Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling met

Titel: Differential Methods in Character Rigging
Richting: master in de informatica - multimedia Jaar: 2008

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

VAN LOMMEL, Brecht

Datum: 5.11.2008

Differential Methods in Character Rigging

Brecht Van Lommel

promotor :
Prof. dr. Frank VAN REETH

Eindverhandeling voorgedragen tot het bekomen van de
graad master in de informatica multimedia

Differential Methods in Character Rigging

Brecht Van Lommel

Masterproef Informatica - Multimedia

Universiteit Hasselt

2007-2008

Promotor: Frank Van Reeth

Begeleiders: Fabian Di Fiore, Tom Van Laerhoven

1 Abstract

Most 3D animation software comes with various mesh deformers for character rigging, based on long known and widely used algorithms, like skeletal subspace deformation or freeform deformation. We demonstrate how these methods can be improved upon and made more flexible while still working interactively as part of complicated character rigs, using the same workflow as before.

We first consider the interpolation of affine transformations, and the solution of differential equations, whose properties will be taken advantage of in the algorithms we implemented. Next we consider the problem of shape interpolation, and discuss a representation for mesh deformations that encodes them in affine transformations per triangle, such that they can be interpolated and edited in ways difficult to achieve otherwise. Then we discuss skeletal subspace deformation and recent methods to get rid of its well known artifacts while still being nearly as efficient. Additionally we demonstrate how to automatically compute bone-vertex weights for skeletal deformation that give good looking deformations by default, which helps avoid manual editing and corrections to such weights. Next we demonstrate an alternative to freeform deformation that is more flexible and generic, in that it permits arbitrary and overlapping cages, which make it possible to model cages according to the shape of the character meshes.

We show their application in animated short movie, *Big Buck Bunny*, and discuss the problems they solved, and how the methods discussed can be improved and extended further.

2 Woord Vooraf

Ik wens iedereen te bedanken die mij bij het maken van deze masterproef heeft geholpen: mijn promotor Frank Van Reeth en begeleirders Fabian Di Fiore en Tom Van Laerhoven voor alle hulp en geduld, mijn familie Reinhilde, Katelijne en Stijn voor de mentale steun, Ton Roosendaal als producer van *Big Buck Bunny*, Nathan Veghdal als character rigger voor het testen van en het geven van feedback op mijn implementatie, en de rest van het *BBB* team.

Contents

3 Character Rigging

Character rigging or articulation is the process of turning a static model into an animatable character, providing high level controls for the animator. A three dimensional mesh model alone does not provide any information on what the skeletal structure is, how the character smiles, how its arms bend or how its fat jiggles. Rigging adds structure to characters such that high level controls can be animated, rather than individual vertex positions, abstracting away the details and allowing the animator to create poses quickly.

3.1 Animation Production Pipeline

The process of character rigging is just one step in the pipeline of tasks in an animation production. In a sufficiently large production, character rigging can be a full time job, from creating the initial rig, to supporting animators as they use it, extending and improving the rig for any issues that arise in the animation process.

Before rigging starts, the character model must be finished, after going through initial designs, sketches and modelling. Depending on the importance and screen time of the character, the rig may be more flexible and provide more possibilities, to make it suitable for the animations it will be used in. Once the rig is finished, it is used by animators to animate characters in shots of the movie. After those animations are finished, the character will then be rendered in the environment.

Depending on the complexity, there may be a difference between what the animators works with, and how the characters look in the final frames. The rig as used by the animator may be an approximation and have features disabled that are used for the final result, like subdivision surfaces, displacement and physics simulation for muscles, hair or cloth. This is to ensure the rig runs at interactive speed on the animator's workstation, the required framerate typically being around 24-30 frames per second.

Since animators work with character rigs day in day out, a user friendly and easy to control setup is important. For the rigger implementing a rig is not unlike making a user interface: the rig must be easy to use and efficient, providing enough control while abstracting away the unimportant parts. Simple rigs rely purely on the features provided by the animation program, while more complicated ones often involve custom behaviour through scripting for complete control.

3.2 Character Rigs

A rigged character may be thought of having a number of inputs that the animator can control. We call these animation channels, and the inputs may be for example positions, rotations, boolean values or arbitrary floating point values, corresponding to skeleton bone positions or rotations, facial expressions or any other control that the rig may provide. Based on the values of these channels

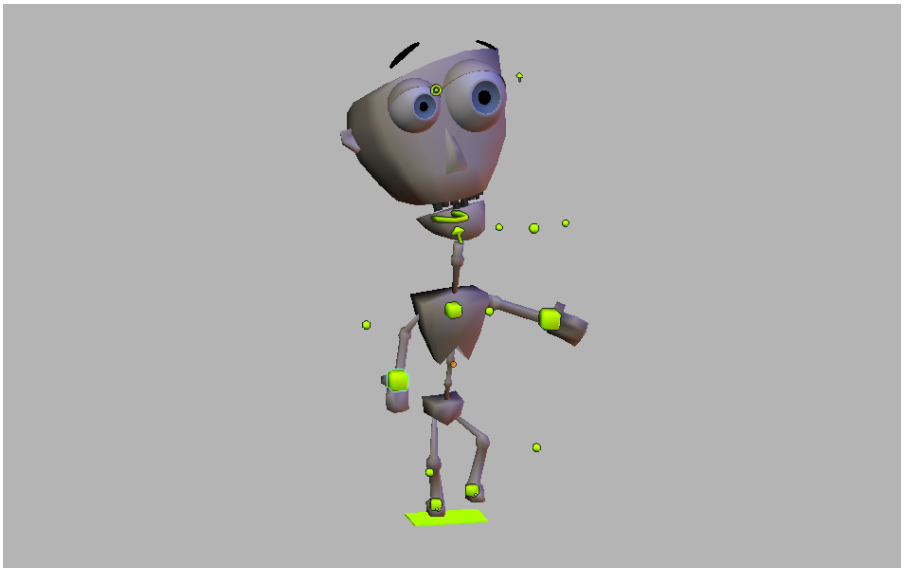

Figure 1: A character rig, showing how a number of control objects are used to pose the model, simplifying the manipulation by only requiring the animator to work with a high level representation rather than the mesh or in this case even the underlying skeleton.

at the current frame, the rig then outputs the posed character model. The inputs of these channels are animated over time by specifying keyframe values at different points in time, often using animation curves (figure 2).

In other words, in if we generalize this view, we can say the rig is like a computer program, taking a number of inputs and producing an output. Since these can become quite complicated, a good organization is necessary. A layered model is typically used, and different setups are possible depending on the algorithms or the specific character. The organization that we will be using in this text is as in figure 3.

First are the user interface controls used by the animator, in which we include skeleton bones and joints or other buttons or numeric sliders part of the rig. These provide input for the animation channels. Next there is the skeleton, a hierarchical structure consisting of bones and joints. The position of these is either directly specified by the user, or indirectly computed through constraints. Examples of such constraints range from inverse kinematics, tracking a target, to simply copying a location from one bone to another. These constraints between bones and other elements in the scene can for example be used to make the character's eyes look at a certain point, or to constraint the position of a hand to a table.

Next we we compute the deformed mesh through a series of deformers. These include skeleton bones deforming the mesh or freeform deformations. We also

