bn \times n$ region in the input textures, which are two RGBA floating-point textures to hold the two tangents, and a single-channel fixed-point texture to hold the bone indices. Each sample within the tile is represented by a $b \times 1$ region of the textures. The algorithm is given in pseudo-code in Algorithm 3 and illustrated in Figure 7.5.

The remaining details are very similar to the fragment program for vertex transformation, with a separate quadrilateral rendered for each tile. We have found that tile sizes of around 4×4 to 16×16 work well in practice; for smaller sizes the number of quadrilaterals causes the vertex program to form a bottleneck, while larger sizes reduce the packing efficiency and hence increase demand on memory bandwidth and cache.

Compared to directly rendering the normals, generating a normal map has the additional advantage that each tangent map sample is transformed only once. On the other hand, every sample is always transformed, even if it is not visible in the scene (e.g., it may apply to a back face). This is particularly important for distant characters, since they will generate far fewer fragments and so should not require as large a normal map. We handle this by generating versions of the

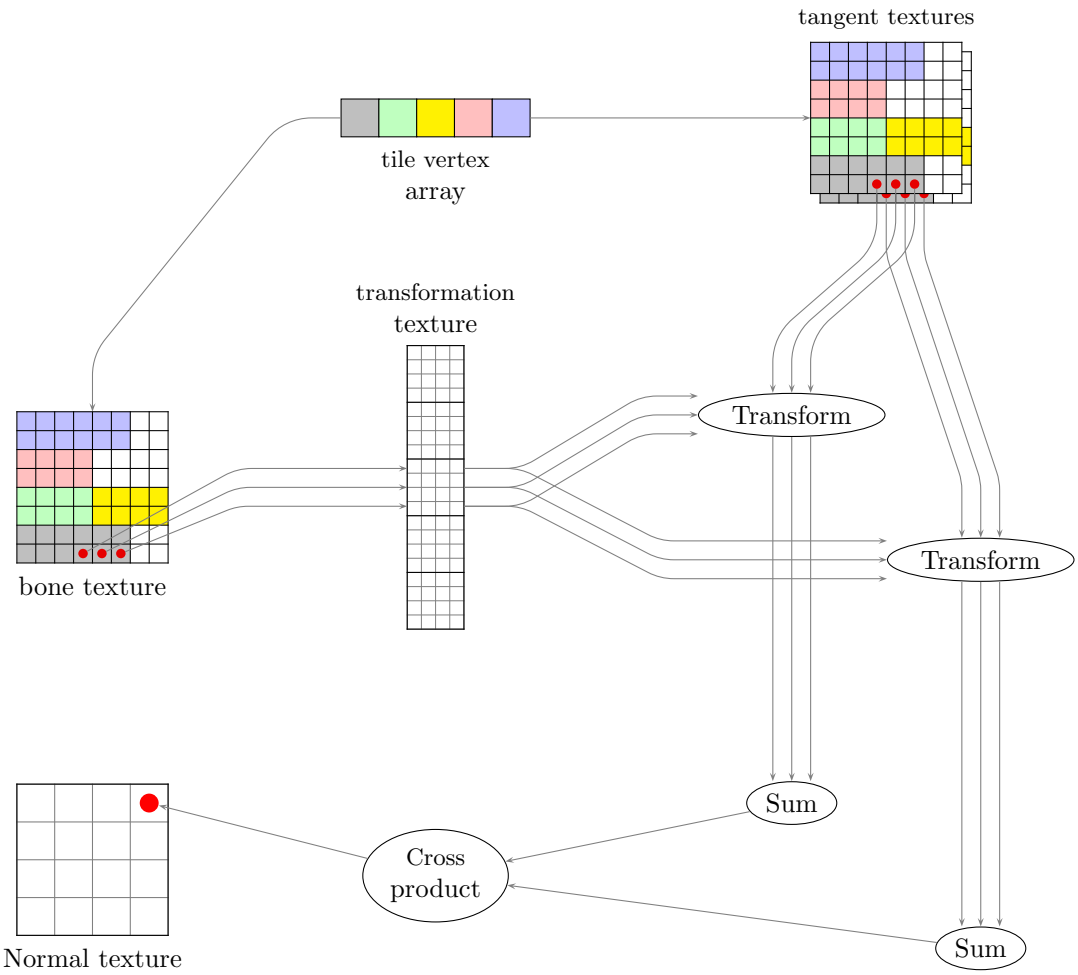


Figure 7.5: Illustration of the normal map generation process. Each colour indicates a 2×2 tile, and the red dots indicate a single vertex being processed.

Algorithm 3 Pseudo-code for generating a normal map

CPU:

```
for all tiles do
    render a rectangle
end for
```

Fragment program:

```
t_1 ← 0      //first tangent
t_2 ← 0      //second tangent
for s = s_first to s_last do
    j ← bone-texture(s, t)      //bone index
    G_j ← transform-texture({0, 1, 2}, j)      //transformation matrix
    for i = 1, 2 do
        p_j ← tangent-texture_i(s, t)
        t_i ← t_i + G_j p_j
    end for
end for
n ← t_1 × t_2
n ← n/‖n‖
```

tangent map at multiple resolutions (stepping in powers of two), and selecting the one with the appropriate resolution at normal-map generation time. This is similar to the idea of mipmaps in OpenGL, except that we select one level for the entire model rather than on a per-fragment basis. The minified tangent maps are produced by averaging blocks of 2×2 samples to produce a single sample in the reduced map, using the algorithm in Appendix A.1. Note that this level-of-detail selection is unrelated to the LOD described in Chapter 6 and Section 7.3.

7.6 Summary

Previous chapters have established many advantages of animation space, but without an efficient way to render animation-space models, its use would be limited to animation-space computations applied to SSD models. The biggest potential problem with animation space is that the larger number of per-vertex parameters, particularly when combined with geomorphing, may exceed hardware limitations. We have demonstrated several algorithms that are not subject to these hardware limits; in the next chapter, we will show that these methods perform well in practice. Each generation of video cards is also more generic and less limited than the previous one, so it is likely that this will become even less of an issue over time.

We also considered the problem of reproducing the original lighting on simplified models. We noted that an existing method, tangent-space normal maps, applies just as well to animation-space models, but does not capture dynamically changing features such as wrinkles. We introduced a new method, the tangent map, which stores tangents in animation space. Tangent maps can capture dynamically changing features, but they are more expensive to store and render.

Chapter 8

Results and discussion

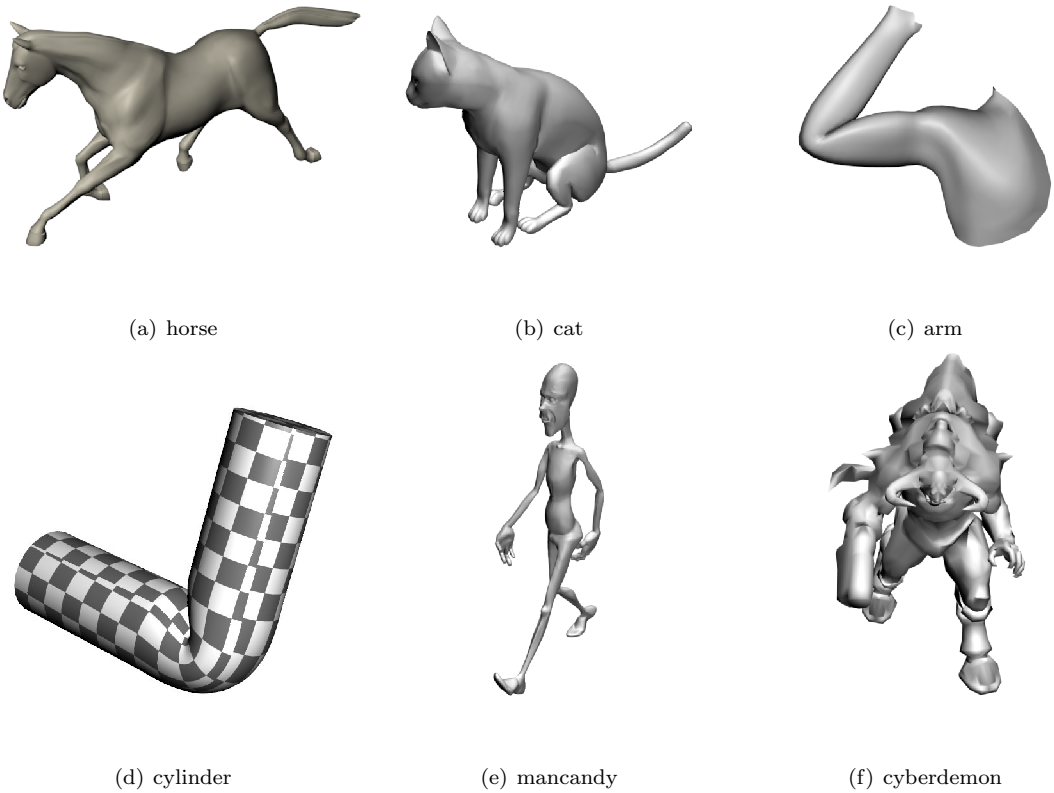
We now present our evaluation of the methods described in this thesis. We have used a combination of computed quality metrics, user evaluations, and benchmarking. Section 8.1 covers the hardware and software setup used for evaluations. Section 8.2 describes the test models used, and in particular how we applied our fitting procedure (Section 4.1) to produce realistic models. Section 8.3 provides an external validation of animation space as a modelling tool, by comparing the quality of fitted models based on the skeletal subspace deformation (SSD), animation space, and multi-weighted enveloping (MWE) frameworks.

Our applications of animation space as an algorithmic framework, namely parametrisation and level-of-detail, are evaluated in Sections 8.4 and 8.5 respectively. In Section 8.6 we evaluate our renderer, comparing our methods for vertex transformation and lighting, and also compare performance to SSD and MWE.

8.1 Test setup

We implemented the algorithms discussed throughout the thesis using C++, compiled with GNU GCC 4.1 and executed under Gentoo GNU/Linux. The hardware consisted of an Athlon XP 2800+ (clock speed of 2.083GHz) with 512MB of DDR400 RAM. With one exception discussed later, we did not use any assembly code or SIMD (Single Instruction Multiple Data) extensions for the CPU. In particular, it should be noted that Meschach [Stewart and Leyk, 1994], the linear algebra library that we used, has a straightforward C implementation of low-level matrix operations and does not benefit from SIMD extensions, and it also does not perform block-based computations for cache efficiency as is done by ATLAS [Whaley and Petitet, 2005].

For all operations except rendering we used double precision. While using single precision may improve performance (particularly when combined with SIMD instructions), our experience was that even with double precision it was still necessary to compensate for numerical errors in calculations. This is not surprising, given the well-known problems of maintaining accuracy in matrix



(a) horse (b) cat (c) arm

(d) cylinder (e) mancandy (f) cyberdemon

Figure 8.1: The models used in the experiments. Horse, cat, arm and cylinder are fits to sample poses. Mancandy is a sample Blender model with two iterations of Catmull-Clark subdivision. Cyberdemon is a character from Doom III (copyright Id Software and used with permission).

computations [Golub and Van Loan, 1996].

8.2 Models and fitting

We used six models for our experiments. These are shown in Figure 8.1, while Table 8.1 summarises their properties. Cyberdemon is a character from Doom III and is used unaltered. Mancandy is a test model for Blender; the Blender file specifies two iterations of Catmull-Clark subdivision, which we have applied as a pre-process. This subdivision gives mancandy a high vertex count and average number of influences per vertex, which is useful in testing the scalability of our system. The remaining models were produced using the fitting process described in Chapter 4.1. For horse, cat and arm we downloaded reference poses from the Internet. These reference poses had vertices in correspondence, but no skeletons, so we fit skeletons as described in Section 4.1.4. It is not known exactly what method was used to produce the downloaded models, but we have no reason to believe that they should be unusually favourable to an animation-space approximation, and in particular they exhibit none of the flaws that we would expect if they were produced with skeletal subspace deformation.

For the arm model, we found that the rest pose selected (an example pose with the elbow bent) led

Table 8.1: Properties of the models used in the experiments. *Influences* is the number of bones influencing each vertex. *Holes* is the number of boundary loops and not the genus.

Model	**Vertices**	**Bones**	**Influences**		**Holes**
			Max	**Avg**	
Horse	8431	25	6	2.6	0
Cat	7207	26	5	2.2	0
Arm	1600	4	3	2.1	2
Cylinder	2426	2	2	2.0	0
Mancandy	42654	96	12	2.4	12
Cyberdemon	2282	65	6	1.5	4

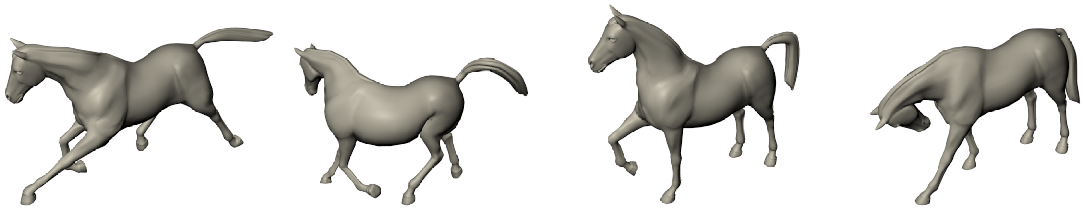
to unnatural-looking shadows when using tangent-space normal maps, so we replaced it with one in which the elbow was only half bent. For all the other models, we used the provided references: neutral standing poses for horse, cat, mancandy and cyberdemon, and a straight cylinder.

The reference models for the horse consists of 11 poses, covering a range of head positions and standing, walking and running poses. We manually marked influence sets, using the vertices with only one influence to determine the skeleton, then further expanded the influence sets to obtain a better fit. We found that using $\delta = 0.005$ in Equation (4.4) produced good results. The iterative least-squares solver converged in 4185 iterations, and the whole fitting process took 524 seconds (spent almost entirely in the least-squares solver). In fitting the 8431 vertices to the 11 models, slightly less than 40MB of resident memory was used.

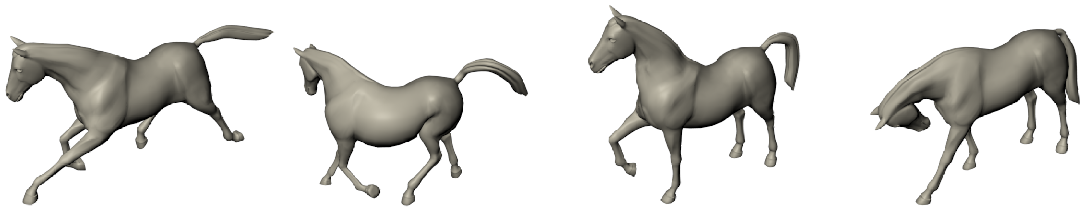
Figure 8.2(a) shows four of the reference poses for the horse, while Figure 8.2(b) shows our fitted model in the same four poses. It is clear that we have obtained visually identical results. In addition to the reference poses, we also obtained a 48-frame animation of the horse galloping (these poses were not used in fitting the model). We used our skeleton extraction to determine the ideal skeletal configurations corresponding to these poses, and thus produced a gallop animation from our fitted model. Figure 8.2(c) shows four frames from this animation, and it is clear that the model generalises well.

In Figure 8.3 we show the effects of regularisation, using a pose from the gallop sequence (i.e., not one used for the fitting). Figure 8.3(a) has no regularisation, and as a result there are prominent artefacts. A large bump appears on the top of the front leg, and there is a shear in the front of the neck at the boundary between two influence sets. Figure 8.3(b) shows vertex regularisation (Equation 4.3) with $\lambda = 0.02$. Although the artefacts are reduced, they are still present, and there are even new artefacts: the back-right leg is crooked, and there is a black spot on the tail. Figure 8.3(c) is the final, edge-regularised model we used ($\delta = 0.005$); it has minimal protrusion from the front-left leg, no shearing of the neck, and the tail and back-right leg are smooth. The edge regularisation does, however, cause the horse to shrink slightly, as the edges seek to become as short as possible to minimise the regularisation term; as a result, the overall geometric error is actually slightly higher in Figure 8.3(c) than in 8.3(b).

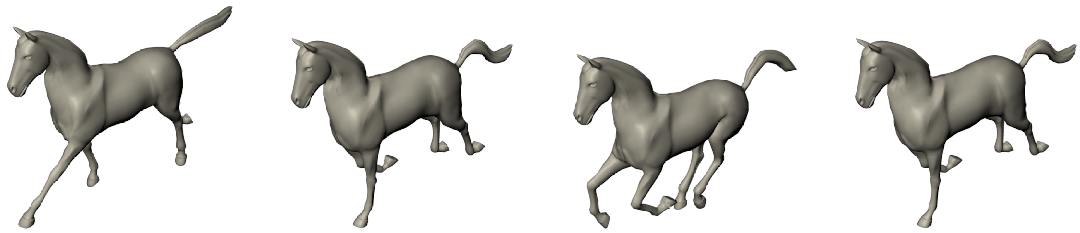
The process for the cat was quite similar, the significant differences being that only ten poses were available (and no separate animation equivalent to the gallop sequence), and we used $\delta = 0.01$.



(a) 4 of the 11 examples meshes used to create the model

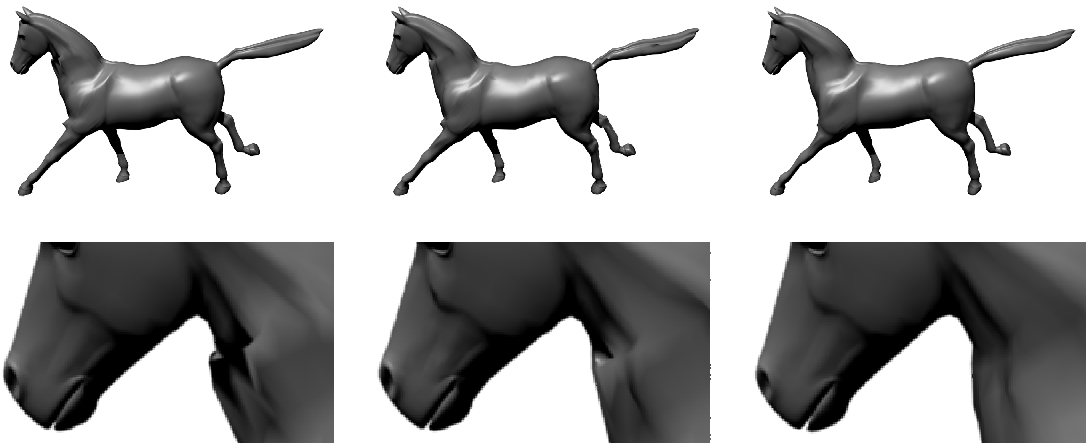


(b) the fitted model in the same four poses



(c) new poses generated from the fitted model

Figure 8.2: Horse model created from 11 example meshes



(a) No regularisation

(b) Vertex regularisation, $\lambda = 0.02$

(c) Edge regularisation, $\delta = 0.005$

Figure 8.3: Effect of different types of regularisation on fitting. The second row shows closeups of the neck.

The arm was more challenging, as only four reference poses (Figure 8.4(a)) were available. The influence sets we used are shown in Figure 8.4(e): the top of the upper arm is included in the influence set for the forearm, because that muscle bulges depending on the angle of the elbow joint. We found that $\delta = 0.02$ produced reasonable results, and that the quality was far more sensitive to the choice of δ than for the other models. Smaller values tend to allow discontinuities, while larger values cause volume loss in convex areas (such as the muscles) as the edge lengths are minimised. The iterative solver converged in 554 iterations, and the entire fitting process took 8 seconds. The results for the original poses can be seen in Figure 8.4(b), and other poses are shown in Figures 8.4(c) and (d).

Finally, the cylinder is an artificial example, designed to show the advantages of animation space. We procedurally generated 40 poses of the cylinder as it bent through 90°, maintaining a uniform cross-section, then fitted an animation-space model to this ideal. Although it is not used further in our experiments, we also fitted a variant in which the cylinder is twisted (like a wrist) rather than bent (like an elbow). Figure 8.5 compares these models to the equivalent SSD models, showing that animation-space is better able to preserve the shape of the cylinder as it is bent.

In the case of the cat and the arm, no existing animations were available. We used Blender to create short animations by positioning the skeleton in key frames and using Blender's animation system to interpolate between them.

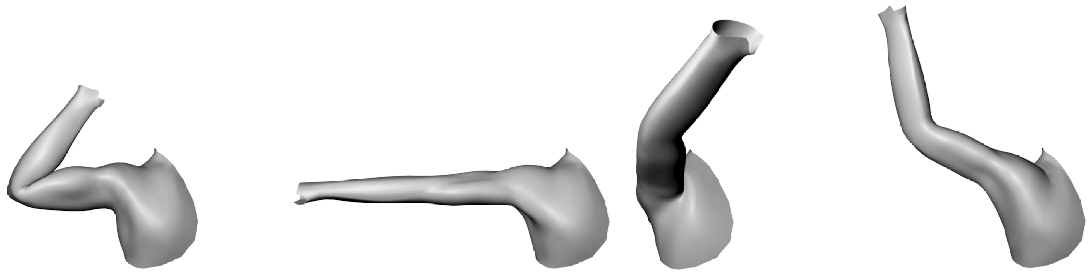
8.3 Comparison to SSD and MWE

A qualitative and quantitative comparison of skeletal subspace deformation (SSD), multi-weight enveloping (MWE) and animation space was undertaken by two fourth-year students supervised by the author. Their findings are available as a separate technical report [Jacka et al., 2007], but we reproduce their results here as they pertain to animation space.

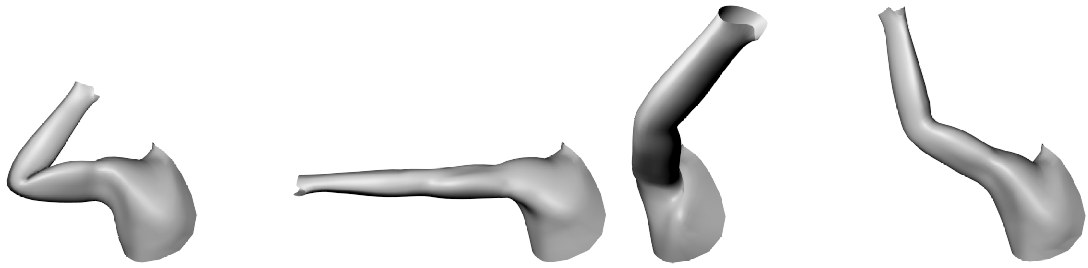
Jacka et al. implemented a fully automated skeleton extraction, based on the work of Anguelov et al. [2004b] and James and Twigg [2005], and also produced an independent implementation of our fitting method, using Equation (4.3) on page 35: that is, they applied vertex regularisation, but no edge regularisation. This allowed the implementation to execute significantly faster and with less memory use, as without edge regularisation it is possible to solve for vertices independently rather than as a single large optimisation. They found that the higher quality of the extracted skeletons made the artefacts shown in Figure 8.3(b) far less significant.

For comparison, they implemented similar fitting techniques for SSD and MWE, as described by James and Twigg [2005] and Wang and Phillips [2002] respectively. The fits were compared objectively for geometric and normal deviation from an ideal (either one of the training poses, or a non-training pose for which an ideal was available), and subjectively by obtaining ratings from human subjects.

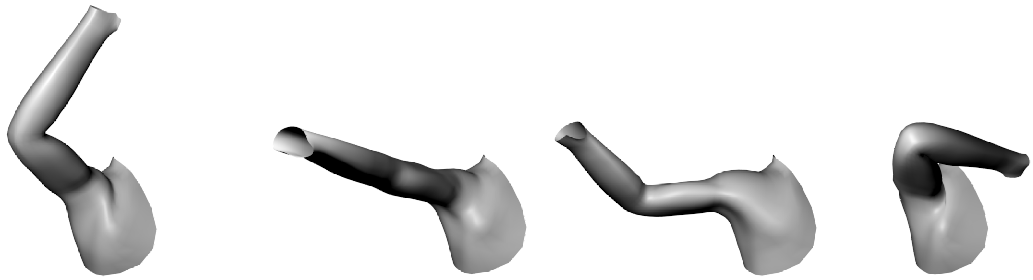
The models used for testing are the "horse" and "arm" models shown in Figure 8.1, and the "twist" and "camel" models shown in the bottom two rows of Figure 8.7. It should be noted that



(a) example meshes used to create the model

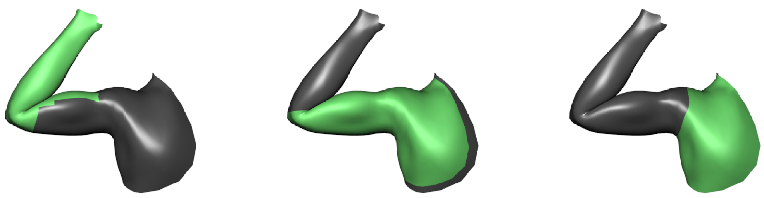


(b) the fitted model in the same poses



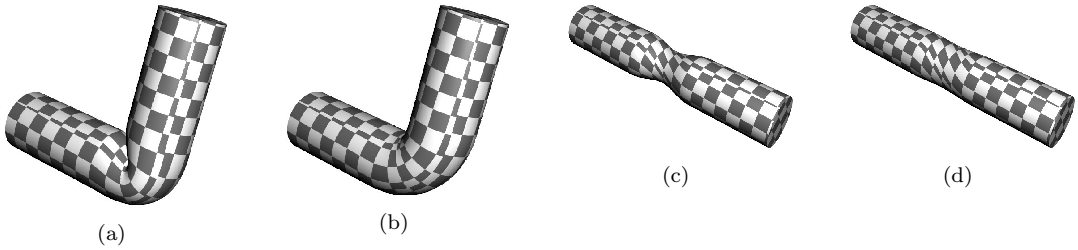
(c) new poses generated from the fitted model

(d) large extrapolation



(e) influence sets used

Figure 8.4: Arm model, generated from the four poses in (a)



(a) (b) (c) (d)

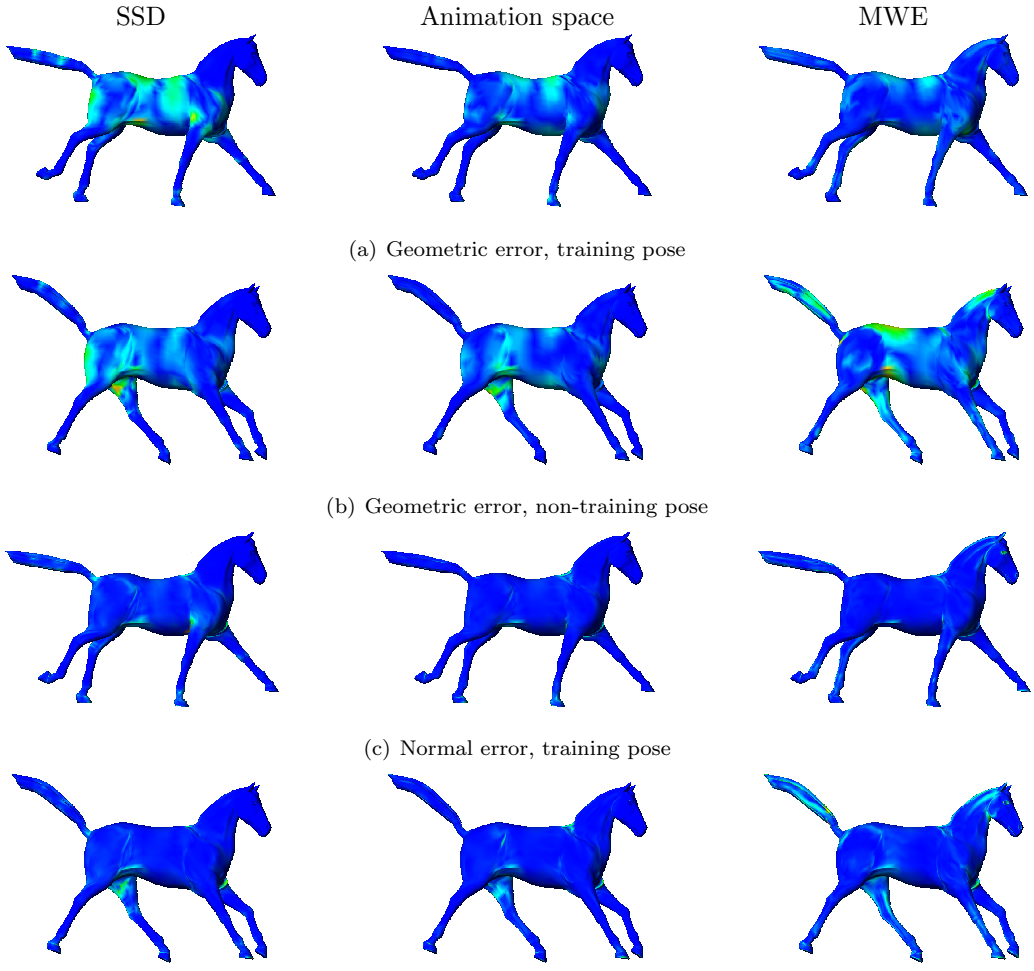
Figure 8.5: (a) and (c): Cylinder animated with SSD, showing the collapsing elbow and candy wrapper wrapper effects. (b) and (d): Cylinders modelled in animation space, relatively free of these effects.

the non-training poses of the camel were produced from those of the horse by deformation transfer [Sumner and Popović, 2004], and hence are not expected to provide significant further insight.

8.3.1 Error metrics

The fitted models were compared to the reference models, both in the training poses, and in new poses for which a reference was available but which were not used in fitting. This comparison was done using Polymeco [Silva et al., 2005], a tool that computes geometric and normal deviations between meshes. Figure 8.6 provides a visual comparison of these errors in the case of the horse. The geometric errors are the most informative: when examining a pose used in the training set (first row), SSD produces large errors, as it is simply not general enough to represent the animation. In contrast, the errors for both animation space and MWE are quite small, and relatively little benefit is gained from the extra degrees of freedom in MWE. When considering a new pose, animation space still out-performs SSD, but the extra degrees of freedom in MWE have clearly led to over-fitting, causing it to generalise quite poorly to the new pose. Figure 8.7 shows the geometric deviations of a single (non-training) pose for the other three models considered, which show the same trends: under-fitting in SSD for the twist model (middle row), and over-fitting in MWE for the arm and camel models (top and bottom). The data sets each have between 3 and 10 training poses, which may account for the surprisingly poor fits for MWE, in contrast with the superior results obtained by Wang and Phillips [2002] using hundreds of training poses. Such large training sets are practical when they are generated from a non-realtime animation system, but are infeasible to design by hand.

Table 8.2 summarises the geometric and normal deviations for the four models across all available poses. In every metric except maximum normal error, animation space has the least deviation in every case. The large maximum normal deviations in three of the models suggests that the discontinuity/shearing effect discussed in Section 4.1.1 may be present: this may cause one piece of the mesh to slide very slightly over another, introducing a minimal geometric error but causing a small number of triangles to flip over.



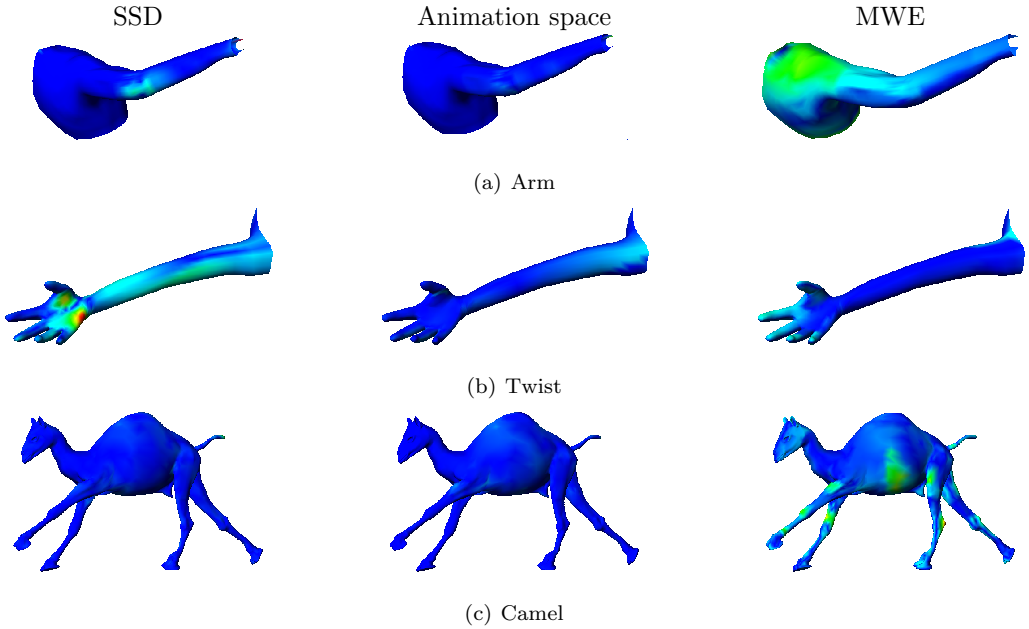
(a) Geometric error, training pose

(b) Geometric error, non-training pose

(c) Normal error, training pose

(d) Normal error, non-training pose

Figure 8.6: Geometric and normal errors for the horse model. A rainbow colour scale is used, with blue representing no error and red representing large error. The scale is consistent within each row, but varies between rows.



(a) Arm

(b) Twist

(c) Camel

Figure 8.7: Geometric error in the arm, twist and camel models. The colouring is as for Figure 8.6, and once again the scale is consistent within each row but varies between rows. The poses shown were not part of the training sets.

Table 8.2: Mean and maximum geometric and normal errors. The mean and maximum values are taken across all the available poses, including non-training poses. The best result in each section is shown in bold.

Model	**Framework**	**Mean Geom Error** ($\times 10^{-3}$)	**Max Geom Error** ($\times 10^{-3}$)	**Mean Normal Error** (°)	**Max Normal Error** (°)
Horse	SSD	1.17	68.7	2.43	**177.3**
	AS	**0.73**	**53.3**	**2.41**	179.1
	MWE	1.43	99.1	4.29	178.8
Camel	SSD	1.22	78.5	3.18	180.0
	AS	**0.31**	**50.2**	**2.29**	180.0
	MWE	6.13	109.5	7.66	180.0
Arm	SSD	26.9	961.	1.25	55.6
	AS	**16.9**	**475.**	**0.68**	**21.6**
	MWE	229.9	535.	1.38	22.3
Twist	SSD	5.20	42.1	6.23	**85.**
	AS	**2.18**	**26.0**	**3.42**	166.
	MWE	18.24	126.7	14.46	179.

8.3.2 User testing

The perceptual quality of the fits was evaluated by 19 test subjects. Each subject was shown several pairs of animations, and asked to provide a rating of how similar the animations in each pair were (on a 1–10 scale from least similar to most similar). In each case, one animation was produced from the reference model (generally including poses not used for training), and the other from a model fitted to either the SSD, animation space or MWE framework.

The mean ratings for SSD, animation space and MWE were respectively 7.88, 8.88 and 6.29, and t-tests confirmed that the difference between each pair of means was highly statistically significant ($p < 10^{-6}$). This indicates that animation space was able to produce the best models, confirming the objective tests. Further details are available in the technical report [Jacka et al., 2007].

8.4 Parametrisation

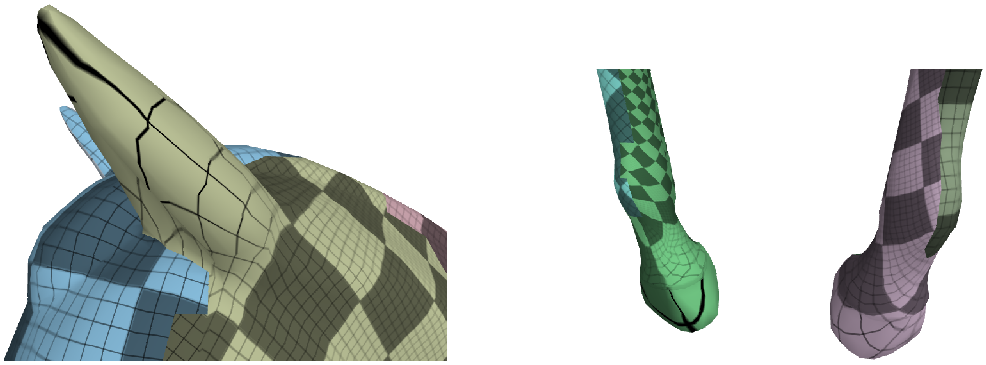
The parametrisation method itself is not the focus of this thesis, and so in evaluating it, we have not undertaken a methodical analysis of the various tuning parameters. Rather, we used ad-hoc trial and error to find values that gave the desired effect. Except where otherwise stated, we have used the following parameters:

- A compactness term weight ρ in Equation (5.4) of 0.2.
- Geometric edge straightening (Section 5.2.3): if the first attempt at straightening increases the segmentation metric by a factor of more than 1.2, the constrained form of straightening is used.
- Parametric edge straightening (Section 5.3): if forcing the chart boundaries to follow straight lines in parametric space causes the L_2 stretch metric to increase by a factor of more than 4.0, then we introduce extra corners.

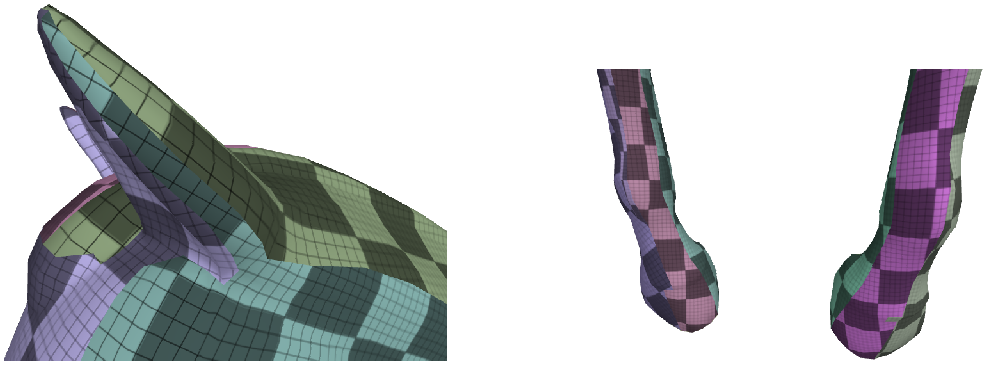
8.4.1 Segmentation

In Section 5.2, we introduced a new segmentation metric, which we now show is an improvement on the "mean-squared" metric used by Sander et al. [2001]. Figure 8.8 shows both methods applied to the horse model, which is then parametrised. We made the following choices for both cases:

- In the interests of comparability, no compactness term was used, since they take different forms for the two metrics (ours uses projected area rather than surface area).
- We disabled both edge straightening mechanisms, since their effect on quality is quite variable and unpredictable.
- We performed the segmentation calculations in 3D rather than animation space, because we have not adapted the mean-squared metric for animation space.



(a) Mean-squared metric [Sander et al., 2001]



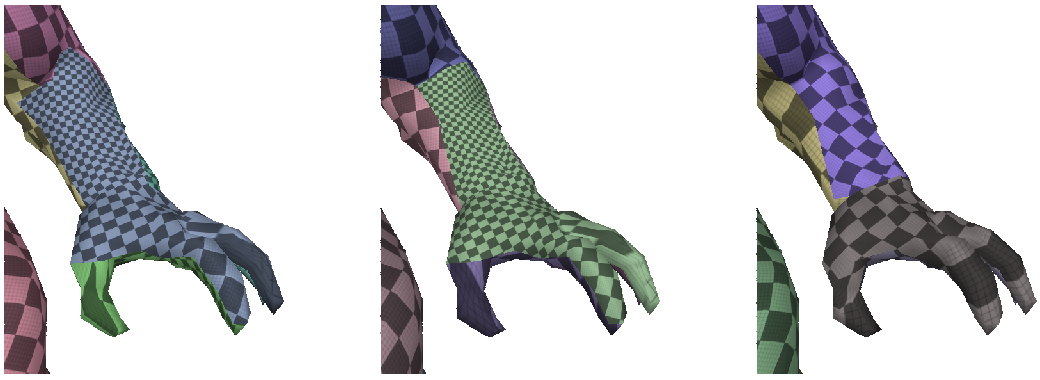
(b) Our area ratio metric

Figure 8.8: Comparison of mesh segmentation metrics, showing the head and legs of the horse. The mean-squared metric causes the left ear and front feet to be completely wrapped, leading to poor parametrisations as shown by the wide variation in scales.

From the figure, it is clear that by failing to consider orientation, the mean-squared metric is not able to handle flat areas such as the ear of the horse. Narrow limbs (such the front legs in the example) are also problematic for the mean-square metric, whereas our area-ratio metric is able to avoid creating sock-like charts.

Unfortunately, while the segmentation metric is an improvement on prior work, it is still lacking in some respects. A common failure is that it allows a sock to form, provided that it is attached to a larger planar region. Since the sock is small, it contributes little to either the numerator or denominator in Equation (5.4), and so there is only a small penalty. However, the detrimental effect of the sock on the parametrisation is far greater than is suggested by the metric. Figure 8.9 shows an example of such a problem: one finger of the cyberdemon is completely enveloped by the chart covering the top of the arm. When this chart is parametrised, the finger-tip takes a disproportionately small area in parameter space, forcing the rest of the chart to be over-sampled.

In order to ensure that these problems did not adversely affect our results, we chose to manually segment the models. Common 3D modelling packages support this task (we used Blender), and so it is not overly onerous. Even for the more complicated models (cyberdemon and mancandy),



(a) automatic, without geometric edge straightening

(b) automatic, with geometric edge straightening

(c) manual

Figure 8.9: Top-down view of cyberdemon, segmented manually and automatically, parametrised, and textured with a chequerboard pattern. The automatic methods completely enclose one of the fingers, which causes the disparate scale on the arm. The effect of automatic straightening can be seen on the top of the head.

segmentation required no more than a few hours, with little prior experience required. While reliable automatic segmentation would of course be preferable, this is a small investment in time compared to that needed to model, rig, skin and animate a detailed character.

Table 8.3 shows the value of the segmentation metric and the L_2 stretch for a parametrisation of this segmentation, all computed in animation space. In most cases, the manual segmentation has a higher area-ratio metric, but a lower L_2 stretch. This is most likely because in designing the manual segmentation, we aimed for developable charts, rather than planar ones. As an example, we generally used two charts per arm or leg (excluding the hand or foot), since a hemicylinder is perfectly developable despite being non-planar.

8.4.2 Flattening

One of the motivations for developing animation-space metrics was that they would allow parametrisation to be performed in animation space, and thereby improve the overall quality of the parametrisation across all poses. We now present results to support this. For the same reasons as supplied for segmentation, we did not attempt to straighten the chart boundaries in parameter space. For each model, we have performed parametrisation using three methods:

1. using only the rest pose, with computations in 3D;
2. using animation space, with the uniform rotations, fixed translations statistical model; that is, using only information from the skeletal structure and no knowledge of the intended animation;
3. using animation space, with expectations estimated from an animation.

Table 8.3: Comparison of manual and automatic segmentation. The methods are A = automatic, AS = automatic with straightening, M = manual. The best result in each block is shown in bold.The automatic methods failed to produce a valid parametrisation for mancandy due to foldovers, and applying straightening caused some triangles to become degenerate (horse) or almost degenerate (cat).

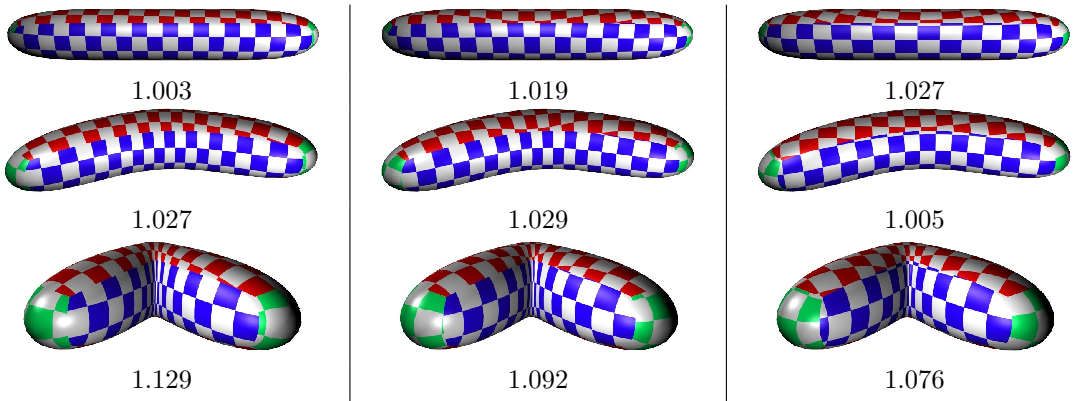
Model	**Charts**	**Method**	**Metric**	L_2 **stretch**
Horse	30	A	**2.15**	5.27
		AS	2.23	∞
		M	2.62	**1.04**
Cat	36	A	**1.95**	**1.08**
		AS	1.96	33.43
		M	2.63	1.10
Arm	6	A	1.96	1.00
		AS	**1.93**	1.00
		M	2.29	1.00
Cylinder	6	A	4.03	1.10
		AS	2.91	1.16
		M	**2.84**	**1.00**
Mancandy	69	M	3.60	1.13
Cyberdemon	67	A	**1.91**	1.22
		AS	2.05	1.27
		M	2.44	**1.17**

Table 8.4 shows the resulting average L_2 stretch. Since our aim is to validate animation space as a meaningful space in which to perform measurements, we do not show animation-space L_2 stretch in the table, but rather the average of 3D-space L_2 stretch across the frames of the animation. In a few instances, using the uniform rotations, fixed translation model causes the chart boundaries to self-intersect in parameter space. Normally, we would split such charts to eliminate the self-intersection, but this would change the segmentation and bias the comparison, so we have left these charts as is. The affected results are shown in brackets in the table.

The metric based only on the skeleton performs poorly, as it assumes that joints have total freedom of motion. In contrast, the 3D model assumes that joints have zero freedom of motion, which is

Table 8.4: Comparison of metrics for flattening. The column headings refer to the information that was used to define the distance metric. The values are the average L_2 stretch metric for the animation, computed in 3D space. The best result in each row is shown in bold. Values in brackets indicate that some charts had self-intersecting boundaries: in the interests of using the same segmentation for all the results, we did not split these charts.

Model	L_2 **stretch**			**Time (s)**	
	None (3D)	**Skeleton**	**Animation**	**3D**	**AS**
Horse	1.096	(1.505)	**1.056**	15.0	19.8
Cat	1.249	1.740	**1.147**	8.6	10.7
Arm	1.044	1.081	**1.025**	1.7	2.2
Cylinder	1.015	1.035	**1.008**	2.9	3.4
Mancandy	**1.088**	(2.392)	1.285	119.0	146.8
Cyberdemon	1.210	(2.187)	**1.182**	1.9	2.4



(a) No dynamic information (b) Skeleton only (c) Skeleton and animation

Figure 8.10: Three parametrisations of a tube, in three different poses. (a) Only the top pose is considered. (b) The skeleton is used, but no information about the animation is considered. (c) Statistics about the animation are used. The numbers below each figure indicate the L_2 texture stretch. The stretch is most visible on the red chart in the area of the joint.

closer to the truth for small ranges of motion. Extracting the freedom of motion (in the form of expectations) from the animation gives the best results in all the models except mancandy. In the case of mancandy, the animation-space method produces distortions around the ankles. We have not yet determined why this is the case, but it is most likely due to an interaction between translations and rotations that is not captured by our statistical model of independent translation and rotation components.[1]

Figure 8.10 shows an example of flattening in 3D and in animation space, using an artificial model of a tube to highlight the differences. The flattening in 3D space is computed from the pose shown in the top row. In this pose the parametrisation is almost distortion-free. However, because this parametrisation is only optimised for this pose, the distortion is worse in other poses.

For this simple example, it is clear that the choice of "rest" pose (i.e., the pose in which to perform flattening) was poor, and that better results could have been obtained from the intermediate pose. In general, however, it is far from obvious how best to select a pose for parametrisation, or even that a single pose can ever be sufficient, while animation-space simplification frees the user from having to make such choices.

We have shown that in most cases, animation space is the appropriate space for flattening, but if it is orders of magnitude slower it may not be justified. Table 8.4 also shows the time required to parametrise a model in 3D space and in animation space; in the worse case (horse) the animation-space method takes only 32% longer. It should be noted that we used the same code in both cases; code that is written specifically for the 3D case is likely to run faster. Nevertheless, parametrisation in animation space is unlikely to be an order of magnitude slower than in 3D, and since it is a once-off process, the potential for an improved parametrisation should justify its use.

[1] Mancandy has quite large and variable translation components in some of the joints, which is anatomically implausible.

Table 8.5: L_2 stretch with and without parametric edge straightening. We used manual segmentations and performed computations (including the computation of L_2 stretch) in animation space.

Model	**Unstraightened**	**Straightened**
Horse	1.040	1.354
Cat	1.101	1.210
Arm	1.003	1.105
Cylinder	1.000	1.016
Mancandy	1.138	1.340
Cyberdemon	1.172	1.510

8.4.3 Edge straightening

In order to use our parametrisations in a progressive mesh, it was necessary for the edges to be straight in parameter space (this is a standard problem, and in no way related to animation space). We implemented an ad-hoc scheme which has a few shortcomings. In particular, our choice of which points to label as corners was rather arbitrary, and it was not unusual for a large number of corners to be added before an acceptable parametrisation was found. Manual segmentation helped alleviate some of the deficiencies, because we were able to see problematic cases and add extra charts in order to force corners at specific points. We were also able to make decisions based on global information, such as placing boundaries in areas where the polygonisation made it possible to produce a straight boundary, which was not possible for the greedy, bottom-up algorithm. With manual segmentation, we are satisfied that the edge straightening was good enough to obtain meaningful LOD results.

Table 8.5 compares the L_2 stretch for parametrisations computed with and without edge straightening. The straightening clearly introduces additional stretch, but because of the quality of the manual segmentation, the stretch remains reasonable. It is also worth comparing the *Unstraightened* column in Table 8.5 with the *Animation* column in Table 8.4 — they are, respectively, the average 3D L_2 stretch and the animation-space L_2 stretch for the same parametrisation. While they are not the same, they are similar, although in the case of mancandy there is a large difference for the same reason that the animation-space parametrisation performed poorly in the first place.

8.4.4 Packing

We added no novelty to the problem of packing charts into a single texture atlas, simply using the alternating left-to-right and right-to-left algorithm of Sander et al. [2001]. Figure 8.11 shows two examples of packings produced by this algorithm.

Table 8.6 summarises the final parametrisations produced for each of the models, including the segmentation and parametrisation. This includes parametric edge straightening, with computations performed in animation space using estimated expectations. The running time is dominated

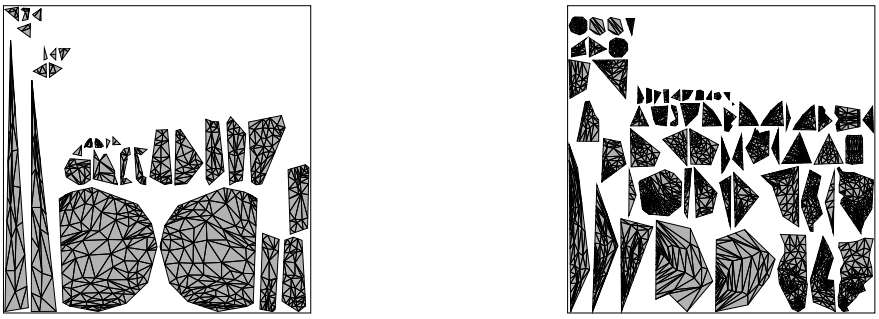


Figure 8.11: Sample parametrisations of the horse (left) and cyberdemon (right), illustrating the packing and edge straightening. Note that in the case of the horse, we show the edges of a simplified version (1000 faces) to ensure the triangles are large enough to be visible.

Table 8.6: Summary of texture efficiency. Packing efficiency is the fraction of the parametric domain that is occupied by the parametrisation. Texture efficiency measures what fraction of the parametric space has been wasted, either by packing or by variable sampling; it is equal to the packing efficiency over the square of L_2 stretch.

Model	Charts	L_2 stretch	Packing Efficiency	Texture Efficiency	Time (s)
Horse	30	1.354	0.386	0.211	47.7
Cat	36	1.210	0.467	0.319	35.4
Arm	6	1.105	0.486	0.398	5.0
Cylinder	6	1.016	0.613	0.593	7.3
Mancandy	62	1.340	0.372	0.207	2681.2
Cyberdemon	63	1.510	0.370	0.162	12.7

by the flattening; the differences in times between Tables 8.4 and 8.6, particularly in the case of mancandy, are due to the edge straightening process that causes some charts to be flattened many times as corners are successively introduced.

Texture efficiency is mostly in the 20%–40% range. For the cyberdemon model it is worse, due to the higher L_2 stretch. The cylinder is a simple and artificial model, which leads to a tighter packing and hence better texture efficiency.

8.5 Level of detail

We improved the level of detail of animated models in two ways: by using the $L_{2,2}$ metric within the framework of appearance preserving simplification (APS), and by removing bone influences on vertices. We compare animation-space LOD with and without influence simplification, and we consider both the standard and memoryless forms of APS.

Figures 8.12 and 8.13 show the errors introduced by the various LOD methods. In each case, the RMS error refers to the square root of the mean squared error, with the mean taken across both space and time. The error is measured between corresponding points in parameter space, for the same reason that APS defines the error metric in this way (namely, to account for texture sliding).

We obtained the measurements by rendering geometry images [Gu et al., 2002] at a resolution of 2048×2048, then taking the difference between geometry images for full-detail and approximated models. A side-effect of this measurement methodology is that the spatial mean is weighted by parametric area, rather than geometric area, but Table 8.5 indicates that these are similar. The models were scaled to fit in an axis-aligned bounding cube of unit size, so that the absolute errors are roughly comparable between graphs.

Figure 8.12 shows the absolute errors on a logarithmic scale. The graphs all have similar shape, and suggest that simplification has three phases[2]. In the first phase (to the right on the graphs), the superfluous vertices and influences are eliminated, but these are quickly exhausted and the logarithm of the error climbs rapidly, although the errors are still negligible. In the middle phase, curves are gradually flattened out and surface detail is lost, while the error climbs smoothly. Once all the detail has been eliminated, the final phase is to remove large-scale structure such as ears and the shape of the face until no further collapses are possible; during this phase the error again increases rapidly.

Figure 8.13 shows relative errors on a linear scale, where it easier to compare different techniques. It is clear that animation-space simplification, influence simplification, and standard APS are a poor combination (green). In a high-dimensional space such as animation space, the axis-aligned bounding boxes used in standard APS are an extremely conservative approximation. The result is that in some cases, an edge collapse would be the best choice but the inaccuracy in the error metric leads to an influence simplification being used instead. The problem is exacerbated by the fact that we have made no attempt to choose sensible axes in normalised animation space. To a lesser extent, this shortcoming of standard APS in animation space can also be seen in a comparison of the two instances of animation-space simplification without influence simplification: in every case the memoryless form (blue) performs better than the standard form (pink). As there is no clear indication that standard APS is an improvement on memoryless APS in rest-pose simplification (cyan versus black), and as memoryless simplification is simpler, faster, and more memory-efficient, we will restrict further discussion to memoryless simplification.

In every case, animation-space simplification without influence simplification (blue) gives better overall results than pure rest-pose simplification (cyan); the only situations in which it is worse are at very low influence counts for some of the models. The effect of including influence simplifications (red) is more variable: on arm, cylinder and cyberdemon it makes a huge improvement across all levels of detail, but for mancandy it performs very poorly in the centre of the range, and for other models it is generally slightly worse across the range. The improvements appear to stem from the ends of the range, most clearly seen for mancandy: superfluous influences can be removed immediately at negligible cost, and for extreme simplifications, influences may be removed in preference to edge collapses of large-scale features.

Our algorithm ranks simplifications purely by geometric deviation, rather than by geometric deviation per removed influence. Influence simplifications remove only a single influence, whereas edge

[2]Two of the graphs for the cylinder model deviate from this shape, but this is because the cylinder model is an artificial example with perfectly straight sides.

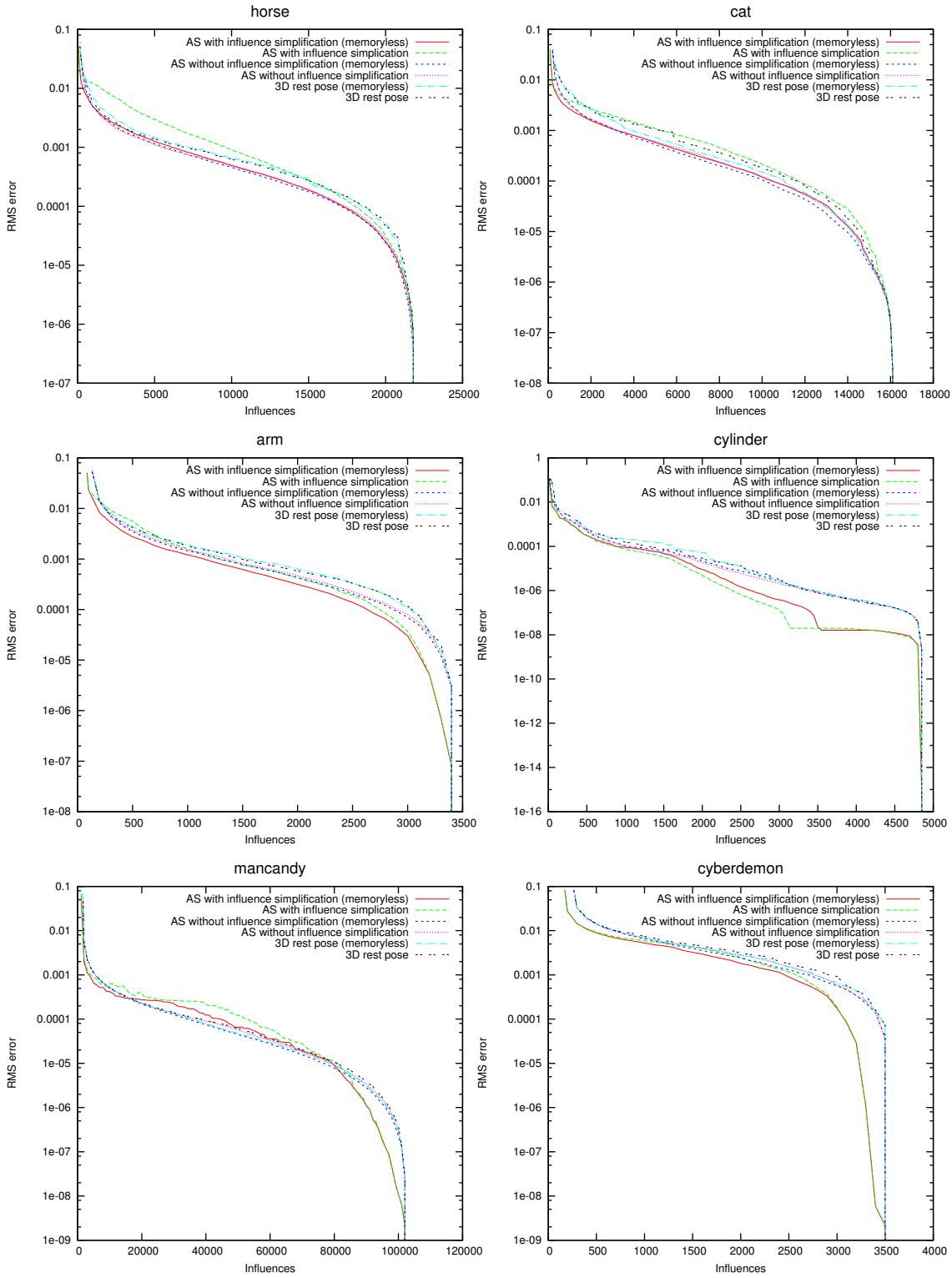


Figure 8.12: Absolute RMS errors for different simplification methods. The errors are shown on a logarithmic scale to handle the large range of values. Note that this makes the differences seem smaller than they actually are; refer to Figure 8.13 for a relative comparison.

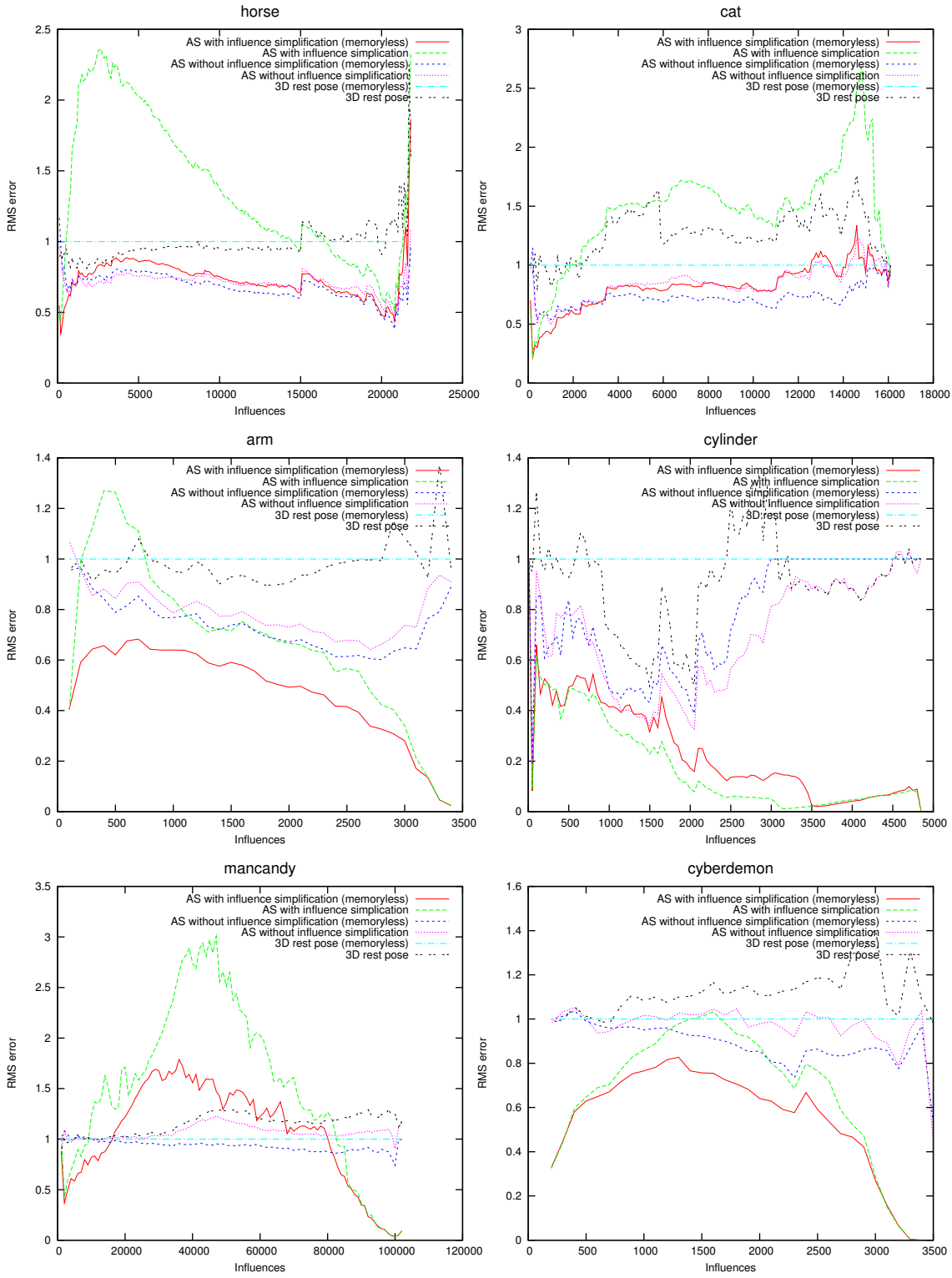


Figure 8.13: Comparison of relative RMS errors for different simplification methods. The errors for each method are divided by the error for the memoryless rest-pose only method.

collapses may remove several. This is why including influence collapses may result in lower quality at the same influence count, even if we have a good estimate of their cost. Ranking simplifications by error per removed influence may yield improvements for mean error, but it will also admit edge collapses with extremely large errors if the vertex that is to be removed has a large number of influences.

Figure 8.14 provides a visual comparison between the results of the three memoryless variants. As expected, a 3D simplification provides particularly poor simplification around the joints of the legs and tail, which are the parts that move most during the animation (note particularly the back-right ankle, which disappears at 680 influences). Using influence simplification reduces the required number of edge collapses, and thus allows more curvature and detail to be maintained. Of course, influence simplification is not free and does introduce some error: at the lowest influence count shown (which would normally only be applied to very distant models), it causes a vertex on the chest to be displaced, leading to the bump on the chest visible in the bottom-right of Figure 8.14.

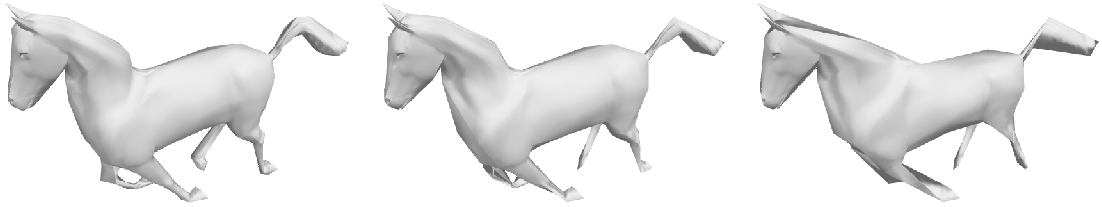
8.5.1 User testing

While animation-space simplification has lower geometric error than rest-pose simplification, this does not guarantee that users will find this to be a visually superior approximation. We thus conducted user experiments to confirm the results above.

The first experiment was run to determine how sensitive users are to different levels of detail. In each test, users were shown pairs of videos. Videos were shown one after the other, rather than side-by-side, so that users would focus on overall quality rather than trying to identify low-level differences. The videos in each pair showed the same scene and used the same camera path (an inward spiral, intended to cover a range of viewing angles and distances), but were created with different rendering options. The videos were kept short (7–8 seconds each) to allow them to be retained in working memory [Baddeley, 1986]. They were rendered at 1280×1024 resolution with anti-aliasing, and were stored with a lossless video codec. After watching each pair of videos, the user was required to specify which one had better quality, with a forced choice. Each pair of videos appeared in two tests, with the order reversed, to eliminate any ordering bias. The tests were also conducted in random order, to avoid any systematic learning bias. Users were told that they were comparing different rendering methods for quality, but nothing more about the nature of those methods.

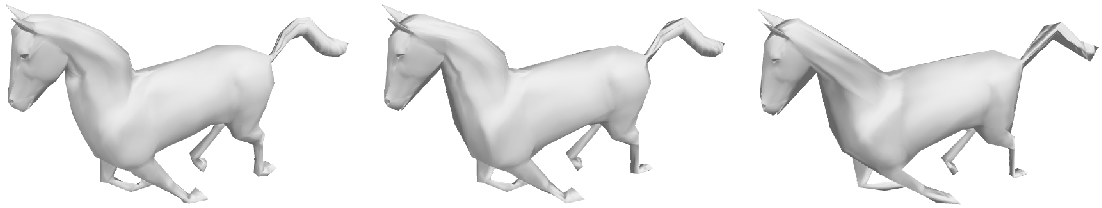
The tests were conducted in a separate room to reduce external noise and to control lighting. We initially intended to conduct the experiments with the lights turned off, to eliminate glare. However, a pilot user complained that the contrast between the video clips (with a black background) and the window that appeared between tests (with a standard light grey background) caused significant eye strain, and so the tests were conducted with the lights on.

For this first experiment, one video was rendered without level-of-detail, while the other was rendered with memoryless animation-space level-of-detail with influence simplification, at a certail



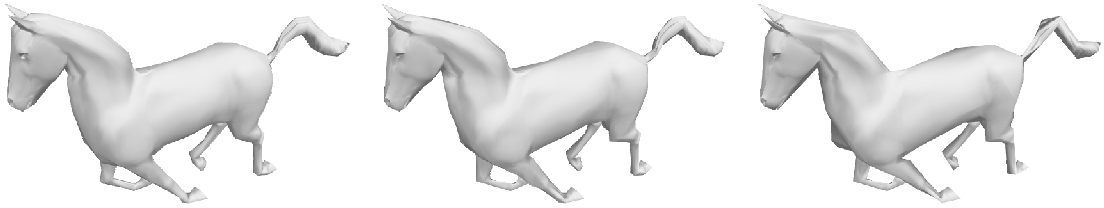
1363 influences 680 influences 337 influences

(a) Simplification from 3D rest pose



1366 influences 680 influences 338 influences

(b) Animation-space simplification, no influence simplification



1366 influences 683 influences 341 influences

(c) Animation-space simplification with influence simplification

Figure 8.14: The horse at three levels of detail, for three simplification methods. Notice how simplification computed purely on the rest pose does not correctly handle the joints at the ankles and in the tail, causing them to vanish.

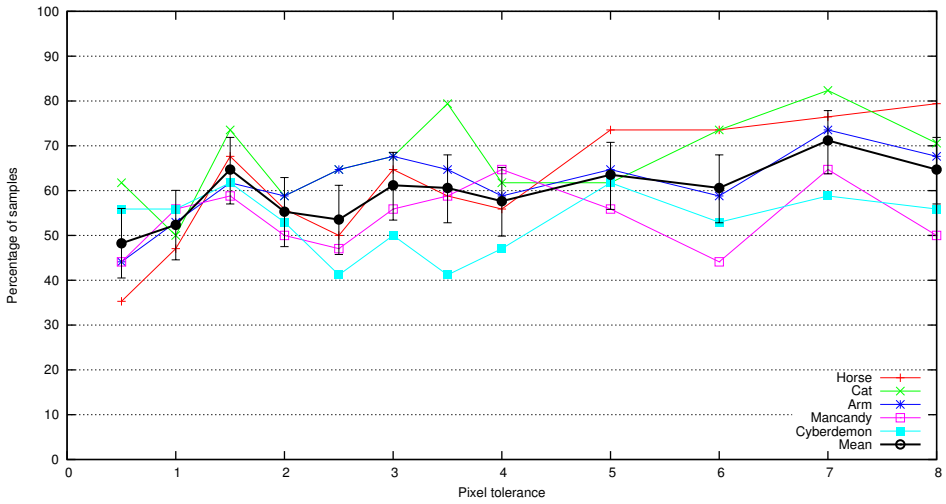


Figure 8.15: Number of users who preferred the full-detail scene to the LOD approximation. Each of the 17 subjects evaluated each scene at each tolerance twice, so the points on the graph reflect 34 samples. The mean graph is the average over the five scenes, so each point reflects 170 samples. The error bars on the mean graph are 95% confidence intervals, although they do not take the learning bias into consideration.

pixel error tolerance. We used a set of twelve different error tolerances, and five scenes. Each scene consisted of one model from Figure 8.1 (cylinder was excluded, as being too artificial), replicated a number of times. Note that the actual errors may be larger than the nominal tolerances, since memoryless simplification only requires that each individual collapse, rather than their accumulation, is within the tolerance.

The raw results may be found in Table D.1, and they are plotted graphically in Figure 8.15. Unfortunately, even with 170 samples per tolerance, there is too much noise in the data for any general conclusions to be reached. We found that the number of times a scene had previously been shown to a subject was a highly significant ($p < 0.002$) predictor of whether the user would pick the full-detail video, indicating that a learning effect was present. This confirms responses to a post-experiment questionnaire, which suggested that subjects would identify specific artefacts (such as mancandy's head developing a point at the top), and then use those artefacts to distinguish between the videos. There was also statistically significant variation between scenes and between subjects, even after accounting for the learning effect.

We ran a further three forced-choice experiments as a single set. The comparisons made in each were:

Experiment 2 animation-space simplification against 3D rest-pose simplification, in both cases without influence simplification (which is meaningless in 3D),

Experiment 3 animation-space simplification with and without influence simplification, and

Experiment 4 two different methods of computing normals during rendering (the results of this experiment are reported later with other rendering results).

The setup was essentially the same as for the first experiment, with users watching pairs of videos and choosing which was preferred. A different set of 23 users, mostly students at the University of Cape Town, took part in this experiment, with only two of them having also taken part in the previous one. We made some improvements based on feedback from the first experiment, and some changes based on the nature of the experiments:

- We used a dark background for the test application, to reduce contrast with the videos, and turned off the lights during the experiment to reduce screen glare.
- In the first experiment, several subjects claimed that there were differences in speed between the videos in a pair. We confirmed that all the videos were recorded correctly, but found that the player was unable to decode the lossless codec fast enough. As a result, videos may have played at slightly variable speed, depending on the CPU load. For the second set of experiments, we reduced the frame rate to 20fps (from a nominal 30fps and roughly 25fps in practice). This had the additional benefit of giving subjects more time to study the animations.
- Because the pixel tolerance is only an approximation of the actual error, it does not make sense to compare different simplifications at the same pixel tolerance. Instead, we used tolerances of 1.5, 3 and 6 pixels for one video in each pair (the animation-space one for Experiment 2, the one with influence simplification for Experiment 3), and computed the pixel tolerance for the second video that gave, as closely as possible, the same number of influences rendered over all the frames of the video. In a few cases for Experiment 3, we were forced to omit the tests with a pixel tolerance of 3 or 6 pixels, as without influence simplification it was impossible to simplify the other video to the necessary extent.
- In order to reduce the learning effect seen in the previous experiment, we added a "training phase". During this first phase, every pair of videos was shown once, in a random order, and preferences were gathered but discarded. After this the real experiment began, in which each pair was shown twice, once in each order (the order of pairs being randomised within each phase). The intent was that subjects would learn to recognise artefacts during the training phase, making the order of pairs during the experiment phase less of a factor. Users were not told that a training phase was used, and the training and experiment phases were indistinguishable to the user.

In Experiment 2, we expected that the error tolerance would be an influencing factor, with more statistical significance at higher error tolerances (since with a zero or small tolerance, the result is indistinguishable from the ideal irrespective of the LOD algorithm). We were surprised to find that the reverse was true. At 1.5 pixels, there was a significant preference ($p < 0.002$) for animation-space simplification over 3D rest-pose simplification, while at 3 and 6 pixels there was no statistically significant result. We conjecture that at 1.5 pixels the errors in the 3D rest-pose simplifications are just noticeable to users but those of animation-space simplifications are not, while at higher tolerances users are unable to distinguish between large errors with little relative difference.

The training phase was successful in reducing the ordering bias seen in Experiment 1. We found no statistically significant dependence on the number of times a scene had been previously shown. The ordering bias was still present (users being more likely to choose the second of the two videos), but was only statistically significant for tolerances of 3 and 6 pixels. This is most likely because at these tolerances users made arbitrary choices, whereas at 1.5 pixels the choices were more informed by the content. We found no dependence on the scene or subject.

In Experiment 3, the choice of scene was highly significant. This is unsurprising, as some models have more redundant influences than others. Only the horse and mancandy models showed statistically significant results, with users preferring influence simplification. This is in spite of the apparently poor performance of influence simplification on mancandy shown in Figure 8.13, suggesting that errors at high and low details are more important than errors at intermediate levels of detail. There was also some variation between subjects for the cyberdemon and mancandy models, but it was not very significant ($0.02 < p < 0.05$).

8.6 Rendering

For benchmarking rendering, we used two test systems. The first is the system described at the start of the chapter, equipped with a 256MB GeForce 6600 on an AGP 8x bus. Graphics hardware continues to make rapid advances, both in raw performance and in architectural improvements, so we also ran tests on a second machine. This machine has an Intel Core2Duo E6600 (clock speed of 2.4GHz), 1GB of DDR800 RAM, and a GeForce 8800GTX with 768MB of RAM on a PCI Express bus.

For maximum performance, we implemented the CPU-based vertex transformation methods in single precision, using SSE.

8.6.1 Vertex transformation

To measure the performance of vertex transformation in isolation, we disabled all shading, LOD selection (we manually specified the discrete level of detail to use, although all the code for geomorphing was still active) and frustum culling, and pointed the camera away from the models in a scene. For each vertex transformation method, we ran tests with a variety of models (the models in Figure 8.1, excluding cylinder), vertices, influences, and number of objects in the scene in order to establish a linear model. In each case we used 200 frames, and discarded the first five, in order to eliminate the effect of startup costs.

We used GNU R [R Development Core Team, 2005] to compute a linear model for each rendering method. Rather than using the default of weighting all observations equally, we assumed that the variance of each observation is proportional to the rendering time, since random fluctuations accumulate over time. The model we fitted was of the form $T = aV + bI + cO + d$, where T is the average time to render a frame, V the number of vertices, I the total number of influences and O

Table 8.7: Linear model for vertex transformation time on a GeForce 6600. The coefficients are given in microseconds for the per-frame and per-object overheads, and nanoseconds for the number of vertices and influences. We also report the R^2 value of the linear regression.

Method	**μs/Frame**	**μs/Object**	**ns/Vertex**	**ns/Influence**	R^2
CPU	2165	−38	60	31	0.977
Alternative CPU	2224	−81	71	36	0.938
Fragment program	2419	125	51	11	0.983
Vertex textures	2122	−23	4	327	0.997

Table 8.8: Linear model for vertex transformation time on a GeForce 8800GTX. The coefficients are given in microseconds for the per-frame and per-object overheads, and nanoseconds for the number of vertices and influences. We also report the R^2 value of the linear regression. The per-influence cost of the fragment program was computed as 0.35ns, but with a standard error of 0.4ns, and so we eliminated it from the model.

Method	**μs/Frame**	**μs/Object**	**ns/Vertex**	**ns/Influence**	R^2
CPU	48	15	9.2	10.3	0.989
Alternative CPU	66	12	6.5	9.4	0.983
Fragment program	48	84	10.9	—	0.978
Vertex textures	63	15	1.8	3.4	0.885

the number of objects (d being a per-frame overhead). Table 8.7 shows these coefficients for the GeForce 6600, while Table 8.8 shows the same data for the GeForce 8800GTX. The raw data are available in Tables D.5 and D.6.

The variation in per-frame overheads between the different rendering methods is well below 1ms and is not statistically significant, so we will ignore it in discussing the relative performance of the different methods. The negative coefficients for the number of objects in Table 8.7 are counter-intuitive, but they are statistically significant. They arise because we instanced a single object multiple times, rather than using different objects. As a result, a single large object will consume 100 times as much memory as 100 instances of a small object with the same total number of vertices and influences; this places greater demands on the memory subsystem, and as a result the rendering is slower, even though per-object code is executed less often. The other test machine (Table 8.8) had more memory, higher memory bandwidth and a larger cache on both the CPU and the video card, and so per-object overheads outweighed these benefits to yield positive coefficients.

On the GeForce 6600, the fragment-program based transformation is faster than CPU-based transformation on a per-vertex and per-influence basis, but also has significantly higher per-object overhead. It will likely be faster for high-detail representations (at least several thousand vertices), but slower for extremely simplified models. A possible approach for a production implementation would be to select the appropriate method on an object-by-object basis to obtain the benefits of both methods, although it it is possible that the overhead of switching methods would be greater than any potential saving. While the GeForce 6600 offers vertex texturing, the performance is clearly inadequate for this purpose.

On the GeForce 8800GTX, the performance is of course much better, but different relationships

emerge. Firstly, the alternative method for CPU transformation (iterating over bones rather than over vertices) becomes slightly better than the original CPU method, at least in terms of per-vertex and per-influence performance; this is likely due to internal differences in the CPUs (Athlon XP compared to a Core2Duo) and memory architectures. The fragment program method becomes essentially independent of the number of influences; we assume that there must be some dependence, but we had insufficient data to determine it with statistical significance (the value reported by R was 0.35ns $\pm$ 0.4ns).

Finally, it is clear that the improved vertex texturing capabilities of the GeForce 8 series make the vertex texturing approach a far more attractive option. The per-object overhead is roughly 3μs higher than the alternative CPU method, so it may under-perform on scenes with many distant objects at extremely low detail (fewer than 250 vertices). It will also have poorer per-vertex performance than the fragment program for models with more than about three influences per vertex (although the standard deviations in the coefficients are too large to identify a specific cross-over point) and the high per-object overhead for the fragment program means that this is only a concern for large models.

We also emphasise that the vertex texturing scheme was designed to fit the constraints of the GeForce 6 series, and so it is likely that further optimisation is possible. In particular, the GeForce 8 would allow us to use hardware texture filtering to perform geomorphing with a single texture lookup, as discussed in Section 7.4.3.

Comparison to SSD and MWE

So far we have compared different methods of vertex transformation for animation space, but it is also important to know whether animation space sacrifices the high performance that made SSD practical for real-time rendering. In fact, we will see that the opposite is true.

To keep this comparison as simple and as general as possible, we did not use run-time LOD, and lighting was ignored. We implemented vertex programs, shown in listings C.12 through C.14, that performed the necessary computations for SSD, animation space and MWE respectively, taking all per-vertex data as OpenGL vertex attributes. This limited the number of influences per vertex to 8 in the case of animation space, and 4 in the case of MWE. In order to reduce the impact of overheads not related to vertex transformation, we used models with high vertex counts (in some cases, by applying Catmull-Clark subdivision to a model that we have used previously) and placed the model outside the view frustum to avoid per-fragment costs. Some of the models exist in fully general animation space, and have no exact SSD equivalent; we produced SSD approximations by simply taking the weight components from the animation-space vectors as weights for SSD. This gives a poor approximation, but quality is of no concern for this test.

Table 8.9 shows the frame rates for the three skinning methods, on the GeForce 6600. Surprisingly, animation space is faster than a straightforward implementation of SSD. The reason is that SSD requires the weight to be multiplied with the position, whereas for animation space it is pre-multiplied and this reduces the instruction count. Of course, the vertex program used for an-

Table 8.9: Comparison of rendering performance for skeletal subspace deformation (SSD), animation space (AS) and multi-weight enveloping (MWE). MWE can only support four influences on any bone in a straightforward implementation, and hence some entries are absent.

Model	Vertices	Bones	Average Influences	FPS		
				SSD	AS	MWE
Cylinder	38786	2	2.0	538	748	358
Tube	57346	2	1.8	366	500	192
Arm	100160	3	1.8	206	286	142
Hand	102274	29	2.5	204	285	—
Horse	137634	24	3.0	154	214	—

imation space could equally well apply to SSD models, since they are a subset of animation-space models. NVIDIA [Dominé, 2003] present a very similar method, but they fail to gain the extra instruction because they do not pre-multiply the weight. For multi-weight enveloping, we based our implementation on Equation (3.3), and the resulting vertex program has almost identical structure to the animation-space program; however, the cost of the extra weights that must be passed (12 per vertex instead of 4) becomes significant, and so MWE is far slower to render than both SSD and animation space.

8.6.2 Normal transformation

For each of the models, we sampled a 256×256 tangent map. To prevent aliasing, we started with a 1024×1024 map, then down-sampled it. We used a simple box filter and the algorithm in Appendix A.1 to average together tangent-plane samples. This resolution is not sufficient for a one-to-one sample-to-pixel correspondence in close-up shots; however, we expect that in a production implementation, close-up models would be rendered at full geometric detail with per-vertex normals for maximum quality. Tangent-space normal maps (TSNMs) were also sampled at 256×256 resolution.

Performance

Tables 8.10 and 8.11 show the performance of three different methods for computing normals: Phong shading (per-vertex normals are interpolated across faces), TSNMs, and tangent maps. We rendered the scenes using the same spiralling camera path employed in user experiments (Section 8.5.1), but the models were deliberately forced to the most extreme simplification to reduce the impact of per-vertex costs on the measurements. In the case of mancandy, this was clearly insufficient, as over 1000 vertices remained and the rendering was an order of magnitude slower than for other models, but we have included the results for completeness.

With both video cards, but particularly on the newer GeForce 8 card, TSNMs have little impact on performance (at worst, rendering takes 50% longer). In contrast, tangent maps are far more expensive, as is to be expected from a method that performs two animation projections on the fly.

Table 8.10: Shading performance on a GeForce 6600. The numbers represent time per fragment, in nanoseconds. *Vertex* is Phong shading (normals interpolated between vertices). The results for tangent maps are reported by tile size, and the result for the best tile size is shown in bold. The ratios are the ratio between the TSNM or best tangent-map time and the Phong-shading time. The models were rendered at the lowest level of detail to reduce the influence of per-vertex costs.

Model	Vertex	TSNM	Tangent map (TM)					TSNM	TM
			2×2	4×4	8×8	16×16	32×32	Ratio	Ratio
horse	88.4	98.3	12690.1	9364.3	**8535.1**	10675.5	11128.6	1.11	96.52
cat	43.0	48.0	4129.3	2752.1	**2253.4**	2557.2	2920.1	1.12	52.35
arm	54.2	67.6	3238.4	1891.5	**1469.0**	1506.0	1566.9	1.25	27.11
cylinder	28.0	34.7	648.1	330.5	279.2	264.1	**259.2**	1.24	9.27
mancandy	1170.7	1770.6	11972.1	**9134.0**	9227.2	12568.0	15218.3	1.51	7.80
cyberdemon	78.1	82.6	3041.3	**2193.7**	2311.7	3151.8	4122.3	1.06	28.09

Table 8.11: Shading performance on a GeForce 8800GTX. The columns have the same meaning as Table 8.10.

Model	Vertex	TSNM	Tangent map (TM)					TSNM	TM
			2×2	4×4	8×8	16×16	32×32	Ratio	Ratio
horse	37.7	41.2	768.4	384.3	**302.7**	313.4	306.3	1.09	8.03
cat	18.8	19.9	268.0	137.0	**102.6**	107.3	102.8	1.06	5.47
arm	25.3	23.3	239.6	143.3	119.2	114.1	**110.4**	0.92	4.37
cylinder	11.4	12.4	61.7	49.6	46.8	46.6	**46.1**	1.08	4.03
mancandy	223.5	228.3	867.4	482.5	**359.9**	388.3	415.5	1.02	1.61
cyberdemon	33.3	34.3	215.1	106.5	**86.7**	95.5	107.4	1.03	2.61

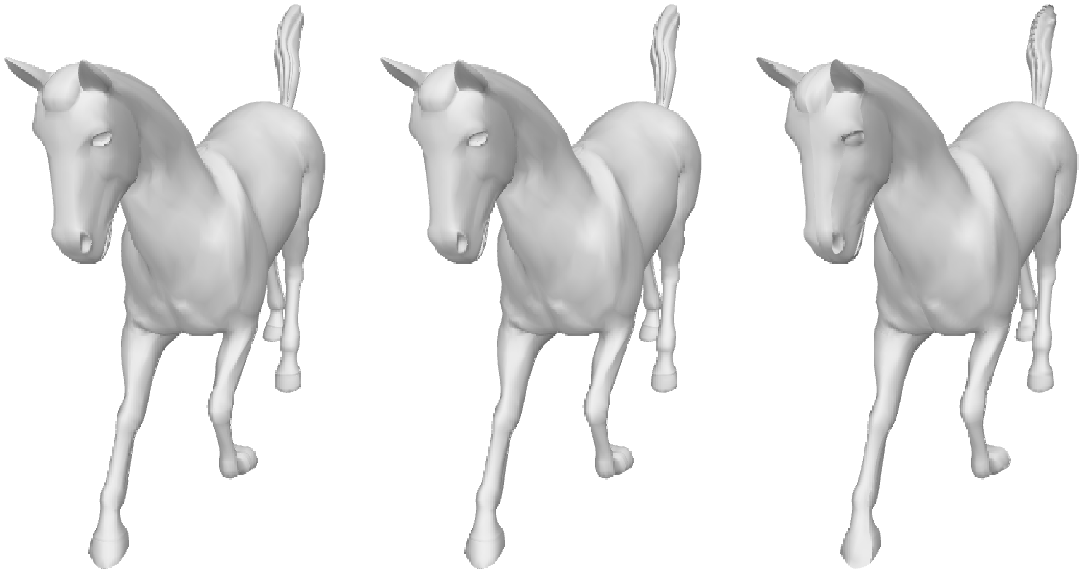
The impact is particularly noticeable on the older hardware, where it can reduce performance by two orders of magnitude. The newer hardware is able to adapt better to this type of workload, and reduces performance by at most one order of magnitude. It is also worth noting that the best tangent-map performance on the GeForce 8800GTX is in several cases less than double that of the TSNM performance on the GeForce 6600; it is thus likely that future hardware will be able to render tangent maps at the same rate as current consumer-level hardware renders TSNMs.

For both hardware configurations, it appears that 8×8 tiles produce a reasonable trade-off between tiles that are too small (and do not fully exploit the available parallelism, as can clearly be seen for 2×2 tiles) and too large (and waste memory when influence counts vary over the surface). Although this is not always the optimal size, the performance is consistently within 10% of optimal.

Quality

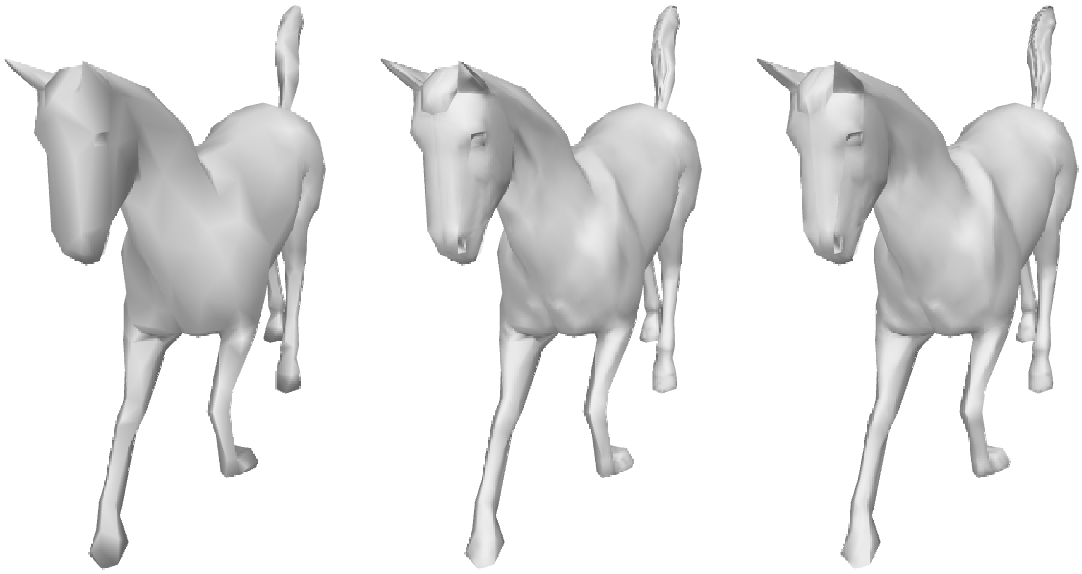
Figures 8.16 and 8.17 show the horse shaded with interpolated per-vertex normals and with our two per-pixel normal methods. Figure 8.16 shows the full-detail horse, and in particular, 8.16(a) is the ideal image. In this figure it is clear that the tangent-space normal map produces visually identical results; this is to be expected, as the normal map contains corrections to the per-vertex normal, and at full detail there are no corrections to make. The tangent map does less well: there are visible seams on the face due to chart boundaries, the eye is darker than it should be, and the surface of the tail appears jagged because it runs at an angle through the texture.

Figure 8.17 shows a simplified version of the horse, which is of course the circumstance under



(a) Vertex normals (b) Static normals (c) Dynamic normals

Figure 8.16: Normal calculation methods for the horse at full detail



(a) Vertex normals (b) Static normals (c) Dynamic normals

Figure 8.17: Normal calculation methods for the horse at reduced detail

which we expect the texture-based shading methods to be applied. The geometry of the face has been almost entirely removed, but the shading restores the nostril, eye, and start of the mane, as well as adding definition to the muscles of the front shoulders and correcting the shading on the front-right foot. In this case, there is very little difference between the two texture-based methods: both show seams at chart boundaries and artefacts on the tail.

Since the level of detail shown in Figure 8.17 would normally only be used for a distant horse, it is unlikely that users would be able to distinguish between the shading methods. Our fourth user experiment, conducted as described in Section 8.5.1, aimed to determine this. We found that overall there was no statistically significant preference for one method over the other and that there was no variation between scenes or subjects. The order in which videos were shown was a significant factor, with users more likely to choose the second video shown. The probability of selecting a TSNM over a tangent map has a 95% confidence interval of $[0.46, 0.54]$, i.e., the results not necessarily identical, but any differences are quite minor.

8.7 Summary

Fitting an animation-space model to a set of examples provides a good fit. Edge regularisation eliminates some shearing artefacts, but even without edge regularisation (i.e., using vertex regularisation), animation space provides a close yet robust fit. In contrast, SSD provides a poor fit while MWE suffers from over-fitting.

Our area-ratio metric for segmentation produces better results than the prior mean-squared distance metric, but our ad-hoc methods for straightening chart edges behave poorly in many cases. Flattening benefits from computations performed in animation space, provided that expectations are estimated from some poses (rather than relying on the uniform rotations, fixed rotations model).

Level of detail approximations computed in animation space are better than those computed from only a single pose, both on the basis of geometric deviation and users' perception. Influence simplification can substantially reduce geometric errors at a particular level of detail, but can also increase them (particularly when used with standard, rather than memoryless, APS). Perceptually, influence simplification was either an improvement or made no measurable difference, depending on the model used.

Animation space models are far cheaper to render than MWE models, and the most efficient way to render SSD models is to transform them into animation space. On the GeForce 6600, which has poor vertex texturing support, the most efficient of our vertex transformation methods is the fragment program, at least for representations with many vertices. On the GeForce 8800GTX, which has good vertex texturing support, vertex texturing is faster for representations with few influences per vertex. Tangent-space normal maps are an order of magnitude faster to render than tangent maps, and we found no perceptual difference between them.

Chapter 9

Conclusions and future work

Our primary contribution is the animation-space framework, based on the simple and elegant linear equation $\mathbf{v} = G\mathbf{p}$ (where $\mathbf{p}$, G and $\mathbf{v}$ are respectively the model-space position, the animation projection matrix, and the animation-space position), together with the $L_{2,2}$ and $L_{2,\infty}$ metrics we defined on it. This makes animation space a powerful framework, both for its flexibility in modelling, and as a mathematical basis for adapting existing 3D algorithms to operate on animated characters. Furthermore, it is highly efficient and suitable for real-time rendering.

9.1 Modelling

Animation space provides four degrees of freedom per influencing bone, in contrast to the one provided by SSD. Although some of these degrees of freedom are redundant, this generality allows it to represent shapes that are impossible with SSD.

Extra degrees of freedom are not always desirable, since they complicate the modelling process. We have, however, shown that it is practical to create animation-space models with a variety of methods. The easiest is to simply create an SSD model, or indeed reuse an existing SSD model: since animation space generalises SSD, the applications in the next section can be readily applied to the large existing base of SSD models. However, the extra degrees of freedom make it possible to create higher-quality models than is possible with SSD. We have presented three methods to create animation-space models that cannot be represented in SSD:

fitting a model to examples This requires that examples are created or are available from some other source, such as a high-end physically-based animation system that is too expensive for real-time applications. We found that animation space was robust to over-fitting, even for small numbers of sample poses, making this a practical approach.

subdivision surfaces Unlike SSD, applying any subdivision scheme in animation space is precisely equivalent to applying it to each 3D pose of a model. While there are currently no

user interfaces to allow editing in animation space, this theoretically makes multi-resolution editing feasible in a way that is not possible with SSD.

influence simplification Although we have not presented influence simplification in these terms, applying it to an SSD model will generally produce an output that is no longer representable in SSD. This means that modellers may continue to use existing software tools, and use helper bones or pseudo-bones [Mohr and Gleicher, 2003b] to work around the limitations of SSD, then use influence simplification to eliminate these crutches. This will produce an animation-space model that has fewer influences and will hence be faster to render.

9.2 Applications

Our first validation of animation space as a computational framework was in parametrisation. We found that computing parametrisations using the animation-space $L_{2,2}$ metric reduces the overall texture stretch across a range of poses. Independently of animation space (or indeed of animation in general), we can conclude that the greedy bottom-up merging process proposed by Garland et al. [2001] produces charts that are poorly shaped for level-of-detail applications (due to the ragged boundaries), even when we introduce an alternative cost metric that avoids some shortcomings of the original.

We also demonstrated the value of using animation space by applying it to level of detail (LOD). Again, we were able to leverage an existing algorithm, appearance preserving simplification (APS), by replacing the existing Euclidean metric by our $L_{2,2}$ metric. We obtained good results only with the memoryless form of APS, but since this is faster, more memory efficient and has similar quality to the standard form when used in 3D, this is not really a disadvantage. For the same number of influences (and hence a similar rendering cost), we found that the simplifications produced by animation space have far less error on a number of practical examples. User tests confirmed this by showing that subjects had a statistically significant preference for animation-space simplifications. Furthermore, the significant results were found at low error tolerances, where LOD is typically used in practice, rather than on models that were deliberately over-simplified in order to emphasise distortions. While other methods exist to take multiple poses into account, they are based on a sampling of the pose space, and the computational cost scales with the number of samples used. In contrast, we use samples only to compute statistics, and the time required to evaluate the $L_{2,2}$ metric is independent of the number of samples.

We further improved the LOD of animated models with a novel approach, influence simplification, in which we simplify the position of a vertex by removing the influence of a bone on it. We combined influence simplification with a standard progressive mesh structure to obtain superior quality in simplification. The results vary according to the number of extraneous influences in the original model, but in some cases this approach reduces the deviation at a particular influence count by an order of magnitude. Similar results were seen in user testing, with no difference found for some models, but a highly significant preference for influence simplification in others. Apart from

reducing approximation error, influence simplification also makes it possible to simplify a model further than is possible with edge collapses alone, potentially reducing rendering cost significantly for large crowd scenes where many of the models are extremely distant.

These two examples were selected to illustrate the value of animation space as a framework for bringing static 3D algorithms into an animated context, but are by no means the only possible applications.

9.3 Rendering

Finally, we considered rendering aspects. We showed that the extra per-vertex data needed by animation space, particularly when doing geomorphing and projecting tangent planes, can be handled by exploiting the features of modern hardware. The latest generation of commodity GPU hardware has extra capabilities (namely, the ability to load data into the vertex shader other than through vertex attributes) that make this relatively straightforward, and as video hardware continues to evolve we expect that this will soon entirely cease to be a concern. Furthermore, we found that unlike all other improvements to SSD, there is no loss in performance, at least on NVIDIA hardware — in fact, implementing SSD in the form suggested by animation space (that is, by pre-multiplying the weight) makes it faster.

We also implemented two shading methods: tangent-space normal maps (TSNMs), which encode only non-animated information, and tangent maps, which encode the animation-space tangent plane at every sample point on the mesh. Unlike the use of animation space in creating simplifications, which has no extra cost during rendering, tangent maps are an order of magnitude slower than TSNMs. However, tangent maps provide close to ideal shading, and are thus useful as a baseline against which to measure TSNMs, in spite of their relatively poor performance. We conducted user tests, and found no statistically significant preference for either method. We can thus recommend TSNMs for rendering most animation-space models, even though they are tied to a single rest pose.

9.4 Future work

While we have presented methods to create a model as a whole, we have not developed any methods or user interfaces for editing such models. A possible application would be in refining an existing SSD model, by adjusting it only in areas where SSD produces unacceptable results.

We encountered several problems in trying to produce charts with edges that lay along straight lines in parameter space, which was necessary to obtain good results from APS (none of these problems were related to animation). It appears that this restriction of the parametrisation problem has not received much attention in the literature. A promising approach is circle-pattern parametrisation [Kharevych et al., 2006], which provides control over the angle at each bound-

ary point, thus allowing the boundary to be forced straight, without constraining the lengths of boundary segments. Since circle-pattern parametrisation, and indeed all the flattening schemes with which we are familiar, depend only on the intrinsic properties of the surface, they can be adapted to animation space in the same way that we adapted LSCM.

When creating a progressive mesh, we used only the half-edge collapse. While this eased implementation, it also constrains any simplification to lie within the convex hull of the original, and as a result models appear to shrink as they are simplified. Applying a full edge collapse requires a position to be computed for the newly created vertex; we expect that optimisation techniques similar to those we used for influence simplification should be applicable here.

We considered only two applications of animation space as a computational framework, but there are many other possibilities. An example is run-time subdivision, where animation space would make it possible to determine the error in any particular representation, much as is currently done in our level-of-detail scheme. The $L_{2,\infty}$ metric was not heavily used in this thesis (the one exception being view frustum culling), but it should be applicable wherever bounding volumes are appropriate, such as collision detection or visibility determination. It should also be possible to apply the $L_{2,\infty}$ metric (in place of the $L_{2,2}$ metric) to level-of-detail, which would give error bounds that are guaranteed for any legal pose.

A theoretical inconvenience of animation space is that the expectation matrix $P = E[G^T G]$ may be singular, meaning that the $L_{2,2}$ "norm" is not a true norm. In many cases this is not an issue, but it adds complications to fitting and influence simplification, where the optimum may be ill-defined. We handled this on a case-by-case basis, but a more elegant and unified solution would be preferable. A possible avenue of exploration is to apply some form of regularisation to the computation of the expected values, i.e., assume that every joint has at least some freedom of rotation around any axis and of translation in any direction, even when this is not observed in example poses. Apart from resolving the problem of a singular P matrix, this would make models more robust to new poses that lie outside the space spanned by their statistical models.

Another issue related to the $L_{2,2}$ norm is that, since it is non-Euclidean, an axis-aligned bounding box is a poor choice for algorithms such as standard APS. We solved this by using a bounding box in *normalised* animation space, a linear transformation of animation space in which the $L_{2,2}$ norm is transformed to the Euclidean norm, but the transformation destroys the sparsity of the original animation-space coordinates. As a result, algorithms using it scale poorly with the number of bones in the entire model. Furthermore, the axes of normalised animation space may be chosen freely, and our essentially arbitrary choice means that the bounding boxes may be poor approximations. It would be interesting to determine whether it is possible to choose a set of axes which either makes the transformation from animation space sparser, or has some meaningful geometric interpretation, or both. An alternative bounding volume that overcomes the density problem, but whose approximation quality would need investigation, is a bounding ellipsoid defined by $\{\mathbf{p} : \|\mathbf{p} - \mathbf{c}\|_{2,2} \leq r\}$ (a sphere in normalised animation space). When combining bounding volumes, the new centre would lie within the span of the existing centres, and so sparsity would be maintained as long as the original centres all had similar influence sets.

Although we found that tangent-space normal maps provide acceptable shading, it is nevertheless unsatisfying to have this one piece of the modelling and rendering process depend on a specific rest pose. It should be possible to compute an optimal normal map that minimises the errors in the normals across all poses. However, the $L_{2,2}$ norm is likely to be insufficient, and some non-linear optimisation may be required.

Our rendering method based on vertex textures is designed to operate within the constraints of the GeForce 6 series. There are many features in DirectX 10 [Blythe, 2006] and the equivalent OpenGL extensions that could allow further optimisations. Most notably, vertex texturing is required to support filtering, which could be exploited to do geomorphing at almost no cost, and transform feedback would make it possible to transform every vertex exactly once per model rather than each time it is encountered in the triangle list.

Currently, our system uses only one vertex transformation method and one normal transformation method at a time. The results indicate that some methods are better for representations with few vertices (due to low overheads) while others are better for large numbers of vertices (due to high throughput). It is worth investigating a hybrid scheme which selects a rendering method on a per-object basis within each frame. The key question is whether the benefits of using the optimal rendering method for each model outweigh the costs of switching between the methods, or the extra memory required to store the same model in multiple formats.

References

Matt Aitken, Greg Butler, Dan Lemmon, Eric Saindon, Dana Peters, and Guy Williams. The Lord of the Rings: the visual effects that brought middle earth to the screen. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 11, New York, NY, USA, 2004. ACM Press. doi: http://doi.acm.org/10.1145/1103900.1103911.

Marc Alexa. Linear combination of transformations. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 380–387, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566592.

Brett Allen, Brian Curless, and Zoran Popović. Articulated body deformation from range scan data. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 612–619, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566626.

D. Anguelov, P. Srinivasan, D. Koller, S. Thrun, H. Pang, and J. Davis. The correlated correspondence algorithm for unsupervised registration of nonrigid surfaces. In *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2004a.

Dragomir Anguelov, Daphne Koller, Hoi-Cheung Pang, Praveen Srinivasan, and Sebastian Thrun. Recovering articulated object models from 3D range data. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 18–26, Arlington, Virginia, United States, 2004b. AUAI Press. ISBN 0-9749039-0-6.

A. D. Baddeley. *Working memory.* Clarendon Press, 1986.

James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292, New York, NY, USA, 1978. ACM Press. doi: http://doi.acm.org/10.1145/800248.507101.

David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1141911.1141947.

George Borshukov. Measured BRDF in film production: realistic cloth appearance for "The Matrix Reloaded". In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches and Applications*, New York, NY, USA, 2003. ACM Press. doi: http://doi.acm.org/10.1145/965400.965468.

George Borshukov, Dan Piponi, Oystein Larsen, J. P. Lewis, and Christina Tempelaar-Lietz. Universal capture: image-based facial animation for "The Matrix Reloaded". In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches and Applications*, New York, NY, USA, 2003. ACM Press. doi: http://doi.acm.org/10.1145/965400.965469.

Hector M. Briceño, Pedro V. Sander, Leonard McMillan, Steven Gortler, and Hugues Hoppe. Geometry videos: a new representation for 3D animations. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 136–146. Eurographics Association, 2003. ISBN 1-58113-659-5.

Ross Brown, Luke Cooper, and Binh Pham. Visual attention-based polygon level of detail management. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Austalasia and South East Asia*, pages 55–ff. ACM Press, 2003. ISBN 1-58113-578-5. doi: http://doi.acm.org/10.1145/604471.604485.

S. R. Buss. Introduction to inverse kinematics with Jacobian transpose, pseudoinverse and damped least squares methods. URL http://www.math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/index.html. April 2004.

Cal3d. cal3d — an open source 3D character animation library that can be used on almost any platform. URL http://gna.org/projects/cal3d/.

Edwin Catmull and James Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, 1978.

J. E. Chadwick, D. R. Haumann, and R. E. Parent. Layered construction for deformable animated characters. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 243–252, New York, NY, USA, 1989. ACM Press. ISBN 0-201-50434-0. doi: http://doi.acm.org/10.1145/74333.74358.

Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122. ACM Press, 1998. ISBN 0-89791-999-8. doi: http://doi.acm.org/10.1145/280814.280832.

David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. *ACM Trans. Graph.*, 23(3):905–914, 2004. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1015706.1015817.

G. Collins and A. Hilton. Modelling for character animation. *Software Focus*, 2(2):44–51, 2001.

Brian Curless. From range scans to 3D models. *SIGGRAPH Comput. Graph.*, 33(4):38–41, 2000. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/345370.345399.

Christopher DeCoro and Szymon Rusinkiewicz. Pose-independent simplification of articulated meshes. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 17–24, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-013-2. doi: http://doi.acm.org/10.1145/1053427.1053430.

Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 85–94. ACM Press, 1998. ISBN 0-89791-999-8. doi: http://doi.acm.org/10.1145/280814.280826.

Mathieu Desbrun, Mark Meyer, and Pierre Alliez. Intrinsic parameterizations of surface meshes. *Computer Graphics Forum*, 21(3):209–218, 2002.

Sim Dietrich. Optimizing for hardware transform and lighting, 2000. URL http://developer.nvidia.com/object/hardware_tnl.html.

Sébastien Dominé. Mesh skinning, 2003. URL http://developer.nvidia.com/object/skinning.html.

Leo Dorst and Stephen Mann. Geometric algebra: A computational framework for geometrical applications (part 1). *IEEE Computer Graphics and Applications*, 22(3):24–31, 2002.

Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 173–182. ACM Press, 1995. ISBN 0-89791-701-4. doi: http://doi.acm.org/10.1145/218380.218440.

Jihad El-Sana, Elvir Azanli, and Amitabh Varshney. Skip strips: maintaining triangle strips for view-dependent rendering. In *Proceedings of the conference on Visualization '99*, pages 131–138. IEEE Computer Society Press, 1999. ISBN 0-7803-5897.

Olivier Faugeras. *Three-dimensional computer vision: a geometric viewpoint*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-06158-9.

M. S. Floater and K. Hormann. Surface parameterization: a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Multiresolution in Geometric Modelling*. Springer, 2004.

Michael S. Floater. Mean value coordinates. *Comput. Aided Geom. Des.*, 20(1):19–27, 2003. ISSN 0167-8396. doi: http://dx.doi.org/10.1016/S0167-8396(02)00002-5.

Sven Forstmann and Jun Ohya. Fast skeletal animation by skinned arc-spline based deformation. In *EG 2006 Short Papers*, 2006.

Michael Garland. Multiresolution modeling: Survey and future opportunities. In *Eurographics '99 – State of the Art Reports*, pages 111–131, 1999.

Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216. ACM Press/Addison-Wesley Publishing Co., 1997. ISBN 0-89791-896-7. doi: http://doi.acm.org/10.1145/258734.258849.

Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98*, pages 263–269. IEEE Computer Society Press, 1998. ISBN 1-58113-106-2.

Michael Garland, Andrew Willmott, and Paul S. Heckbert. Hierarchical face clustering on polygonal surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 49–58. ACM Press, 2001. ISBN 1-58113-292-1. doi: http://doi.acm.org/10.1145/364338.364345.

Philipp Gerasimov, Randima Fernando, and Simon Green. Shader model 3.0: Using vertex textures, 2004. URL http://download.nvidia.com/developer/Papers/2004/Vertex_Textures/Vertex_Textures.pdf.

T. Giang, R. Mooney, C. Peters, and C. O'Sullivan. ALOHA: Adaptive level of detail for human animation: towards a new framework. In *Eurographics 2000 short paper proceedings*, pages 71–77, 2000.

Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins University Press, 3rd edition, 1996. ISBN 0-8018-5414-8.

Xianfeng Gu and Shing-Tung Yau. Global conformal surface parameterization. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 127–137. Eurographics Association, 2003. ISBN 1-58113-687-0.

Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361. ACM Press, 2002. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566589.

David Henry. MD5Mesh and MD5Anim file formats (Doom 3's models), 2005. URL http://tfc.duke.free.fr/coding/md5-specs-en.html.

Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108. ACM Press, 1996. ISBN 0-89791-746-4. doi: http://doi.acm.org/10.1145/237170.237216.

Hugues Hoppe. New quadric metric for simplifiying meshes with appearance attributes. In *Proceedings of the conference on Visualization '99*, pages 59–66. IEEE Computer Society Press, 1999. ISBN 0-7803-5897.

Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198. ACM Press/Addison-Wesley Publishing Co., 1997. ISBN 0-89791-896-7. doi: http://doi.acm.org/10.1145/258734.258843.

Fu-Chung Huang, Bing-Yu Chen, and Yung-Yu Chuang. Progressive deforming meshes based on deformation oriented decimation and dynamic connectivity updating. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 53–62, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. ISBN 3-905673-34-7.

David Jacka, Ashley Reid, Bruce Merry, and James Gain. A comparison of linear skinning techniques for character animation. Technical Report CS07-03-00, Department of Computer Science, University of Cape Town, 2007.

Doug L. James and Christopher D. Twigg. Skinning mesh animations. *ACM Trans. Graph.*, 24 (3):399–407, 2005. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1073204.1073206.

I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, 1986.

Kolja Kähler, Jörg Haber, and Hans-Peter Seidel. Dynamic refinement of deformable triangle meshes for rendering. In *Proceedings of Computer Graphics International '01*, pages 285–290. IEEE Computer Society Press, 2001.

James T. Kajiya. Anisotropic reflection models. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 15–21, New York, NY, USA, 1985. ACM Press. ISBN 0-89791-166-0. doi: http://doi.acm.org/10.1145/325334.325167.

Ladislav Kavan and Jiří Žára. Spherical blend skinning: a real-time deformation of articulated models. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 9–16, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-013-2. doi: http://doi.acm.org/10.1145/1053427.1053429.

Ladislav Kavan, Steven Collins, Jiri Zara, and Carol O'Sullivan. Skinning with dual quaternions. In *2007 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM Press, April/May 2007.

Liliya Kharevych, Boris Springborn, and Peter Schröder. Discrete conformal mappings via circle patterns. *ACM Trans. Graph.*, 25(2):412–438, 2006. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1138450.1138461.

Andrei Khodakovsky, Nathan Litke, and Peter Schröder. Globally smooth parameterizations with low distortion. *ACM Trans. Graph.*, 22(3):350–357, 2003. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/882262.882275.

Leif Kobbelt. $\sqrt{3}$-subdivision. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 103–112, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN 1-58113-208-5. doi: http://doi.acm.org/10.1145/344779.344835.

Paul G. Kry, Doug L. James, and Dinesh K. Pai. Eigenskin: real time large deformation character skinning in hardware. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 153–159. ACM Press, 2002. ISBN 1-58113-573-4. doi: http://doi.acm.org/10.1145/545261.545286.

Tsuneya Kurihara and Natsuki Miyata. Modeling deformable human hands from medical images. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 355–363, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. ISBN 3-905673-14-2. doi: http://doi.acm.org/10.1145/1028523.1028571.

Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 362–371. ACM Press, 2002. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566590.

J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer Graphics and Interactive Techniques*, pages 165–172. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 1-58113-208-5. doi: http://doi.acm.org/10.1145/344779.344862.

Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM Press, 2001. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383274.

Charles Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Dept. of Mathematics, 1987.

N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. Joint-dependent local deformations for hand animation and object grasping. In *Proceedings on Graphics interface '88*, pages 26–33. Canadian Information Processing Society, 1988.

Nadia Magnenat-Thalmann, Frederic Cordier, Hyewon Seo, and George Papagianakis. Modeling of bodies and clothes for virtual environments. In *Third International Conference on Cyberworlds (CW'04)*, pages 201–208, 2004.

Bruce Merry, Patrick Marais, and James Gain. Normal transformations for articulated models. In *ACM SIGGRAPH 2006 Conference Abstracts and Applications*, 2006.

Alex Mohr and Michael Gleicher. Deformation sensitive decimation. Technical Report 4/7/2003, University of Wisconsin, Madison, 2003a.

Alex Mohr and Michael Gleicher. Building efficient, accurate character skins from examples. *ACM Trans. Graphics*, 22(3):562–568, 2003b. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/882262.882308.

Alex Mohr, Luke Tokheim, and Michael Gleicher. Direct manipulation of interactive character skins. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 27–30. ACM Press, 2003. ISBN 1-58113-645-5. doi: http://doi.acm.org/10.1145/641480.641488.

NVIDIA. Cg toolkit user's manual — a developer's guide to programmable graphics, September 2005a. URL http://developer.nvidia.com/object/cg_toolkit.html.

NVIDIA. NVIDIA CUDA: Compute unified device architecture, 2007. URL http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

NVIDIA. NVIDIA GPU programming guide, July 2005b. Version 2.4.0. URL http://download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf.

Christopher C. Paige and Michael A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Softw.*, 8(1):43–71, 1982. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/355984.355989.

Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 303–306, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. doi: http://doi.acm.org/10.1145/258734.258873.

Michael Potmesil. Generating octree models of 3D objects from their silhouettes in a sequence of images. *Comput. Vision Graph. Image Process.*, 40(1):1–29, 1987. ISSN 0734-189X. doi: http://dx.doi.org/10.1016/0734-189X(87)90053-3.

Emil Praun and Hugues Hoppe. Spherical parametrization and remeshing. *ACM Trans. Graph.*, 22(3):340–349, 2003. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/882262.882274.

R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. URL http://www.R-project.org. ISBN 3-900051-07-0.

Taehyun Rhee, J.P. Lewis, and Ulrich Neumann. Real-time weighted pose-space deformation on the GPU. *Computer Graphics Forum*, 25(3):439–448, 2006.

Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the ICP algorithm. In *Proceedings of the Third Intl. Conf. on 3D Digital Imaging and Modeling*, pages 145–152, 2001.

Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 409–416. ACM Press, 2001. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383307.

Pedro V. Sander, Steven J. Gortler, John Snyder, and Hugues Hoppe. Signal-specialized parametrization. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 87–98. Eurographics Association, 2002. ISBN 1-58113-534-3.

Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 151–160, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-196-2. doi: http://doi.acm.org/10.1145/15922.15903.

Mark Segal and Kurt Akeley. The OpenGL graphics system: a specification, December 2006. URL http://www.opengl.org/registry/doc/glspec21.20061201.pdf. Version 2.1.

Andy Serkis. *Gollum: A Behind the Scenes Guide of the Making of Gollum*. Houghton Mifflin, 2003. ISBN 0618391045.

Ariel Shamir and Valerio Pascucci. Temporal and spatial level of details for dynamic meshes. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 77–84. ACM Press, 2001. ISBN 1-58113-427-4. doi: http://doi.acm.org/10.1145/505008.505023.

A. Sheffer and E. de Sturler. Surface parameterization for meshing by triangulation flattening. In *Proc. 9th International Meshing Roundtable*, pages 161–172, 2000.

Le-Jeng Shiue, Ian Jones, and Jörg Peters. A realtime GPU subdivision kernel. *ACM Trans. Graph.*, 24(3):1010–1015, 2005. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1073204.1073304.

Ken Shoemake. Uniform random rotations. In David Kirk, editor, *Graphics Gems III*, pages 124–132. Academic Press, 1992.

Samuel Silva, Joaquim Madeira, and Beatriz Sousa Santos. POLYMECO: A polygonal mesh comparison tool. In *IV '05: Proceedings of the Ninth International Conference on Information Visualisation (IV'05)*, pages 842–847, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2397-8. doi: http://dx.doi.org/10.1109/IV.2005.98.

Karan Singh and Eugene Fiume. Wires: a geometric deformation technique. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 405–414, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-999-8. doi: http://doi.acm.org/10.1145/280814.280946.

Peter-Pike J. Sloan, Charles F. Rose, III, and Michael F. Cohen. Shape by example. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 135–143. ACM Press, 2001. ISBN 1-58113-292-1. doi: http://doi.acm.org/10.1145/364338.364382.

David Stewart and Zbigniew Leyk. Meschach library version 1.2b, 1994. URL http://www.math.uiowa.edu/~dstewart/meschach/README.

Robert W. Sumner and Jovan Popović. Deformation transfer for triangle meshes. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 399–405, New York, NY, USA, 2004. ACM Press. doi: http://doi.acm.org/10.1145/1186562.1015736.

Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. Polycube-maps. *ACM Trans. Graph.*, 23(3):853–860, 2004. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1015706.1015810.

Ken Turkowski. Transformations of surface normal vectors. Technical Report 22, Apple Computer, Inc., July 1990.

Georg Umlauf. Analysis and tuning of subdivision algorithms. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics*, pages 33–40, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-203-6. doi: http://doi.acm.org/10.1145/1090122.1090128.

Xiaohuan Corina Wang and Cary Phillips. Multi-weight enveloping: least-squares approximation techniques for skin animation. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 129–138. ACM Press, 2002. ISBN 1-58113-573-4. doi: http://doi.acm.org/10.1145/545261.545283.

R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. URL http://www.cs.utsa.edu/~whaley/papers/spercw04.ps.

Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of the 7th conference on Visualization '96*, pages 327–ff. IEEE Computer Society Press, 1996. ISBN 0-89791-864-9.

G. Zigelman, R. Kimmel, and N. Kiryati. Texture mapping using surface flattening via multidimensional scaling. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):198–207, 2002. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/2945.998671.

Appendix A

Algorithms

A.1 Fitting a plane

In this section, we derive an algorithm to fit a plane to a set of given oriented planes (all passing through the origin), within a space of arbitrary dimension. The algorithm is used in Sections 4.3.1 (page 39) and 5.2.1 (page 47). To define and solve the problem, we will use geometric algebra, following the notation of Dorst and Mann [2002].

A.1.1 Basic algorithm

To represent a plane, we will use the bivectors (2-blades) of geometric algebra, which we will denote with uppercase bold letters. Bivectors also have magnitude, which is useful as it allows us to weight the elements we want to fit. To measure the "distance" between $\mathbf{B}_1$ and $\mathbf{B}_2$, we will use the negative of the inner product $-\mathbf{B}_1 \rfloor \mathbf{B}_2$ (we negate the inner product since we want the value to be larger the closer together the planes are). In three dimensions, this is equivalent to the dot product of the normal vectors. If $\mathbf{B}_1$ and $\mathbf{B}_2$ are unit bivectors, then this metric can also be visualised as the area obtained by projecting a unit square in the one plane onto the other plane, as shown in Figure A.1.

To set up the optimisation problem, let the input planes be represented by the bivectors $\mathbf{B}_i$ (with possibly non-uniform magnitudes, if there is a need to weight the inputs), and the fitted plane be the unit bivector $\mathbf{B}$. We aim to maximise

$$D = -\sum \mathbf{B}_i \rfloor \mathbf{B}. \tag{A.1}$$

We proceed by expanding the bivectors into outer products: let $\mathbf{B}_i = \mathbf{a}_i \wedge \mathbf{b}_i$ and let $\mathbf{B} = \mathbf{a} \wedge \mathbf{b}$. To regularise the problem, we will only consider orthonormal bases $\{\mathbf{a}, \mathbf{b}\}$ for $\mathbf{B}$. Using the rules

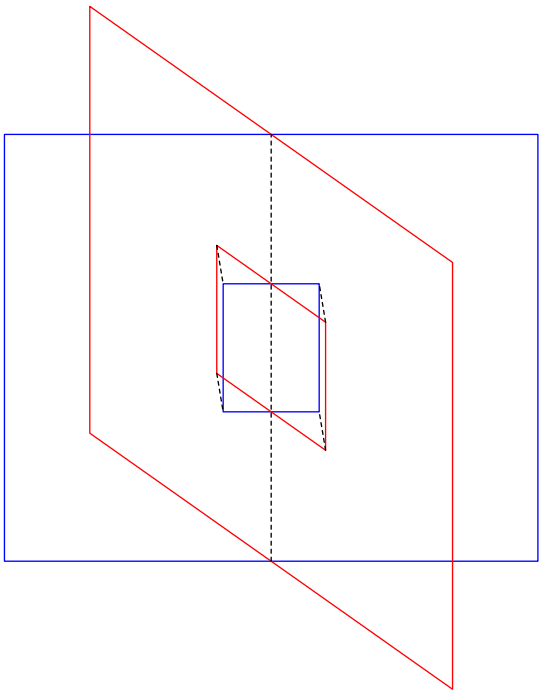


Figure A.1: Visualisation of the geometric algebra inner product for two planes. The red unit square in the red plane is projected onto the blue plane. The resulting blue rectangle has a directed area of 1 if the planes are parallel, 0 if they are perpendicular, and -1 if they are parallel but with opposite orientation.

for expanding inner products,

$$\begin{aligned} D &= -\sum(\mathbf{B}_i \rfloor \mathbf{B}) \\ &= -\sum(\mathbf{a}_i \wedge \mathbf{b}_i \rfloor \mathbf{a} \wedge \mathbf{b}) \\ &= \sum\left[(\mathbf{a}_i \cdot \mathbf{a})(\mathbf{b}_i \cdot \mathbf{b}) - (\mathbf{a}_i \cdot \mathbf{b})(\mathbf{b}_i \cdot \mathbf{a})\right] \\ &= \sum\left[\mathbf{a}^T\mathbf{a}_i\mathbf{b}_i^T\mathbf{b} - \mathbf{a}^T\mathbf{b}_i\mathbf{a}_i^T\mathbf{b}\right] \\ &= \mathbf{a}^T\left[\sum \mathbf{a}_i\mathbf{b}_i^T - \mathbf{b}_i\mathbf{a}_i^T\right]\mathbf{b}. \end{aligned} \tag{A.2}$$

Let A be the anti-symmetric matrix $\sum(\mathbf{a}_i\mathbf{b}_i^T - \mathbf{b}_i\mathbf{a}_i^T)$, and let σ be the largest singular value of A. Then $\mathbf{a}^T A\mathbf{b} \leq \|A\mathbf{b}\| \leq \sigma$. Conversely, let $\mathbf{a}$ and $\mathbf{b}$ be left and right singular vectors corresponding to σ, in which case $\sigma\mathbf{b}^T\mathbf{a} = \mathbf{a}^T A\mathbf{a} = 0$ (since A is anti-symmetric), and therefore $\mathbf{a}$ and $\mathbf{b}$ are orthonormal with $\mathbf{a}^T A\mathbf{b} = \sigma$. It follows that this choice of $\mathbf{a}$ and $\mathbf{b}$ will produce an optimal tangent plane.

A.1.2 Accounting for probabilities

So far we have been fitting the plane on the assumption that animation space is Euclidean. However, in Section 3.3.1 (page 20) we spent considerable effort in devising a non-Euclidean norm

that estimates corresponding lengths in 3D space. The norm is defined as $\|\mathbf{p}\|_{2,2} = \mathbf{p}^T P \mathbf{p}$, where P is a non-negative definite symmetric matrix. Normalised animation space, on the other hand, was defined in Section 3.3.2 as a space whose Euclidean norm was equivalent to the $L_{2,2}$ norm of animation space. If $\mathbf{p}$ is an element of animation space, then the equivalent point in normalised animation space is $C\mathbf{p}$, for some matrix C satisfying $C^T C = P$.

Conceptually, we could solve the plane-fitting problem in normalised animation space, then transform the solution back to standard animation space. Doing this directly would be quite inefficient though, as C may be a dense matrix, so we first apply some algebra.

If we let $\mathbf{u} = C\mathbf{a}$, $\mathbf{v} = C\mathbf{b}$ then the objective function becomes

$$\begin{aligned} D &= \mathbf{u}^T \left[\sum (C\mathbf{a}_i)(C\mathbf{b}_i)^T - (C\mathbf{b}_i)(C\mathbf{a}_i)^T \right] \mathbf{v} \\ &= \mathbf{u}^T C \left[\sum \mathbf{a}_i \mathbf{b}_i^T - \mathbf{b}_i \mathbf{a}_i^T \right] C^T \mathbf{v} \\ &= \mathbf{u}^T (CAC^T) \mathbf{v} \end{aligned} \tag{A.3}$$

and the constraints become

$$\|\mathbf{u}\| = 1, \|\mathbf{v}\| = 1, \mathbf{u} \cdot \mathbf{v} = 0. \tag{A.4}$$

Using the same reasoning above, it follows that the optimal solution is obtained by choosing $\mathbf{u}$ and $\mathbf{v}$ as left and right singular vectors corresponding to the largest singular value of CAC^T. We still need to find solutions to $C\mathbf{a} = \mathbf{u}$ and $C\mathbf{b} = \mathbf{v}$. Fortunately,

$$\sigma \mathbf{u} = CAC^T \mathbf{v} \tag{A.5}$$

$$\sigma \mathbf{v} = (CAC^T)^T \mathbf{u} = -CAC^T \mathbf{u} \tag{A.6}$$

and hence we can define

$$\mathbf{a} = \frac{AC^T \mathbf{v}}{\sigma} \tag{A.7}$$

$$\mathbf{b} = -\frac{AC^T \mathbf{u}}{\sigma}. \tag{A.8}$$

Up to this point, we have ignored three crucial questions:

1. Are the computed tangents guaranteed to have zero weight? Tangents with non-zero weight would be problematic, since they would alter under global translations of the model.
2. Are the tangents sparse? Common sense tells us that any bones with no effect on the input planes should also not affect the fitted plane, but does the algorithm guarantee this?
3. Can the algorithm be optimised for sparsity? As presented, we would have to compute the matrix CAC^T, which is potentially dense even if A is sparse.

As we will show, the answer to all three questions is yes. Suppose $\mathbf{w}$ is an element of the null space of A i.e., $A\mathbf{w} = \mathbf{0}$. Note that since A is anti-symmetric, $\mathbf{w}^T A = (A^T \mathbf{w})^T = -(A\mathbf{w})^T = \mathbf{0}^T$

as well. It follows that

$$\mathbf{w} \cdot \mathbf{a} = \mathbf{w}^T A C^T \mathbf{v}/\sigma = 0 \tag{A.9}$$

$$\mathbf{w} \cdot \mathbf{b} = -\mathbf{w}^T A C^T \mathbf{v}/\sigma = 0. \tag{A.10}$$

In other words, the fitted tangent plane is orthogonal to the null space of A.

Since $A = \sum(\mathbf{a}_i \mathbf{b}_i^T - \mathbf{b}_i \mathbf{a}_i^T)$, it immediately follows that whenever $\mathbf{x} \cdot \mathbf{a}_i = \mathbf{x} \cdot \mathbf{b}_i = 0$ for each input plane, we will also have $\mathbf{x} \cdot \mathbf{a} = \mathbf{x} \cdot \mathbf{b} = 0$. But the equation describing a zero-weight vector has precisely this form, so the fitted tangents will have zero weight provided that the input tangents have zero weight (which should always be the case).

Similarly, if there is a bone with no effect on the input planes (i.e., the corresponding elements of the tangents are all zero), then the same must apply to $\mathbf{a}$ and $\mathbf{b}$. This guarantees that $\mathbf{a}$ and $\mathbf{b}$ will be reasonably sparse if the set of input planes is. Furthermore, since we know in advance that $\mathbf{a}$ and $\mathbf{b}$ will be empty in particular elements, we can eliminate the corresponding rows and columns from CAC^T before we even compute the SVD. Indeed, we need not even compute the full value of CAC^T, only the submatrix corresponding to indices that may appear in $\mathbf{a}$ and $\mathbf{b}$.

A.1.3 Other measures of distance

Linear algebra also has a notion of distances between subspaces, which is not the same as the one defined above. For completeness, we provide a comparison here. We deal only with the Euclidean version of the problem.

Between two 2D subspaces (planes), there are two *principle angles*, $0 \leq \theta_1 \leq \theta_2 \leq \pi/2$. These are the minimum and maximum angles between vectors within the two planes. Let the planes be $\mathbf{a}_1 \wedge \mathbf{b}_1$ and $\mathbf{a}_2 \wedge \mathbf{b}_2$, with each basis being orthonormal. Golub and Van Loan [1996, p. 604] show that the cosines of the principle angles are the singular values of the matrix

$$\begin{pmatrix} \mathbf{a}_1 \cdot \mathbf{a}_2 & \mathbf{a}_1 \cdot \mathbf{b}_2 \\ \mathbf{b}_1 \cdot \mathbf{a}_2 & \mathbf{b}_1 \cdot \mathbf{b}_2 \end{pmatrix}. \tag{A.11}$$

Our fit metric is the determinant of this matrix, so it equals $\pm \cos\theta_1 \cos\theta_2$. In contrast, linear algebra defines the distance to be $\sin\theta_2$. Note that geometric algebra, unlike linear algebra, considers planes to be oriented, so the principle angles are insufficient to compute the geometric inner product (hence the $\pm$ sign). This orientation is important for our applications, which is why we have used geometric algebra to formulate our metric.

A.2 Constrained least squares

In this section we describe the implementation we used to solve a number of constrained least-squares problems. Each problem is a generalisation of the previous one, and also uses the previous

ones to solve the more general version. The most general version is used in Section 6.3 to select optimal positions when performing influence simplifications.

All the problems have in common an $m \times n$ matrix A and an m-vector $\mathbf{b}$, and the linear constraint $A\mathbf{x} = \mathbf{b}$. In every case we require that $\operatorname{rank} A = m$ i.e., that the constraints are independent. For our application this is always the case, but in fact this is not a critical restriction. If the constraints are not independent then they are either inconsistent or redundant. In the first case there can be no solutions, and in the second the redundant constraints can be eliminated by computing a basis for the constraints.

In the following derivations, we will use a number of concepts from computational linear algebra without reference. Golub and Van Loan [1996] provide a thorough guide to the theory of matrix computations, including all the techniques used here. The computations were implemented using the Meschach linear algebra library [Stewart and Leyk, 1994].

A.2.1 Unweighted case

The first problem we consider is that of minimising $\|\mathbf{x}\|^2$, subject to $A\mathbf{x} = \mathbf{b}$.

The Gram-Schmidt process gives the decomposition $LA = Q$ where L is an invertible lower triangular matrix and $QQ^T = I$ (the fact that L is invertible is equivalent to A having full rank). The given constraint is thus equivalent to $Q\mathbf{x} = L\mathbf{b}$. Let $\mathbf{x} = Q^T L\mathbf{b} + \mathbf{y}$; the linear constraint then forces $Q\mathbf{y} = \mathbf{0}$ and so

$$\|\mathbf{x}\|^2 = \|Q^T L\mathbf{b}\|^2 + \|\mathbf{y}\|^2.$$

Clearly, this is minimised when $\mathbf{y} = \mathbf{0}$ and so $\mathbf{x} = Q^T L\mathbf{b}$.

A.2.2 Positive weights

We now consider a generalisation of the problem to the weighted case. Specifically, the problem is to minimise $\sum \lambda_i x_i^2$ subject to a linear constraint $A\mathbf{x} = \mathbf{b}$, with the same limitations on A as before, and $\lambda_i > 0$ for each i.

Let $S = \operatorname{diag}(\lambda_1^{\frac{1}{2}}, \ldots, \lambda_m^{\frac{1}{2}})$, $\mathbf{y} = S\mathbf{x}$ and $B = AS^{-1}$. The problem can now be rewritten as minimising $\|\mathbf{y}\|^2$, subject to the constraint $B\mathbf{y} = \mathbf{b}$. This can be solved using the method for the unweighted case discussed previously; the solution to the original problem is then retrieved from the equation $\mathbf{x} = S^{-1}\mathbf{y}$. Note that since the columns of A are only scaled by non-zero values, their span remains the same and so the rank of A is not reduced.

A.2.3 Non-negative weights

The previous derivation breaks down if any of the weights are zero, because the diagonal matrix S becomes non-invertible. Without loss of generality, assume that the weights are ordered so that

all positive weights precede all zero weights. We can then make the partitions

$$A = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \qquad \text{and} \qquad \mathbf{x} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}$$

where the first part corresponds to the non-zero weights and the second to the zero weights. Next, compute the QR factorisation $A_2 = Q_2 R_2$. The original constraint may now be re-written as $Q_2^T A\mathbf{x} = Q_2^T \mathbf{b}$, which is equivalent to $Q_2^T A_1 \mathbf{x}_1 + R_2 \mathbf{x}_2 = Q_2^T \mathbf{b}$. We can now separate these constraints into two groups: those corresponding to non-zero rows of R_2, and those corresponding to zero rows of R_2 and hence only constraining $\mathbf{x}_1$.

At this point we solve for $\mathbf{x}_1$, taking into account only the latter set of constraints, using the technique from the previous subsection (recall that $\mathbf{x}_1$ contains the variables with the non-zero weights). Since the weights for $\mathbf{x}_2$ are all zero, this will be the optimal solution for the full problem, provided that we can solve for $\mathbf{x}_2$ to satisfy the remaining constraints in the equation

$$R_2 \mathbf{x}_2 = Q_2^T \mathbf{b} - Q_2^T A_1 \mathbf{x}_1.$$

But R_2 is upper triangular, so this can always be done by backward substitution. In practice, we minimise $\|\mathbf{x}_2\|^2$ subject to this linear constraint, in order to handle the case of a rank-deficient R_2 in a predicable manner.

A.2.4 Matrix weight

The most general case we consider is to minimise $\mathbf{x}^T P \mathbf{x}$ where P is non-negative definite symmetric, subject to the constraint $A\mathbf{x} = \mathbf{b}$, with the same requirements on A as before.

We start by computing the diagonalisation $P = U \Lambda U^T$, where U is orthogonal and all the eigenvalues are non-negative (this is guaranteed by the conditions on P). Let $\hat{A} = AU$ and $\hat{\mathbf{x}} = U^T \mathbf{x}$, so that the problem can be restated as minimising $\hat{\mathbf{x}}^T \Lambda \hat{\mathbf{x}} = \sum \lambda_i \hat{x}_i^2$ subject to the constraint $\hat{A}\hat{\mathbf{x}} = \mathbf{b}$. This is the weighted case that can be solved as before, and the solution to the original problem may be extracted as $\mathbf{x} = U\hat{\mathbf{x}}$.

Appendix B

Proofs

B.1 Incremental constraints

When doing influence simplification, a vertex will be simplified several times. Rather than always starting with the original position and solving a constrained least-squares problem to eliminate a set of influences, we instead take the previously reduced position and eliminate a single influence. This is cheaper to compute, and the following theorem proves that it gives the same result.

Theorem 1. *Given are a non-negative definite symmetric matrix P and two consistent sets of constraints $A_1\mathbf{x} = \mathbf{b}_1$ and $A_2\mathbf{x} = \mathbf{b}_2$. Let $A\mathbf{x} = \mathbf{b}$ be the equation that results from combining the two sets of constraints (i.e., redundancies between the sets are eliminated). Let*

$$\mathbf{x}_1 \text{ minimise } \mathbf{x}_1^T P\mathbf{x}_1 \text{ subject to } A_1\mathbf{x}_1 = \mathbf{b}_1$$
$$\mathbf{x}_2 \text{ minimise } \mathbf{x}_2^T P\mathbf{x}_2 \text{ subject to } A\mathbf{x}_2 = \mathbf{b}.$$

Then amongst all vectors $\mathbf{x}$ satisfying $A\mathbf{x} = \mathbf{b}$, $\mathbf{x}_2$ minimises $(\mathbf{x} - \mathbf{x}_1)^T P(\mathbf{x} - \mathbf{x}_1)$.

In other words, applying one set of constraints to get $\mathbf{x}_1$, then solving for the minimum correction to $\mathbf{x}_1$ to additionally satisfy a second set, gives an equivalent result to finding a minimum vector to satisfy both sets in one step. Note that the solutions are not necessarily identical if the minima are not unique.

Proof. Consider any $\mathbf{x}$ satisfying $A\mathbf{x} = \mathbf{b}$, and let $\mathbf{x} = \mathbf{x}_2 + \mathbf{y}$. Then $A\mathbf{y} = \mathbf{0}$ which implies that $A_1(\mathbf{x}_1 + \mathbf{y}) = \mathbf{b}_1$. From the definitions of $\mathbf{x}_1$ and $\mathbf{x}_2$,

$$\mathbf{x}_1^T P\mathbf{x}_1 \leq (\mathbf{x}_1 - \mathbf{y})^T P(\mathbf{x}_1 - \mathbf{y}) = \mathbf{x}_1^T P\mathbf{x}_1 + (\mathbf{y} - 2\mathbf{x}_1)^T P\mathbf{y} \tag{B.1}$$
$$\iff \quad 0 \leq (\mathbf{y} - 2\mathbf{x}_1)^T P\mathbf{y} \tag{B.2}$$

and

$$\mathbf{x}_2^T P\mathbf{x}_2 \leq (\mathbf{x}_2+\mathbf{y})^T P(\mathbf{x}_2+\mathbf{y}) = \mathbf{x}_2^T P\mathbf{x}_2 + (\mathbf{y}+2\mathbf{x}_2)^T P\mathbf{y} \tag{B.3}$$

$$\iff \quad 0 \leq (\mathbf{y}+2\mathbf{x}_2)^T P\mathbf{y}. \tag{B.4}$$

It follows that $0 \leq (\mathbf{y}+2(\mathbf{x}_2-\mathbf{x}_1))^T P\mathbf{y}$, and so

$$(\mathbf{x}-\mathbf{x}_1)^T P(\mathbf{x}-\mathbf{x}_1) = (\mathbf{y}+(\mathbf{x}_2-\mathbf{x}_1))^T P(\mathbf{y}+(\mathbf{x}_2-\mathbf{x}_1)) \tag{B.5}$$

$$= (\mathbf{x}_2-\mathbf{x}_1)^T P(\mathbf{x}_2-\mathbf{x}_1) + (\mathbf{y}+2(\mathbf{x}_2-\mathbf{x}_1))^T P\mathbf{y} \tag{B.6}$$

$$\geq (\mathbf{x}_2-\mathbf{x}_1)^T P(\mathbf{x}_2-\mathbf{x}_1). \tag{B.7}$$

Since $\mathbf{x}_2$ also satisfies the linear constraint on $\mathbf{x}$, the result follows. □

B.2 Expected value of $E[R^T MR]$

Theorem 2. *Let M be a fixed symmetric 3×3 matrix and let R be a 3×3 random rotation matrix with a uniform distribution. Then*

$$E[R^T MR] = \frac{\operatorname{tr} M}{3} I. \tag{B.8}$$

Proof. Let the singular values of $E[R^T MR]$ be $\sigma_1 \geq \sigma_2 \geq \sigma_3$. Let v_1 and v_3 be unit-length right singular vectors corresponding to σ_1 and σ_3, and let A be a fixed rotation matrix such that $Av_1 = v_3$. The Haar criterion [Shoemake, 1992] requires that RA has the same distribution as R, so

$$\sigma_1 = \|E[R^T MR]v_1\| = \|E[A^T R^T MRA]v_1\| = \|A^T E[R^T MR]Av_1\| = \|E[R^T MR]v_3\| = \sigma_3. \tag{B.9}$$

It follows that the singular values of $E[R^T MR]$ are all equal and hence it has the form σI (since it is also symmetric).

The trace operator is invariant under similarly transforms and is also a linear operator, so

$$\operatorname{tr} E[R^T MR] = E[\operatorname{tr} R^T MR] = E[\operatorname{tr} M] = \operatorname{tr} M. \tag{B.10}$$

The result follows since $\operatorname{tr} \sigma I = 3\sigma$. □

B.3 Under-determined coordinates in a simple skeleton

Theorem 3. *If the joints in a b-bone model have fixed translations relative to their parents, then there exist at least $b-1$ independent vectors in animation space that animation projection will always map to a zero vector in model space.*

This seems somewhat surprising: it says that in such models, animation-space vectors are not well defined. In particular, it says that although we found a way to convert an SSD model to an equivalent animation-space model (see Section 3.1), it is not unique. It does make sense when considering degrees of freedom: each joint has three degrees of freedom for rotation, but animation space has $4b$ dimensions.

Proof. Let $\mathbf{1} = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}^T$; we will use it frequently so it is useful to have a symbol.

Choose a particular bone $i \neq 0$, and let $\mathbf{p}$ be the animation-space position which is zero everywhere except for $\mathbf{p}_i = \mathbf{1}$ (i.e., a weight of 1 associated with bone i). Write the joint matrix L_i as $R_i T_i$, where R_i is a linear matrix (no translation) and T_i is a constant translation matrix. Then

$$\begin{aligned} G\mathbf{p} &= \sum_{j=0}^{b-1} G_j \mathbf{p}_j \\ &= G_i \mathbf{p}_i \\ &= G_{\phi(i)} L_i \mathbf{p}_i \\ &= G_{\phi(i)} R_i T_i \mathbf{p}_i \\ &= G_{\phi(i)} R_i \begin{pmatrix} \underline{T_i} \\ 1 \end{pmatrix} \\ &= G_{\phi(i)} R_i \begin{pmatrix} \underline{T_i} \\ 0 \end{pmatrix} + G_{\phi(i)} R_i \mathbf{1} \\ &= G_{\phi(i)} R_i T_i \begin{pmatrix} \underline{T_i} \\ 0 \end{pmatrix} + G_{\phi(i)} \mathbf{1} \\ &= G_i \begin{pmatrix} \underline{T_i} \\ 0 \end{pmatrix} + G_{\phi(i)} \mathbf{1} \\ &= G\mathbf{q}, \end{aligned} \tag{B.11}$$

where

$$\mathbf{q}_{\phi(i)} = \mathbf{1}, \mathbf{q}_i = \begin{pmatrix} \underline{T_i} \\ 0 \end{pmatrix}.$$

It follows that the vector $\mathbf{r} = \mathbf{p} - \mathbf{q}$ satisfies $G\mathbf{r} = \mathbf{0}$ for all G, and is non-zero because $\underline{\mathbf{r}_i} = 1$.

Now if we apply this to each bone i in turn, we can generate a set of vectors $\mathbf{r}^1, \mathbf{r}^2, \ldots, \mathbf{r}^{b-1}$ with the required properties. It remains only to show that they are independent. Without loss of generality, assume that the bones are numbered in a top-down manner; in particular that $\phi(i) < i$ for all i. Then for any $i < j$, $\mathbf{r}^i_j = \mathbf{0}$ and hence $\mathbf{r}^j$ is independent of the previous vectors $\mathbf{r}^1, \ldots, \mathbf{r}^{j-1}$. □

In fact the proof shows that the result can be strengthened: the weights of an animation vector can be chosen arbitrarily, as long as the total weight is kept constant. This can be achieved by adding appropriate multiples of the $\mathbf{r}^i$s, in decreasing order of i. Because $\mathbf{r}^i$ only affects bones i and $\phi(i)$ (and $\phi(i) < i$), the later adjustments do not disturb those already made.

Appendix C

Shader code

Listing C.1: `vpcommon.cg`: common structures and variables for vertex shaders. The `IGeometryVP` interface is responsible for computing the position, tangents and vertex normal, while `IShaderVP` is for any per-vertex colouring. Per-vertex colouring was used for debugging purposes, such as a shader that assigned a different colour to each parametrisation chart.

```
1  #ifndef VPCOMMON_CG
2  #define VPCOMMON_CG
3
4  #if defined(VERTEX_TANGENTS) || defined(VERTEX_NORMALS)
5  #define VERTEX_DERIVATIVES
6  #endif
7
8  uniform const float u_flod = 0.0;   // fractional part of the LOD level
9
10 struct OutputData
11 {
12     float4 position : POSITION;
13 #ifdef TEXCOORDS
14     float2 texcoords : TEXCOORD0;
15 #endif
16 #ifdef VERTEX_NORMALS
17     float3 normal : TEXCOORD1;
18 #endif
19 #ifdef VERTEX_TANGENTS
20     float3 tangents[2] : TEXCOORD2;
21 #endif
22     float4 color : COLOR;   // alpha carries flod
23 };
24
25 interface IGeometryVP
26 {
27     void process(inout OutputData o);
28 };
29
30 interface IShaderVP
```

```
31 {
32     float4 color();
33 };
34
35 #endif
```

Listing C.2: `fpcommon.cg`: common structures and variables for fragment shaders. An `INormalFP` produces a normal vector, which is then passed to an `ILightingFP`, which determines the light level. The `IShaderFP` interface is used for shaders that compute a colour on the surface.

```
1  #ifndef FPCOMMON_CG
2  #define FPCOMMON_CG
3
4  uniform half u_light_ambient;
5  uniform half u_light_diffuse;
6  uniform half3 u_light_direction;
7  uniform half u_flod = 0.0;
8
9  struct InputData
10 {
11 #ifdef TEXCOORDS
12     float2 texcoords : TEXCOORD0;
13 #endif
14 #ifdef VERTEX_NORMALS
15     float3 normal : TEXCOORD1;
16 #endif
17 #ifdef VERTEX_TANGENTS
18     float3 tangents[2] : TEXCOORD2;
19 #endif
20     half4 color : COLOR;      // alpha carries flod
21 };
22
23 interface IShaderFP
24 {
25     half4 color(const InputData id);
26 };
27
28 interface INormalFP
29 {
30     half3 normal(const InputData id);
31 };
32
33 interface ILightingFP
34 {
35     half4 luminance(const half3 normal);
36 };
37
38 #endif
```

Listing C.3: `vertex.cg`: driver shader for vertex processing.

```
1 #include "vpcommon.cg"
```

```
2  #include "geometry.cg"
3  #include "geometry_texture.cg"
4  #include "patches.cg"
5  #include "influences.cg"
6
7  uniform IGeometryVP geometry;
8  uniform IShaderVP shaders[];
9
10 void main(out OutputData o)
11 {
12     geometry.process(o);
13     if (shaders.length)
14     {
15         o.color = shaders[0].color();
16         for (int i = 1; i < shaders.length; i++)
17             o.color *= shaders[i].color();
18     }
19     else
20         o.color.xyz = 1.0;
21     o.color.w = u_flod;
22 }
```

Listing C.4: `vertex.cg`: driver shader for fragment processing.

```
1  #include "fpcommon.cg"
2  #include "normal_vertex.cg"
3  #include "normal_static.cg"
4  #include "normal_dynamic.cg"
5  #include "normal_null.cg"
6  #include "lighting_dot.cg"
7  #include "lighting_cube.cg"
8  #include "lighting_normal.cg"
9  #include "isolines.cg"
10 #include "distance.cg"
11 #include "decal.cg"
12
13 uniform IShaderFP shaders[];
14 uniform INormalFP normal;
15 uniform ILightingFP lighting;
16
17 void main(const InputData id,
18           out half4 color : COLOR)
19 {
20     color = id.color;
21     for (int i = 0; i < shaders.length; i++)
22         color *= shaders[i].color(id);
23     color *= lighting.luminance(normal.normal(id));
24     color.w = 1.0;
25 }
```

Listing C.5: `geometry.cg`: vertex program for pre-transformed vertices. This program is used for the CPU-based vertex transformation (Section 7.4.1) and the rendering pass of the fragment

program-based vertex transformation (Section 7.4.2). It applies the model-view-projection transformation, computes the vertex normal and interpolates texture coordinates between levels of detail.

```
#include "vpcommon.cg"

uniform float4x4 mvp : state.matrix.mvp;
uniform float4x4 mv : state.matrix.modelview;
uniform float4x4 imv : state.matrix.modelview.invtrans;

varying float4 i_position : POSITION;
#ifdef VERTEX_DERIVATIVES
varying float3 i_tangents[2];
#endif
#ifdef TEXCOORDS
varying float4 i_texcoords;
#endif

struct GeometryVP : IGeometryVP
{
    void process(inout OutputData o)
    {
        o.position = mul(mvp, i_position);
#ifdef VERTEX_TANGENTS
        o.tangents[0] = normalize(mul(float3x3(mv), i_tangents[0]));
        o.tangents[1] = normalize(mul(float3x3(mv), i_tangents[1]));
#endif

#ifdef VERTEX_NORMALS
#ifdef VERTEX_TANGENTS
        o.normal = normalize(cross(o.tangents[0], o.tangents[1]));
#else
        o.normal = normalize(mul(float3x3(imv),
                                 cross(i_tangents[0], i_tangents[1])));
#endif
#endif

#ifdef TEXCOORDS
        o.texcoords = i_texcoords.xy + u_flod * i_texcoords.zw;
#endif
    }
};
```

Listing C.6: `rtvb_frag_fp.cg`: the fragment program for fragment program-based vertex transformation (Section 7.4.2). It applies animation projection to the position and optionally the tangents, writing either one or three outputs.

```
// REPS is #defined to be either 1 (positions only) or 3 (positions and tangents)

void main(half3 tc : TEXCOORD0,
          out half4 col[REPS] : COLOR0,
          uniform samplerRECT bone_tex,
```

```
6             uniform samplerRECT transform_tex,
7             uniform samplerRECT inf_tex[REPS][2],
8             uniform half flod)
9 {
10     half4 ans[REPS];
11     half4 bc = half4(0.5, 1.5, 2.5, 0.0);
12
13     for (int i = 0; i < REPS; i++)
14         ans[i] = half4(0.0h, 0.0h, 0.0h, 1.0h);
15
16     // tc.z holds the maximum coordinate
17     for (; tc.y < tc.z; tc.y++)
18     {
19         bc.w = h4texRECT(bone_tex, tc.xy).x * 255.0h + 0.5h;
20         half3x4 T = half3x4(h4texRECT(transform_tex, bc.xw),
21                             h4texRECT(transform_tex, bc.yw),
22                             h4texRECT(transform_tex, bc.zw));
23
24         for (int r = 0; r < REPS; r++)
25             ans[r].xyz += mul(T, h4texRECT(inf_tex[r][0], tc.xy)
26                                  + flod * h4texRECT(inf_tex[r][1], tc.xy));
27     }
28     for (int r = 0; r < REPS; r++)
29         col[r] = ans[r];
30 }
```

Listing C.7: geometry_texture.cg: vertex program for vertex-texturing vertex transformation (Section 7.4.3).

```
1  #include "vpcommon.cg"
2
3  varying float4 i_offsets : POSITION;
4
5  #ifdef VERTEX_DERIVATIVES
6  #define CHANNELS 3
7  #else
8  #define CHANNELS 1
9  #endif
10 uniform sampler2D bones;
11 uniform sampler2D influences[CHANNELS];
12 uniform float3x4 u_transforms[80];       // bone transformations
13 uniform float u_xstep;                   // 1 / texture width
14
15 struct GeometryTextureVP : IGeometryVP
16 {
17     void process(inout OutputData o)
18     {
19         // x: first s coordinate  y: t coordinate for base
20         // z: t coordinate for delta  w: last s coordinate
21         float4 offsets = i_offsets;
22
23         float3 coords[CHANNELS];
```

```
24          for (int c = 0; c < CHANNELS; c++)
25              coords[c] = 0.0;
26          while (offsets.x < offsets.w)
27          {
28              float bone = tex2D(bones, offsets.xy).x;
29              float4 inf[CHANNELS][2];
30              for (int c = 0; c < CHANNELS; c++)
31              {
32                  inf[c][0] = tex2D(influences[c], offsets.xy);
33                  inf[c][1] = tex2D(influences[c], offsets.xz);
34              }
35              for (int c = 0; c < CHANNELS; c++)
36                  coords[c] += mul(u_transforms[(int) bone],
37                                   inf[c][0] + u_flod * inf[c][1]);
38              offsets.x += u_xstep;
39          }
40          o.position = mul(mvp, float4(coords[0], 1.0));
41  #ifdef VERTEX_TANGENTS
42          o.tangents[0] = normalize(mul(float3x3(mv), coords[1]));
43          o.tangents[1] = normalize(mul(float3x3(mv), coords[2]));
44  #endif
45
46  #ifdef VERTEX_NORMALS
47  #ifdef VERTEX_TANGENTS
48          o.normal = normalize(cross(o.tangents[0], o.tangents[1]));
49  #else
50          o.normal = normalize(mul(float3x3(imv), cross(coords[1], coords[2])));
51  #endif
52  #endif
53
54  #ifdef TEXCOORDS
55          o.texcoords = i_texcoords.xy + u_flod * i_texcoords.zw;
56  #endif
57      }
58  };
```

Listing C.8: `dnm_fp.cg`: normals from tangent maps, pre-computation pass. The shader applies animation projection to two tangent vectors, as described in Section 7.5.1, then takes their cross product to determine a normal. The corresponding vertex program is not shown, as it simply passes through the position and texture coordinates.

```
1   uniform samplerRECT bone_tex;
2   uniform samplerRECT tangent_tex[2];
3   uniform samplerRECT transform_tex;
4   uniform float3x3 imv : state.matrix.modelview.invtrans;
5
6   void main(float3 tc : TEXCOORD0, out half4 color : COLOR)
7   {
8       half3 tangents[2];
9       half4 bc = half4(0.5, 1.5, 2.5, 0.0);
10
11      for (int i = 0; i < 2; i++)
```

```
12          tangents[i] = half3(0.0h, 0.0h, 0.0h);
13
14      for (; tc.x < tc.z; tc.x++)
15      {
16          bc.w = h4texRECT(bone_tex, tc.xy).x * 255.0h + 0.5h;
17          half3x4 T = half3x4(h4texRECT(transform_tex, bc.xw),
18                              h4texRECT(transform_tex, bc.yw),
19                              h4texRECT(transform_tex, bc.zw));
20
21          for (int i = 0; i < 2; i++)
22              tangents[i] += mul(T, h4texRECT(tangent_tex[i], tc.xy));
23      }
24
25      color.xyz = 0.5h * normalize(mul(float3x3(imv), cross(tangents[0], tangents[1])))
26                  + 0.5h;
27  }
```

Listing C.9: `normal_dynamic.cg`: normals from tangent maps, rendering pass. This shader simply extracts the normal that was computed during the first pass, and remaps the $[0, 1]$ colour range to a $[-1, 1]$ coordinate range.

```
1   #include "fpcommon.cg"
2
3   #ifdef TEXCOORDS
4
5   struct NormalDynamicFP : INormalFP
6   {
7       uniform sampler2D normal_texture;
8       half3 normal(const InputData id)
9       {
10          return 2.0h * h4tex2D(normal_texture, id.texcoords.xy).xyz - 1.0h;
11      }
12  };
13
14  #endif
```

Listing C.10: `normal_static.cg`: tangent-space normal maps. This shader interpolates the tangent-space normal between discrete levels of detail, then transforms it from tangent space to a global space.

```
1   #include "fpcommon.cg"
2
3   #if defined(VERTEX_TANGENTS) && defined(VERTEX_NORMALS) && defined(TEXCOORDS)
4
5   struct NormalStaticFP : INormalFP
6   {
7       uniform sampler2D normal_map_a; // from higher-res LOD
8       uniform sampler2D normal_map_b; // from lower-res LOD
9
10      half3 normal(const InputData id)
11      {
```

```
12          half3 local_normal_a = h4tex2D(normal_map_a, id.texcoords).xyz;
13          half3 local_normal_b = h4tex2D(normal_map_b, id.texcoords).xyz;
14          // color.w carries flod (fractional part of LOD)
15          half3 local_normal = lerp(local_normal_a, local_normal_b, id.color.w);
16          // map [0, 1] colours to [-1, 1] coordinates
17          local_normal = 2.0h * local_normal - 1.0h;
18          // set up tangent-space space for transformation to model space
19          half3x3 trans = half3x3(id.normal, id.tangents[0], id.tangents[1]);
20          return normalize(mul(local_normal, trans));
21      }
22  };
23
24  #endif
```

Listing C.11: `lighting_dot.cg`: simple dot-product lighting. There is one light in the positive Z direction and another in a user-specified direction, as well as an ambient term. The `saturate` function is a Cg built-in that clamps the argument to $[0, 1]$; on some platforms, this is a hardware operation which is cheaper than calculating `max(x, 0)`.

```
1   #include "fpcommon.cg"
2
3   struct LightingDotFP : ILightingFP
4   {
5       half4 luminance(const half3 normal)
6       {
7           return saturate(dot(normal, u_light_direction)) * u_light_diffuse
8                + saturate(normal.z) * u_light_diffuse
9                + u_light_ambient;
10      }
11  };
```

Listing C.12: `ssd.vpm`: low-level SSD shader used to generate Table 8.9. Note that the shader is run through a macro pre-processor to generate the actual assembly code, to create a specialisation that is tuned for a specific maximum number of influences

```
1   include(`shaders.m4')dnl
2   !!ARBvp1.0
3
4   ATTRIB iRest = vertex.attrib[0];
5   frombone(0, `ATTRIB iBonesA = vertex.attrib[1];')
6   frombone(0, `ATTRIB iWeightsA = vertex.attrib[2];')
7   frombone(4, `ATTRIB iBonesB = vertex.attrib[3];')
8   frombone(4, `ATTRIB iWeightsB = vertex.attrib[4];')
9   PARAM mvp[4] = { state.matrix.mvp };
10  PARAM matrices[parameters] = { program.env[0..lastparameter] };
11  ADDRESS index;
12  TEMP local, pos;
13  OUTPUT oPos = result.position;
14  OUTPUT oCol = result.color;
15
16  frombone(0, `
```

```
17 ARL index.x, iBonesA.x;
18 DP4 local.x, matrices[index.x + 0], iRest;
19 DP4 local.y, matrices[index.x + 1], iRest;
20 DP4 local.z, matrices[index.x + 2], iRest;
21 MUL pos.xyz, local, iWeightsA.x;
22 ')
23
24 frombone(1, '
25 ARL index.x, iBonesA.y;
26 DP4 local.x, matrices[index.x + 0], iRest;
27 DP4 local.y, matrices[index.x + 1], iRest;
28 DP4 local.z, matrices[index.x + 2], iRest;
29 MAD pos.xyz, local, iWeightsA.y, pos;
30 ')
31
32 frombone(2, '
33 ARL index.x, iBonesA.z;
34 DP4 local.x, matrices[index.x + 0], iRest;
35 DP4 local.y, matrices[index.x + 1], iRest;
36 DP4 local.z, matrices[index.x + 2], iRest;
37 MAD pos.xyz, local, iWeightsA.z, pos;
38 ')
39
40 frombone(3, '
41 ARL index.x, iBonesA.w;
42 DP4 local.x, matrices[index.x + 0], iRest;
43 DP4 local.y, matrices[index.x + 1], iRest;
44 DP4 local.z, matrices[index.x + 2], iRest;
45 MAD pos.xyz, local, iWeightsA.w, pos;
46 ')
47
48 frombone(4, '
49 ARL index.x, iBonesB.x;
50 DP4 local.x, matrices[index.x + 0], iRest;
51 DP4 local.y, matrices[index.x + 1], iRest;
52 DP4 local.z, matrices[index.x + 2], iRest;
53 MAD pos.xyz, local, iWeightsB.x, pos;
54 ')
55
56 frombone(5, '
57 ARL index.x, iBonesB.y;
58 DP4 local.x, matrices[index.x + 0], iRest;
59 DP4 local.y, matrices[index.x + 1], iRest;
60 DP4 local.z, matrices[index.x + 2], iRest;
61 MAD pos.xyz, local, iWeightsB.y, pos;
62 ')
63
64 frombone(6, '
65 ARL index.x, iBonesB.z;
66 DP4 local.x, matrices[index.x + 0], iRest;
67 DP4 local.y, matrices[index.x + 1], iRest;
68 DP4 local.z, matrices[index.x + 2], iRest;
```

```
69 MAD pos.xyz, local, iWeightsB.z, pos;
70 ')
71
72 frombone(7, `
73 ARL index.x, iBonesB.w;
74 DP4 local.x, matrices[index.x + 0], iRest;
75 DP4 local.y, matrices[index.x + 1], iRest;
76 DP4 local.z, matrices[index.x + 2], iRest;
77 MAD pos.xyz, local, iWeightsB.w, pos;
78 ')
79
80 DPH oPos.x, pos, mvp[0];
81 DPH oPos.y, pos, mvp[1];
82 DPH oPos.z, pos, mvp[2];
83 DPH oPos.w, pos, mvp[3];
84 MOV oCol, 1.0;
85 END
```

Listing C.13: `ssd.vpm`: low-level animation-space shader used to generate Table 8.9. Note that the shader is run through a macro pre-processor to generate the actual assembly code, to create a specialisation that is tuned for a specific maximum number of influences.

```
1  include(`shaders.m4')dnl
2  !!ARBvp1.0
3
4  frombone(0, `ATTRIB iBonesA = vertex.attrib[0];')
5  frombone(0, `ATTRIB P0 = vertex.attrib[1];')
6  frombone(1, `ATTRIB P1 = vertex.attrib[2];')
7  frombone(2, `ATTRIB P2 = vertex.attrib[3];')
8  frombone(3, `ATTRIB P3 = vertex.attrib[4];')
9  frombone(4, `ATTRIB iBonesB = vertex.attrib[5];')
10 frombone(4, `ATTRIB P4 = vertex.attrib[6];')
11 frombone(5, `ATTRIB P5 = vertex.attrib[7];')
12 frombone(6, `ATTRIB P6 = vertex.attrib[8];')
13 frombone(7, `ATTRIB P7 = vertex.attrib[9];')
14 PARAM mvp[4] = { state.matrix.mvp };
15 PARAM matrices[parameters] = { program.env[0..lastparameter] };
16 ADDRESS index;
17 TEMP local, pos;
18 OUTPUT oPos = result.position;
19 OUTPUT oCol = result.color;
20
21 frombone(0, `
22 ARL index.x, iBonesA.x;
23 DP4 pos.x, matrices[index.x + 0], P0;
24 DP4 pos.y, matrices[index.x + 1], P0;
25 DP4 pos.z, matrices[index.x + 2], P0;
26 ')
27
28 frombone(1, `
29 ARL index.x, iBonesA.y;
30 DP4 local.x, matrices[index.x + 0], P1;
```

```
31 DP4 local.y, matrices[index.x + 1], P1;
32 DP4 local.z, matrices[index.x + 2], P1;
33 ADD pos.xyz, pos, local;
34 ')
35
36 frombone(2, '
37 ARL index.x, iBonesA.z;
38 DP4 local.x, matrices[index.x + 0], P2;
39 DP4 local.y, matrices[index.x + 1], P2;
40 DP4 local.z, matrices[index.x + 2], P2;
41 ADD pos.xyz, pos, local;
42 ')
43
44 frombone(3, '
45 ARL index.x, iBonesA.w;
46 DP4 local.x, matrices[index.x + 0], P3;
47 DP4 local.y, matrices[index.x + 1], P3;
48 DP4 local.z, matrices[index.x + 2], P3;
49 ADD pos.xyz, pos, local;
50 ')
51
52 frombone(4, '
53 ARL index.x, iBonesB.x;
54 DP4 local.x, matrices[index.x + 0], P4;
55 DP4 local.y, matrices[index.x + 1], P4;
56 DP4 local.z, matrices[index.x + 2], P4;
57 ADD pos.xyz, pos, local;
58 ')
59
60 frombone(5, '
61 ARL index.x, iBonesB.y;
62 DP4 local.x, matrices[index.x + 0], P5;
63 DP4 local.y, matrices[index.x + 1], P5;
64 DP4 local.z, matrices[index.x + 2], P5;
65 ADD pos.xyz, pos, local;
66 ')
67
68 frombone(6, '
69 ARL index.x, iBonesB.z;
70 DP4 local.x, matrices[index.x + 0], P6;
71 DP4 local.y, matrices[index.x + 1], P6;
72 DP4 local.z, matrices[index.x + 2], P6;
73 ADD pos.xyz, pos, local;
74 ')
75
76 frombone(7, '
77 ARL index.x, iBonesB.w;
78 DP4 local.x, matrices[index.x + 0], P7;
79 DP4 local.y, matrices[index.x + 1], P7;
80 DP4 local.z, matrices[index.x + 2], P7;
81 ADD pos.xyz, pos, local;
82 ')
```

```
83
84 DPH oPos.x, pos, mvp[0];
85 DPH oPos.y, pos, mvp[1];
86 DPH oPos.z, pos, mvp[2];
87 DPH oPos.w, pos, mvp[3];
88 MOV oCol, 1.0;
89 END
```

Listing C.14: `ssd.vpm`: low-level MWE shader used to generate Table 8.9. Note that the shader is run through a macro pre-processor to generate the actual assembly code, to create a specialisation that is tuned for a specific maximum number of influences.

```
1  include(‘shaders.m4’)dnl
2  !!ARBvp1.0
3
4  frombone(0, ‘ATTRIB iBonesA = vertex.attrib[0];’)
5  frombone(0, ‘ATTRIB iWeights00 = vertex.attrib[1];’)
6  frombone(0, ‘ATTRIB iWeights01 = vertex.attrib[2];’)
7  frombone(0, ‘ATTRIB iWeights02 = vertex.attrib[3];’)
8  frombone(1, ‘ATTRIB iWeights10 = vertex.attrib[4];’)
9  frombone(1, ‘ATTRIB iWeights11 = vertex.attrib[5];’)
10 frombone(1, ‘ATTRIB iWeights12 = vertex.attrib[6];’)
11 frombone(2, ‘ATTRIB iWeights20 = vertex.attrib[7];’)
12 frombone(2, ‘ATTRIB iWeights21 = vertex.attrib[8];’)
13 frombone(2, ‘ATTRIB iWeights22 = vertex.attrib[9];’)
14 frombone(3, ‘ATTRIB iWeights30 = vertex.attrib[10];’)
15 frombone(3, ‘ATTRIB iWeights31 = vertex.attrib[11];’)
16 frombone(3, ‘ATTRIB iWeights32 = vertex.attrib[12];’)
17 PARAM mvp[4] = { state.matrix.mvp };
18 PARAM matrices[parameters] = { program.env[0..lastparameter] };
19 ADDRESS index;
20 TEMP local, pos;
21 OUTPUT oPos = result.position;
22 OUTPUT oCol = result.color;
23
24 frombone(0, ‘
25 ARL index.x, iBonesA.x;
26 DP4 pos.x, iWeights00, matrices[index.x + 0];
27 DP4 pos.y, iWeights01, matrices[index.x + 1];
28 DP4 pos.z, iWeights02, matrices[index.x + 2];
29 ’)
30
31 frombone(1, ‘
32 ARL index.x, iBonesA.y;
33 DP4 local.x, iWeights10, matrices[index.x + 0];
34 DP4 local.y, iWeights11, matrices[index.x + 1];
35 DP4 local.z, iWeights12, matrices[index.x + 2];
36 ADD pos.xyz, pos, local;
37 ’)
38
39 frombone(2, ‘
40 ARL index.x, iBonesA.z;
```

```
41 DP4 local.x, iWeights20, matrices[index.x + 0];
42 DP4 local.y, iWeights21, matrices[index.x + 1];
43 DP4 local.z, iWeights22, matrices[index.x + 2];
44 ADD pos.xyz, pos, local;
45 ')
46
47 frombone(3, '
48 ARL index.x, iBonesA.w;
49 DP4 local.x, iWeights30, matrices[index.x + 0];
50 DP4 local.y, iWeights31, matrices[index.x + 1];
51 DP4 local.z, iWeights32, matrices[index.x + 2];
52 ADD pos.xyz, pos, local;
53 ')
54
55 DPH oPos.x, pos, mvp[0];
56 DPH oPos.y, pos, mvp[1];
57 DPH oPos.z, pos, mvp[2];
58 DPH oPos.w, pos, mvp[3];
59 MOV oCol, 1.0;
60 END
```

Appendix D

Raw experiment data

Table D.1: Raw data from user experiment 1. A • indicates that the user chose the full-detail model and a ∘ that the user chose the LOD approximation. The two values in each cell are for the two orders in which the videos could be shown.

Scene	Subject ID	Pixel tolerance											
		0.5	1	1.5	2	2.5	3	3.5	4	5	6	7	8
arm	1	∘ ∘	∘ •	• •	∘ ∘	• ∘	∘ •	∘ •	• •	• •	• ∘	• •	∘ •
	2	∘ •	∘ •	∘ •	∘ •	∘ •	∘ •	• ∘	• •	• •	• •	• •	∘ •
	3	• ∘	• ∘	∘ •	∘ •	∘ •	• •	• ∘	• •	• ∘	• •	• •	• •
	6	∘ •	∘ •	• •	∘ •	∘ ∘	• ∘	• ∘	• ∘	• •	∘ ∘	• •	∘ •
	7	• ∘	• •	• •	• •	• •	• ∘	• ∘	• ∘	∘ •	• •	• •	• •
	8	∘ •	• •	∘ •	• •	• •	• •	• •	• •	• •	• •	• •	• •
	9	• ∘	• •	• •	• ∘	• ∘	• •	• •	• •	• •	• ∘	• •	• •
	10	• ∘	• ∘	∘ •	• ∘	∘ •	∘ ∘	• ∘	∘ ∘	• •	∘ ∘	• •	• ∘
	11	• ∘	∘ ∘	• •	• •	• •	• •	• ∘	• •	∘ •	• •	• •	• •
	12	∘ ∘	∘ •	∘ •	• ∘	• •	• ∘	• ∘	• •	• ∘	• ∘	• •	∘ ∘
	13	∘ •	• ∘	• ∘	• ∘	• •	• •	• •	∘ ∘	∘ ∘	• •	∘ ∘	• ∘
	14	∘ ∘	∘ ∘	• ∘	∘ •	• ∘	• •	∘ •	∘ •	∘ •	∘ •	∘ ∘	• ∘
	15	• ∘	• ∘	∘ ∘	• •	• ∘	• •	• •	• ∘	• •	• •	• •	• •
	16	• •	∘ •	• •	• ∘	• ∘	• ∘	• ∘	∘ ∘	• ∘	• ∘	• ∘	• ∘
	17	• •	• ∘	∘ ∘	∘ •	• ∘	• •	• ∘	∘ ∘	∘ •	∘ ∘	∘ ∘	• ∘
	18	∘ ∘	• ∘	∘ •	∘ •	• •	• ∘	• •	• •	∘ ∘	∘ •	∘ •	• •
	19	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	∘ ∘	• ∘	• ∘	• ∘	• ∘
	Total (/34)	15	18	21	20	22	23	22	20	22	20	25	23
cat	1	∘ •	∘ ∘	• •	∘ •	• •	• •	∘ •	∘ •	∘ •	• •	• •	• •
	2	• •	∘ •	• •	• ∘	• ∘	∘ •	• •	• •	• •	• •	• •	• •
	3	• ∘	• •	• •	• •	∘ ∘	• •	• •	∘ •	• •	• ∘	• •	• •
	6	• •	∘ ∘	• •	∘ •	• •	• ∘	• ∘	∘ ∘	• ∘	∘ •	• ∘	• ∘
	7	• ∘	• •	• ∘	∘ •	• •	• ∘	• •	• ∘	• ∘	• ∘	• ∘	• ∘
	8	∘ •	∘ ∘	• •	• •	∘ •	• •	• ∘	• •	• •	• •	• •	• •
	9	∘ •	∘ •	• ∘	• ∘	• ∘	• •	• •	• •	• •	∘ •	• •	• •
	10	• ∘	• ∘	• •	∘ ∘	• •	• ∘	∘ •	∘ ∘	• ∘	• ∘	• •	∘ ∘
	11	• •	∘ ∘	∘ •	∘ •	∘ •	• ∘	• ∘	∘ •	∘ •	∘ ∘	• •	• ∘
	12	∘ •	• ∘	• •	∘ ∘	• ∘	• ∘	• •	∘ •	• •	• •	• •	• •
	13	• ∘	• ∘	• •	• •	• ∘	• ∘	• •	• ∘	• ∘	• •	• •	• •
	14	∘ ∘	• ∘	∘ •	∘ •	∘ ∘	• •	• •	• •	∘ •	∘ •	∘ •	• ∘
	15	• •	• ∘	∘ •	• ∘	• ∘	• •	• •	• ∘	• ∘	• •	• •	• •
	16	• ∘	• •	∘ ∘	• •	∘ •	• ∘	∘ •	• •	∘ ∘	• ∘	• ∘	• •
	17	• •	• •	• •	∘ •	• •	∘ ∘	∘ •	∘ •	∘ •	• ∘	• •	• ∘
	18	• ∘	• ∘	• ∘	• •	• ∘	• ∘	• ∘	• ∘	∘ ∘	• •	• •	• ∘
	19	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• •	• ∘	• ∘	• •	• ∘	• ∘
	Total (/34)	21	17	25	20	22	23	27	21	21	25	28	24
cyberdemon	1	• ∘	∘ ∘	• •	• ∘	• ∘	∘ ∘	∘ •	• ∘	∘ ∘	• ∘	∘ ∘	∘ ∘
	2	∘ •	• •	∘ ∘	∘ •	∘ ∘	• ∘	∘ •	• ∘	• •	• ∘	• •	• •
	3	• ∘	∘ •	• •	∘ ∘	∘ ∘	∘ •	• ∘	• ∘	∘ •	• ∘	∘ •	• •
	6	• •	∘ •	∘ ∘	• •	• •	• ∘	• ∘	• ∘	• ∘	• •	• •	∘ •
	7	• ∘	• •	• •	• •	• ∘	• ∘	∘ ∘	∘ •	• ∘	∘ ∘	• ∘	• ∘
	8	• •	∘ •	∘ •	∘ •	∘ •	∘ ∘	∘ •	∘ •	∘ •	∘ •	• •	• •
	9	• ∘	∘ ∘	• ∘	• •	• ∘	• ∘	• ∘	• ∘	• ∘	• •	∘ •	• •
	10	• ∘	• •	• •	∘ ∘	∘ •	• ∘	• •	∘ •	• ∘	• ∘	∘ ∘	• ∘
	11	• ∘	• ∘	∘ •	• ∘	∘ ∘	• •	∘ •	∘ •	∘ •	• •	• •	∘ •
	12	∘ •	• ∘	∘ •	• ∘	∘ •	∘ •	∘ •	∘ •	• ∘	• ∘	• ∘	• •
	13	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	∘ ∘	• ∘	• ∘	• ∘	• ∘	• ∘
	14	• ∘	• •	• •	∘ •	∘ •	• ∘	∘ ∘	• ∘	• ∘	• ∘	∘ •	∘ ∘
	15	∘ •	• •	∘ •	∘ ∘	• ∘	• •	• •	• •	• •	• ∘	∘ ∘	∘ •
	16	∘ ∘	• ∘	• ∘	∘ •	∘ •	• •	• ∘	• ∘	• •	• ∘	∘ •	• •
	17	• ∘	• ∘	• •	∘ ∘	• ∘	∘ ∘	∘ ∘	∘ ∘	• ∘	• •	• •	• ∘
	18	∘ •	∘ •	• •	• •	∘ •	• ∘	∘ ∘	∘ ∘	• •	• ∘	• •	∘ ∘
	19	• •	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• •
	Total (/34)	19	19	21	18	14	17	14	16	21	18	20	19
horse	1	• •	∘ •	• •	• •	• •	• •	• •	• ∘	∘ •	• •	∘ •	• •
	2	∘ •	∘ •	• •	• •	∘ ∘	• •	• •	• •	• •	∘ •	• •	• •
	3	• ∘	∘ •	• •	• •	∘ •	• •	• ∘	• •	• •	• •	• •	• •
	6	• ∘	• •	• ∘	∘ ∘	∘ ∘	• ∘	• ∘	∘ ∘	∘ ∘	• ∘	∘ ∘	• •
	7	∘ ∘	∘ ∘	• •	• •	∘ ∘	• ∘	• •	• •	• •	• •	• •	• •
	8	• ∘	• ∘	∘ •	∘ •	∘ ∘	• •	∘ •	∘ •	• •	• •	• •	∘ •
	9	∘ •	• •	• •	∘ •	∘ ∘	• •	• ∘	• ∘	• •	• •	• •	• •
	10	• ∘	∘ •	∘ •	∘ ∘	• ∘	• •	∘ ∘	∘ ∘	∘ •	• •	• •	• ∘
	11	∘ ∘	∘ ∘	∘ •	• •	• •	• ∘	∘ •	• •	∘ •	• •	• •	• •
	12	• ∘	• ∘	∘ ∘	∘ ∘	• ∘	∘ •	∘ •	∘ •	• ∘	• •	∘ •	• •
	13	• ∘	• ∘	• ∘	• ∘	• •	• ∘	• •	• •	• •	• •	• •	• •
	14	∘ ∘	∘ •	• •	• •	• •	• ∘	• ∘	• ∘	• •	∘ ∘	∘ •	• •
	15	∘ ∘	∘ •	• •	• •	∘ ∘	∘ ∘	• •	• ∘	• •	• •	• ∘	• ∘
	16	∘ •	∘ •	∘ •	∘ •	∘ •	• •	∘ ∘	∘ •	• •	∘ •	∘ •	∘ •
	17	∘ •	∘ •	• •	∘ ∘	• •	∘ •	∘ ∘	∘ ∘	∘ •	• ∘	• •	• ∘
	18	∘ ∘	∘ ∘	∘ ∘	∘ ∘	∘ ∘	∘ ∘	• •	∘ •	∘ ∘	• ∘	∘ •	• •
	19	∘ ∘	• ∘	∘ ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• •	∘ ∘	• ∘	∘ ∘
	Total (/34)	12	16	23	19	17	22	20	19	25	25	26	27
mancandy	1	∘ •	• ∘	∘ ∘	∘ •	• •	∘ •	• •	• •	• •	• ∘	• •	∘ •
	2	∘ •	∘ ∘	∘ •	∘ ∘	∘ •	• •	∘ •	• ∘	• ∘	• •	• •	• ∘
	3	• ∘	• •	• •	• ∘	• •	∘ ∘	• •	• ∘	• ∘	• ∘	∘ ∘	∘ •
	6	• •	• ∘	• •	• •	∘ •	• •	• ∘	• ∘	∘ •	• •	∘ ∘	∘ •
	7	• ∘	• ∘	• •	• ∘	• •	• •	∘ •	• •	• •	• ∘	• •	• •
	8	• •	∘ •	• ∘	• ∘	∘ ∘	∘ •	• ∘	• •	∘ •	∘ •	• •	• ∘
	9	∘ ∘	• ∘	• ∘	∘ •	• ∘	∘ ∘	• ∘	• •	• ∘	• ∘	∘ •	• •
	10	∘ ∘	• •	∘ ∘	∘ ∘	• ∘	∘ ∘	∘ ∘	• ∘	• ∘	• ∘	• •	∘ ∘
	11	• •	• ∘	• ∘	• •	• ∘	• •	• •	• •	• •	∘ •	• •	• •
	12	• ∘	∘ •	∘ ∘	• ∘	∘ ∘	• •	• •	• ∘	• ∘	• ∘	∘ •	∘ ∘
	13	∘ ∘	• ∘	• ∘	• ∘	• ∘	• •	∘ ∘	• ∘	∘ ∘	∘ ∘	• ∘	∘ ∘
	14	• ∘	• •	• ∘	• •	∘ ∘	• •	• ∘	• ∘	• ∘	∘ ∘	• •	• ∘
	15	∘ ∘	∘ ∘	• •	∘ •	∘ ∘	∘ ∘	∘ ∘	• •	• •	• ∘	∘ ∘	• •
	16	∘ ∘	∘ ∘	• •	∘ ∘	∘ ∘	∘ ∘	• •	∘ •	∘ ∘	∘ ∘	∘ ∘	∘ ∘
	17	∘ ∘	• •	∘ •	• ∘	∘ ∘	∘ •	∘ ∘	∘ •	∘ •	∘ ∘	• •	• ∘
	18	• •	• •	• •	∘ •	• •	∘ ∘	• ∘	∘ ∘	∘ •	• ∘	• •	• ∘
	19	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘
	Total (/34)	15	19	20	17	16	19	20	22	19	15	22	17
Total (/170)		82	89	110	94	91	104	103	98	108	103	121	110

Table D.2: Raw data from user experiment 2. A • indicates that the user chose the animation-space simplification and a ∘ that the user chose the 3D simplification. The grey cells are from the training phase and are not included in the totals.

Pixel tolerance	Subject ID	Scene					Total
		arm	cat	cyberdemon	horse	mancandy	
1.5	2	● • •	● • •	∘ ∘ •	∘ ∘ ∘	● • •	7/10
	3	● • ∘	● • ∘	● • •	∘ ∘ •	∘ ∘ •	6/10
	4	● ∘ •	∘ • •	● ∘ ∘	∘ • •	∘ ∘ •	6/10
	6	● ∘ •	● ∘ ∘	∘ ∘ ∘	∘ • •	● • •	5/10
	7	∘ • •	● • ∘	∘ • ∘	● • ∘	∘ • •	8/10
	8	● • •	● ∘ •	∘ ∘ •	● • •	∘ ∘ ∘	6/10
	9	∘ ∘ ∘	● • ∘	● • •	∘ ∘ ∘	∘ ∘ •	4/10
	12	● • •	● ∘ •	∘ ∘ •	● • •	∘ ∘ •	7/10
	13	∘ • •	∘ • ∘	● • ∘	● ∘ ∘	● ∘ •	5/10
	15	∘ • •	∘ • •	∘ ∘ ∘	∘ ∘ •	● • •	7/10
	16	● ∘ •	● • •	● • •	● ∘ ∘	● ∘ ∘	5/10
	17	● • ∘	∘ • •	∘ • ∘	● • •	● • •	8/10
	18	● ∘ •	● • ∘	∘ ∘ •	● ∘ •	∘ ∘ ∘	4/10
	19	∘ ∘ •	● ∘ ∘	∘ • ∘	● • •	● • •	6/10
	20	● • ∘	● ∘ •	● ∘ ∘	● • ∘	∘ ∘ •	4/10
	22	● • •	∘ • ∘	∘ • •	● • •	● ∘ •	8/10
	23	∘ ∘ ∘	∘ ∘ •	∘ • ∘	● ∘ •	● • ∘	4/10
	24	● • ∘	∘ • ∘	● • •	∘ • ∘	∘ • •	7/10
	26	● • •	∘ • •	● ∘ ∘	● • ∘	∘ ∘ •	6/10
	27	∘ • •	∘ ∘ •	● ∘ •	∘ ∘ •	∘ • •	7/10
	28	∘ ∘ •	∘ • •	∘ • •	∘ • ∘	∘ • •	8/10
	29	∘ ∘ •	● • •	● ∘ •	∘ • ∘	∘ • ∘	6/10
	30	● ∘ •	∘ • ∘	∘ ∘ •	● • ∘	● • •	6/10
		30/46	30/46	24/46	26/46	30/46	140/230
3	2	● ∘ ∘	● ∘ •	∘ ∘ •	∘ • •	● • •	6/10
	3	● ∘ ∘	● ∘ •	● ∘ ∘	● ∘ •	● ∘ ∘	2/10
	4	∘ ∘ ∘	∘ ∘ •	∘ ∘ •	∘ ∘ •	● ∘ •	4/10
	6	● • ∘	∘ ∘ •	● ∘ ∘	∘ ∘ •	● • ∘	4/10
	7	● ∘ ∘	● ∘ ∘	∘ • ∘	● ∘ •	● ∘ ∘	2/10
	8	∘ • •	● ∘ •	● ∘ ∘	● • ∘	● • •	6/10
	9	∘ ∘ •	● ∘ •	● • ∘	● • ∘	∘ ∘ •	5/10
	12	∘ ∘ •	● ∘ •	∘ ∘ •	∘ ∘ •	● ∘ •	5/10
	13	∘ ∘ ∘	● ∘ ∘	∘ ∘ •	● • ∘	● ∘ ∘	2/10
	15	∘ • •	∘ • •	● ∘ •	● • •	∘ ∘ •	8/10
	16	● ∘ ∘	∘ ∘ •	● ∘ •	● ∘ •	∘ ∘ •	4/10
	17	∘ • ∘	● • ∘	∘ ∘ •	∘ • ∘	● • •	6/10
	18	∘ ∘ ∘	∘ ∘ ∘	● • •	∘ • •	∘ ∘ •	5/10
	19	● • •	● ∘ ∘	∘ ∘ •	∘ ∘ ∘	∘ ∘ •	4/10
	20	∘ • •	● • ∘	∘ ∘ •	∘ • •	● ∘ •	7/10
	22	● • •	∘ • •	● ∘ ∘	● ∘ ∘	∘ ∘ ∘	4/10
	23	∘ ∘ •	● • •	∘ ∘ •	∘ • ∘	● ∘ ∘	5/10
	24	∘ • •	● • ∘	● • •	∘ ∘ •	∘ ∘ •	7/10
	26	● • •	● ∘ •	● • ∘	● • •	∘ ∘ •	7/10
	27	● ∘ •	∘ • ∘	∘ ∘ ∘	● ∘ •	● • •	5/10
	28	∘ ∘ •	● ∘ •	∘ ∘ •	● ∘ ∘	∘ ∘ •	4/10
	29	∘ ∘ ∘	● ∘ ∘	∘ ∘ •	● ∘ ∘	● • ∘	2/10
	30	● • ∘	∘ ∘ ∘	∘ ∘ •	● • ∘	● • ∘	4/10
		22/46	20/46	20/46	24/46	22/46	108/230
6	2	∘ • •	● ∘ ∘	∘ ∘ •	● • ∘	∘ ∘ •	5/10
	3	∘ • ∘	∘ • •	∘ ∘ •	∘ • •	● ∘ •	7/10
	4	∘ ∘ •	∘ • ∘	∘ • ∘	∘ ∘ •	● ∘ ∘	4/10
	6	∘ • •	∘ ∘ •	● • •	∘ • ∘	● • •	8/10
	7	● • •	● • ∘	∘ • ∘	● • •	∘ • ∘	7/10
	8	● • ∘	● • •	● ∘ •	● ∘ •	∘ ∘ ∘	5/10
	9	∘ • •	● ∘ ∘	∘ • ∘	● • ∘	∘ • •	6/10
	12	● ∘ •	● ∘ •	∘ ∘ •	∘ ∘ •	∘ ∘ •	5/10
	13	∘ • ∘	∘ • •	● ∘ ∘	● • ∘	● ∘ ∘	4/10
	15	● • •	∘ ∘ ∘	● ∘ ∘	● ∘ ∘	∘ ∘ •	3/10
	16	● • •	∘ • •	● ∘ •	● ∘ ∘	● ∘ ∘	5/10
	17	● ∘ ∘	● • •	● ∘ ∘	● ∘ ∘	∘ • •	4/10
	18	● ∘ •	∘ ∘ ∘	● ∘ •	● ∘ ∘	● ∘ ∘	2/10
	19	● • ∘	∘ ∘ ∘	● ∘ •	∘ ∘ ∘	∘ ∘ •	3/10
	20	● • ∘	● ∘ ∘	● • ∘	∘ ∘ ∘	● ∘ •	3/10
	22	∘ ∘ ∘	∘ ∘ •	∘ ∘ •	∘ ∘ •	● ∘ ∘	3/10
	23	● ∘ ∘	∘ • •	● ∘ ∘	● • •	● • ∘	5/10
	24	∘ ∘ ∘	● ∘ ∘	∘ • •	∘ ∘ ∘	● • ∘	3/10
	26	● • •	● ∘ ∘	∘ • •	● • •	● ∘ •	7/10
	27	∘ ∘ ∘	● • •	● ∘ •	∘ • •	∘ ∘ •	6/10
	28	∘ ∘ ∘	∘ ∘ •	● ∘ •	● ∘ •	∘ ∘ •	4/10
	29	∘ ∘ •	● ∘ ∘	∘ ∘ •	∘ • •	∘ • ∘	5/10
	30	∘ • •	∘ • ∘	∘ ∘ •	● • ∘	∘ ∘ •	6/10
		25/46	21/46	22/46	22/46	20/46	110/230
Total		77/138	71/138	66/138	72/138	72/138	358/690

Table D.3: Raw data from user experiment 3. A • indicates that the user chose the model that included influence simplification and a ○ that the user chose the model without influence simplification. The grey cells are from the training phase and are not included in the totals.

Pixel tolerance	Subject ID	Scene					Total
		arm	cat	cyberdemon	horse	mancandy	
1.5	2	● ○ ○	○ ○ ○	○ ○ ●	● ● ●	● ● ●	5/10
	3	○ ○ ●	● ○ ●	● ● ○	● ○ ●	● ● ●	6/10
	4	● ● ○	● ● ●	○ ● ●	● ● ○	○ ● ●	8/10
	6	● ● ○	○ ● ●	● ○ ○	● ○ ●	○ ● ○	5/10
	7	○ ● ○	● ● ○	○ ○ ○	● ● ○	● ● ●	5/10
	8	○ ○ ○	● ○ ○	○ ● ●	● ● ●	● ○ ○	4/10
	9	● ○ ●	○ ● ●	● ● ●	○ ● ●	● ● ●	9/10
	12	● ● ●	● ○ ●	○ ○ ●	○ ○ ●	○ ○ ●	6/10
	13	○ ● ●	● ○ ●	○ ● ○	● ○ ○	○ ● ●	6/10
	15	● ○ ○	● ○ ○	○ ● ●	● ○ ●	○ ○ ●	4/10
	16	○ ● ●	● ● ○	○ ○ ○	● ● ●	● ● ●	7/10
	17	● ● ○	○ ○ ○	● ● ○	○ ● ●	○ ○ ○	4/10
	18	● ○ ○	● ● ○	○ ○ ●	● ● ○	● ● ●	5/10
	19	○ ○ ●	● ● ●	● ● ○	● ○ ○	○ ● ●	6/10
	20	● ● ●	○ ● ○	○ ● ●	● ○ ●	● ● ●	8/10
	22	○ ○ ○	● ○ ○	● ● ○	○ ● ○	● ● ●	4/10
	23	● ○ ○	● ○ ●	● ● ●	● ○ ●	● ● ●	6/10
	24	● ○ ●	● ○ ○	○ ● ●	● ○ ○	○ ● ●	5/10
	26	○ ○ ○	○ ● ○	○ ○ ●	○ ● ○	○ ● ●	5/10
	27	● ● ●	○ ○ ●	○ ○ ●	○ ● ○	● ● ○	6/10
	28	● ○ ●	● ○ ●	○ ○ ●	○ ● ●	● ● ●	7/10
	29	● ○ ●	● ● ●	● ○ ●	● ● ○	● ● ○	6/10
	30	● ● ○	○ ● ○	● ○ ●	○ ● ○	● ○ ○	4/10
		21/46	22/46	27/46	26/46	35/46	131/230
3	2	● ○ ●	● ○ ○	● ○ ●	● ● ●		4/8
	3	○ ○ ●	● ● ○	● ● ○	○ ● ●		5/8
	4	○ ○ ○	● ● ○	● ○ ●	● ● ●		4/8
	6	○ ○ ●	● ○ ●	○ ● ○	○ ○ ●		4/8
	7	● ● ●	● ● ○	● ○ ○	○ ● ○		4/8
	8	○ ● ○	● ● ○	○ ○ ●	● ● ●		5/8
	9	● ● ●	● ○ ○	● ● ●	● ○ ●		5/8
	12	○ ○ ●	○ ○ ○	● ○ ●	● ○ ○		2/8
	13	● ○ ○	● ○ ●	○ ○ ○	● ● ●		3/8
	15	○ ○ ●	● ● ●	● ● ●	○ ○ ○		5/8
	16	● ● ●	● ● ●	○ ○ ○	● ● ●		6/8
	17	○ ● ○	● ● ○	● ● ○	● ○ ●		4/8
	18	● ● ○	● ● ○	● ● ●	● ● ●		6/8
	19	○ ○ ○	● ● ●	● ○ ●	● ○ ●		4/8
	20	● ○ ○	○ ○ ●	● ● ●	● ● ●		5/8
	22	○ ● ●	● ○ ●	● ● ○	○ ● ●		6/8
	23	○ ● ●	● ● ●	● ● ○	● ● ●		7/8
	24	○ ● ●	● ○ ○	● ○ ○	● ○ ●		3/8
	26	● ○ ○	● ● ●	○ ● ●	● ● ●		6/8
	27	○ ○ ●	○ ○ ●	● ● ●	○ ● ●		6/8
	28	● ○ ●	○ ● ●	○ ○ ○	● ● ●		5/8
	29	○ ○ ●	● ● ○	● ○ ○	● ● ●		4/8
	30	○ ● ●	● ● ○	○ ○ ●	● ● ○		5/8
		25/46	25/46	23/46	35/46		108/184
Total		46/92	47/92	50/92	61/92	35/46	239/414

Table D.4: Raw data from user experiment 4. A • indicates that the user chose the tangent-space normal map and a ○ that the user chose the tangent map. The grey cells are from the training phase and are not included in the totals.

Pixel tolerance	Subject ID	Scene					Total
		arm	cat	cyberdemon	horse	mancandy	
1.5	2	○ ○ ○	● ○ ●	○ ● ●	○ ○ ●	● ○ ○	4/10
	3	○ ● ○	● ● ●	○ ● ○	○ ● ●	● ○ ○	6/10
	4	○ ○ ●	○ ● ●	● ○ ○	○ ○ ○	● ○ ●	4/10
	6	○ ○ ○	● ○ ●	● ○ ○	○ ● ○	○ ● ●	4/10
	7	○ ● ●	● ● ○	○ ● ○	○ ● ○	● ● ○	6/10
	8	○ ● ○	● ● ○	○ ○ ○	○ ● ●	● ○ ●	5/10
	9	● ● ○	● ○ ○	○ ● ●	● ● ●	○ ○ ○	5/10
	12	● ○ ●	● ○ ●	● ○ ●	○ ○ ●	○ ○ ○	4/10
	13	○ ○ ○	● ○ ●	● ○ ○	○ ○ ○	○ ● ●	3/10
	15	○ ○ ●	● ○ ●	● ○ ●	○ ○ ●	● ○ ●	5/10
	16	● ○ ○	○ ○ ●	○ ○ ●	● ● ●	● ● ○	5/10
	17	○ ○ ●	● ● ○	● ○ ○	● ● ○	○ ○ ●	4/10
	18	● ● ○	○ ○ ○	○ ○ ●	○ ○ ○	● ○ ●	3/10
	19	○ ○ ○	● ○ ●	● ● ●	● ○ ●	● ○ ●	5/10
	20	● ● ○	○ ● ○	● ○ ●	○ ● ●	● ○ ●	6/10
	22	● ● ○	● ● ○	○ ● ○	● ● ●	○ ● ○	6/10
	23	● ● ○	○ ○ ○	● ● ○	○ ○ ●	● ○ ○	3/10
	24	○ ○ ●	● ○ ●	● ○ ●	○ ○ ○	● ● ●	5/10
	26	● ● ●	● ● ○	● ○ ●	○ ○ ●	○ ● ●	7/10
	27	● ○ ○	● ○ ●	○ ○ ●	○ ● ○	● ○ ●	4/10
	28	○ ○ ○	○ ○ ●	● ○ ●	○ ○ ●	● ○ ●	4/10
	29	○ ○ ●	○ ● ●	● ○ ●	● ○ ●	○ ● ○	6/10
	30	○ ○ ●	○ ● ●	● ○ ●	○ ● ○	● ● ●	7/10
		18/46	24/46	21/46	25/46	23/46	111/230
3	2	● ● ○	○ ○ ●	○ ● ●	○ ○ ○	● ○ ●	5/10
	3	● ○ ○	○ ● ○	● ● ○	● ○ ○	○ ● ●	4/10
	4	● ○ ●	● ○ ●	○ ○ ○	● ○ ●	○ ○ ●	4/10
	6	○ ○ ●	● ○ ○	○ ● ○	○ ○ ○	○ ● ○	3/10
	7	● ● ○	○ ● ○	○ ● ○	● ● ●	● ● ●	7/10
	8	● ● ○	○ ● ●	● ● ○	○ ○ ○	● ● ●	6/10
	9	● ● ○	● ● ○	○ ● ●	○ ● ●	○ ○ ●	7/10
	12	○ ○ ●	● ○ ●	● ○ ●	○ ○ ●	● ○ ●	5/10
	13	○ ○ ●	○ ● ○	○ ● ○	● ○ ○	○ ○ ●	4/10
	15	● ○ ●	● ○ ●	○ ○ ●	○ ○ ●	○ ○ ●	5/10
	16	● ● ○	○ ● ●	● ○ ●	● ● ●	● ● ●	8/10
	17	● ● ○	● ● ●	● ○ ○	○ ● ○	○ ● ○	5/10
	18	○ ○ ●	○ ○ ○	● ○ ●	● ○ ●	○ ● ●	5/10
	19	○ ● ○	● ○ ○	● ○ ●	○ ○ ●	○ ○ ○	3/10
	20	○ ○ ●	○ ○ ●	● ○ ○	● ○ ●	○ ○ ○	3/10
	22	● ● ○	● ○ ○	● ● ●	○ ○ ○	● ○ ○	3/10
	23	● ○ ●	○ ○ ●	● ● ●	● ○ ●	● ● ○	6/10
	24	● ○ ○	○ ○ ○	○ ○ ○	○ ● ●	○ ● ○	3/10
	26	○ ● ●	○ ○ ●	● ○ ●	○ ● ○	● ● ●	7/10
	27	● ○ ●	● ● ●	○ ○ ●	○ ● ○	● ○ ○	5/10
	28	○ ○ ●	○ ○ ●	● ○ ●	● ● ○	○ ○ ○	4/10
	29	● ● ●	○ ● ○	○ ○ ●	○ ○ ●	○ ● ○	6/10
	30	● ● ○	● ○ ○	● ○ ●	● ● ○	● ○ ●	4/10
		23/46	21/46	23/46	21/46	24/46	112/230
6	2	○ ● ●	○ ○ ●	○ ● ○	● ● ○	○ ○ ○	5/10
	3	● ● ●	● ○ ○	● ● ●	● ○ ○	○ ● ●	6/10
	4	○ ● ○	● ○ ○	○ ○ ○	○ ● ●	● ○ ●	4/10
	6	○ ○ ○	○ ○ ○	○ ● ○	○ ○ ●	○ ● ●	4/10
	7	○ ○ ●	● ● ○	○ ● ○	● ● ○	○ ● ○	5/10
	8	● ○ ○	● ○ ○	● ● ○	● ○ ●	● ● ●	4/10
	9	○ ● ●	● ● ○	○ ● ●	● ● ○	○ ● ●	8/10
	12	● ○ ●	○ ○ ●	○ ○ ●	○ ○ ●	● ○ ●	5/10
	13	● ● ○	● ● ○	○ ○ ○	○ ○ ○	● ○ ○	2/10
	15	○ ● ●	● ○ ●	● ○ ●	● ○ ●	● ○ ●	6/10
	16	● ● ○	○ ● ○	○ ○ ○	● ● ●	○ ● ○	5/10
	17	○ ○ ●	○ ● ○	● ○ ●	● ○ ●	○ ○ ●	5/10
	18	○ ● ○	○ ● ○	● ● ○	○ ○ ○	● ○ ●	4/10
	19	○ ● ○	○ ● ○	● ● ○	● ○ ●	○ ○ ●	5/10
	20	○ ● ○	● ● ○	○ ○ ○	● ○ ●	● ○ ●	4/10
	22	○ ● ○	● ● ●	○ ● ○	○ ● ●	○ ○ ●	7/10
	23	● ○ ●	○ ○ ○	● ● ○	● ○ ○	○ ● ○	3/10
	24	● ● ○	● ● ●	○ ● ○	○ ● ●	● ○ ○	6/10
	26	● ● ○	○ ● ●	○ ● ●	● ● ●	● ● ●	9/10
	27	○ ○ ●	● ○ ●	● ● ○	○ ○ ○	● ● ●	5/10
	28	● ○ ●	● ○ ●	○ ○ ●	○ ● ○	● ● ●	6/10
	29	○ ○ ●	○ ● ●	● ● ●	● ○ ○	○ ● ○	6/10
	30	○ ● ●	● ● ●	● ○ ●	● ● ○	● ○ ●	7/10
		26/46	23/46	23/46	22/46	27/46	121/230
Total		67/138	68/138	67/138	68/138	74/138	344/690

Table D.5: Raw data for vertex transformation performance on a GeForce 6600. The times reported for each of the four methods are the average number of milliseconds per frame. The linear models computed from these data are shown in Table 8.7. The vertex textures results based on mancandy were omitted from the table (and the linear model) because they exhausted available memory and led to swapping.

Objects	Vertices	Influences	CPU	Alternate CPU	Fragment Program	Vertex Textures	Objects	Vertices	Influences	CPU	Alternate CPU	Fragment Program	Vertex Textures
1	1764	3765	2.291	2.752	2.868	3.479	1	8211	18373	3.106	3.076	3.382	8.193
1	1059	1963	2.244	2.717	2.834	2.758	1	8211	9170	3.057	3.050	3.278	5.225
1	579	1022	2.226	2.699	2.784	2.439	1	8211	8211	3.053	3.046	3.247	4.891
1	340	537	2.220	2.690	2.744	2.290	1	8211	8211	3.051	3.046	3.245	4.893
1	217	279	2.213	2.683	2.739	2.205	1	8211	8211	3.059	3.047	3.247	4.891
1	128	132	2.209	2.681	2.729	2.198	1	8211	8211	3.057	3.050	3.245	4.891
100	176400	376500	13.701	11.111	21.927	136.583	100	821100	1837300	106.078	100.438	74.157	607.738
100	105900	196300	7.766	7.098	21.849	64.218	100	821100	917000	63.502	63.321	63.989	311.196
100	57900	102200	4.729	5.060	21.990	32.626	100	821100	821100	57.403	56.165	60.881	277.705
100	34000	53700	3.584	4.094	22.120	17.726	100	821100	821100	57.187	55.989	60.877	277.696
100	21700	27900	3.109	3.604	21.859	9.328	100	821100	821100	57.024	55.645	60.878	277.685
100	12800	13200	2.797	3.277	21.856	5.314	100	821100	821100	56.619	56.371	60.878	277.699
1	1764	3765	2.277	2.752	2.380	3.397	1	3025	4699	2.797	2.799	2.963	3.525
1	1764	1900	2.276	2.751	2.385	2.769	1	2220	2452	2.760	2.762	2.884	2.855
1	1764	1764	2.279	2.751	2.373	2.723	1	1321	1357	2.724	2.725	2.843	2.513
1	1764	1764	2.276	2.752	2.377	2.723	1	801	805	2.702	2.703	2.804	2.346
1	1764	1764	2.276	2.753	2.373	2.722	1	491	491	2.697	2.696	2.760	2.244
1	1764	1764	2.292	2.751	2.377	2.722	1	420	420	2.691	2.688	2.748	2.216
100	176400	376500	13.944	11.321	21.935	128.310	102	308550	479298	29.098	23.321	33.123	144.489
100	176400	190000	10.215	10.338	21.819	65.558	102	226440	250104	22.114	17.910	31.672	76.185
100	176400	176400	10.970	10.392	21.933	60.848	102	134742	138414	16.596	14.370	31.788	41.062
100	176400	176400	10.523	10.403	21.924	60.858	102	81702	82110	13.707	12.335	31.492	24.789
100	176400	176400	10.499	10.395	21.874	60.853	102	50082	50082	11.941	11.203	31.464	24.952
100	176400	176400	10.579	10.404	21.929	60.846	102	42840	42840	11.283	10.835	31.662	24.867
1	9495	24650	3.166	3.196	2.963	10.374	1	3025	4699	2.797	2.798	2.950	3.510
1	4648	12818	2.888	2.887	2.628	6.356	1	3025	3025	2.793	2.793	2.913	3.007
1	2532	6780	2.781	2.782	2.463	4.279	1	3025	3025	2.795	2.795	2.912	3.006
1	1517	3603	2.736	2.735	2.379	3.214	1	3025	3025	2.793	2.796	2.912	3.007
1	959	1962	2.712	2.711	2.363	2.693	1	3025	3025	2.793	2.795	2.912	3.006
1	638	1066	2.699	2.700	2.306	2.419	1	3025	3025	2.794	2.796	2.913	3.006
100	949500	2465000	128.792	130.377	79.371	826.481	102	308550	479298	29.091	23.374	32.151	142.388
100	464800	1281800	62.032	64.189	45.722	424.220	102	308550	308550	25.226	20.331	31.432	91.079
100	253200	678000	25.309	20.995	29.219	216.449	102	308550	308550	25.007	20.298	31.526	91.093
100	151700	360300	16.369	13.582	25.155	110.240	102	308550	308550	24.996	20.373	31.854	91.079
100	95900	196200	11.273	9.623	25.383	58.010	102	308550	308550	25.014	20.326	31.545	91.083
100	63800	106600	8.344	7.362	25.066	30.627	102	308550	308550	24.985	20.331	31.626	91.107
1	9495	24650	3.174	3.153	2.983	10.373	1	46569	111399	5.755	7.270	5.955	—
1	9495	12404	3.162	3.124	2.827	6.285	1	31581	56703	3.709	4.215	4.567	—
1	9495	9495	3.124	3.109	2.791	5.347	1	16503	29204	3.472	2.461	3.479	—
1	9495	9495	3.128	3.108	2.791	5.348	1	9287	15221	3.128	2.606	2.928	—
1	9495	9495	3.125	3.110	2.791	5.347	1	5588	8129	2.903	2.901	3.176	—
1	9495	9495	3.127	3.107	2.791	5.349	1	3498	4474	2.809	2.809	2.970	—
100	949500	2465000	127.227	130.904	80.986	825.827	100	4656900	11139900	649.640	844.061	407.856	—
100	949500	1240400	86.824	82.224	65.759	417.305	100	3158100	5670300	389.814	474.265	243.228	—
100	949500	949500	75.670	70.245	61.978	323.412	100	1650300	2920400	199.123	219.877	136.757	—
100	949500	949500	76.780	69.480	61.980	323.409	100	928700	1522100	127.983	127.011	92.041	—
100	949500	949500	75.581	69.831	68.749	323.412	100	558800	812900	73.509	73.861	65.189	—
100	949500	949500	76.157	69.990	68.748	323.409	100	349800	447400	57.083	48.196	56.807	—
1	8211	18373	3.106	3.087	3.393	8.168	1	46569	111399	5.411	7.306	6.900	—
1	4467	9727	2.878	2.877	3.101	5.268	1	46569	55839	4.898	5.170	5.284	—
1	2715	5111	2.786	2.784	2.943	3.728	1	46569	46569	4.652	5.030	5.062	—
1	1677	2676	2.742	2.742	2.868	2.944	1	46569	46569	4.713	5.011	5.881	—
1	1068	1430	2.717	2.714	2.830	2.559	1	46569	46569	4.636	5.949	5.879	—
1	695	802	2.701	2.700	2.800	2.369	1	46569	46569	4.754	4.920	5.879	—
100	821100	1837300	104.302	99.194	75.805	605.945	100	4656900	11139900	657.411	845.407	383.077	—
100	446700	972700	41.559	40.694	46.364	315.613	100	4656900	5583900	478.555	627.329	349.729	—
100	271500	511100	23.429	18.387	30.521	161.785	100	4656900	4656900	459.487	598.947	328.242	—
100	167700	267600	14.980	12.308	25.220	83.209	100	4656900	4656900	453.367	587.870	327.091	—
100	106800	143000	10.559	8.817	25.471	44.721	100	4656900	4656900	460.850	597.582	292.402	—
100	69500	80200	8.188	7.168	25.165	25.718	100	4656900	4656900	454.582	604.335	293.095	—

Table D.6: Raw data for vertex transformation performance on a GeForce 8800GTX. The times reported for each of the four methods are the average number of milliseconds per frame. The linear models computed from these data are shown in Table 8.8.

Objects	Vertices	Influences	CPU	Alternate CPU	Fragment Program	Vertex Textures	Objects	Vertices	Influences	CPU	Alternate CPU	Fragment Program	Vertex Textures
1	1764	3765	0.108	0.122	0.166	0.087	1	8211	18373	0.280	0.245	0.221	0.142
1	1059	1963	0.100	0.110	0.130	0.080	1	8211	9170	0.201	0.195	0.219	0.113
1	579	1022	0.095	0.109	0.129	0.078	1	8211	8211	0.188	0.195	0.216	0.110
1	340	537	0.092	0.109	0.129	0.078	1	8211	8211	0.220	0.195	0.216	0.110
1	217	279	0.091	0.109	0.129	0.077	1	8211	8211	0.188	0.195	0.216	0.110
1	128	132	0.091	0.109	0.129	0.076	1	8211	8211	0.218	0.195	0.216	0.110
100	176400	376500	5.671	4.788	9.469	1.545	100	821100	1837300	25.871	20.854	14.378	6.652
100	105900	196300	3.584	2.993	9.435	1.561	100	821100	917000	17.664	13.832	14.194	4.760
100	57900	102200	2.269	1.977	9.439	1.548	100	821100	821100	16.539	12.994	13.890	4.705
100	34000	53700	1.722	1.460	9.487	1.568	100	821100	821100	16.343	12.984	13.892	4.705
100	21700	27900	1.423	1.224	9.541	1.563	100	821100	821100	16.381	12.971	13.880	4.705
100	12800	13200	1.144	1.123	9.455	1.544	100	821100	821100	16.322	12.955	13.879	4.705
1	1764	3765	0.108	0.121	0.138	0.087	1	3025	4699	0.128	0.135	0.158	0.093
1	1764	1900	0.108	0.121	0.138	0.081	1	2220	2452	0.133	0.122	0.158	0.090
1	1764	1764	0.108	0.122	0.137	0.081	1	1321	1357	0.113	0.114	0.158	0.089
1	1764	1764	0.107	0.122	0.137	0.080	1	801	805	0.097	0.110	0.158	0.123
1	1764	1764	0.108	0.121	0.137	0.081	1	491	491	0.094	0.110	0.159	0.090
1	1764	1764	0.109	0.121	0.137	0.081	1	420	420	0.093	0.109	0.158	0.117
100	176400	376500	5.608	4.841	9.476	1.576	102	308550	479298	11.783	9.750	13.921	7.358
100	176400	190000	4.224	3.555	9.473	1.577	102	226440	250104	8.729	7.523	13.950	7.328
100	176400	176400	4.029	3.309	9.404	1.570	102	134742	138414	6.637	6.092	13.836	7.385
100	176400	176400	4.037	3.296	9.543	1.542	102	81702	82110	5.545	5.293	13.907	7.298
100	176400	176400	4.029	3.302	9.429	1.541	102	50082	50082	4.905	4.801	13.934	7.340
100	176400	176400	4.040	3.303	9.472	1.554	102	42840	42840	4.689	4.640	13.882	7.342
1	9495	24650	0.339	0.301	0.244	0.179	1	3025	4699	0.127	0.134	0.157	0.093
1	4648	12818	0.233	0.182	0.184	0.129	1	3025	3025	0.141	0.134	0.158	0.089
1	2532	6780	0.128	0.125	0.153	0.100	1	3025	3025	0.141	0.134	0.158	0.119
1	1517	3603	0.104	0.116	0.140	0.085	1	3025	3025	0.120	0.134	0.157	0.091
1	959	1962	0.099	0.111	0.140	0.080	1	3025	3025	0.120	0.134	0.157	0.090
1	638	1066	0.095	0.109	0.139	0.078	1	3025	3025	0.120	0.134	0.158	0.093
100	949500	2465000	31.865	26.489	16.705	10.183	102	308550	479298	11.831	9.630	13.912	7.403
100	464800	1281800	17.639	14.500	10.759	5.094	102	308550	308550	9.548	8.357	13.868	7.333
100	253200	678000	10.304	8.341	10.588	3.131	102	308550	308550	9.537	8.338	13.886	7.341
100	151700	360300	6.577	5.344	10.598	3.140	102	308550	308550	9.588	8.395	13.902	7.590
100	95900	196200	4.466	3.757	10.620	3.160	102	308550	308550	9.525	8.369	13.945	7.390
100	63800	106600	3.314	2.799	10.597	3.136	102	308550	308550	9.524	8.430	13.968	7.338
1	9495	24650	0.342	0.301	0.245	0.175	1	46569	111399	1.603	1.696	0.730	0.658
1	9495	12404	0.241	0.221	0.234	0.121	1	31581	56703	0.872	0.860	0.544	0.381
1	9495	9495	0.211	0.207	0.230	0.119	1	16503	29204	0.486	0.413	0.345	0.247
1	9495	9495	0.211	0.208	0.230	0.119	1	9287	15221	0.293	0.250	0.244	0.194
1	9495	9495	0.211	0.207	0.230	0.119	1	5588	8129	0.194	0.172	0.192	0.148
1	9495	9495	0.210	0.209	0.230	0.119	1	3498	4474	0.140	0.141	0.172	0.131
100	949500	2465000	32.160	26.512	16.753	10.107	100	4656900	11139900	165.532	147.623	65.668	55.356
100	949500	1240400	21.504	17.049	15.687	5.576	100	3158100	5670300	88.433	72.123	46.788	28.119
100	949500	949500	18.668	14.822	15.318	5.439	100	1650300	2920400	48.759	39.281	27.106	14.803
100	949500	949500	18.646	14.795	15.307	5.439	100	928700	1522100	29.201	23.748	17.079	10.686
100	949500	949500	18.696	14.795	15.295	5.439	100	558800	812900	19.152	15.642	16.036	10.613
100	949500	949500	18.696	14.817	15.316	5.439	100	349800	447400	13.633	11.633	16.045	10.659
1	8211	18373	0.279	0.245	0.221	0.142	1	46569	111399	1.580	1.348	0.737	0.650
1	4467	9727	0.177	0.155	0.178	0.111	1	46569	55839	1.105	0.884	0.713	0.390
1	2715	5111	0.121	0.132	0.152	0.090	1	46569	46569	1.012	0.799	0.694	0.360
1	1677	2676	0.106	0.122	0.140	0.082	1	46569	46569	0.995	0.778	0.694	0.360
1	1068	1430	0.100	0.110	0.140	0.079	1	46569	46569	1.021	0.786	0.694	0.360
1	695	802	0.096	0.109	0.140	0.078	1	46569	46569	1.029	0.822	0.694	0.360
100	821100	1837300	25.793	20.888	14.414	6.662	100	4656900	11139900	164.599	148.685	66.288	55.084
100	446700	972700	15.284	11.914	10.651	3.639	100	4656900	5583900	101.850	84.759	63.755	29.533
100	271500	511100	9.682	7.247	10.705	3.214	100	4656900	4656900	91.264	75.680	61.925	26.364
100	167700	267600	6.203	4.873	10.685	3.233	100	4656900	4656900	92.889	76.531	61.922	26.365
100	106800	143000	4.300	3.434	10.726	3.242	100	4656900	4656900	91.481	75.695	61.925	26.334
100	69500	80200	3.169	2.736	10.664	3.221	100	4656900	4656900	91.414	75.642	61.924	26.330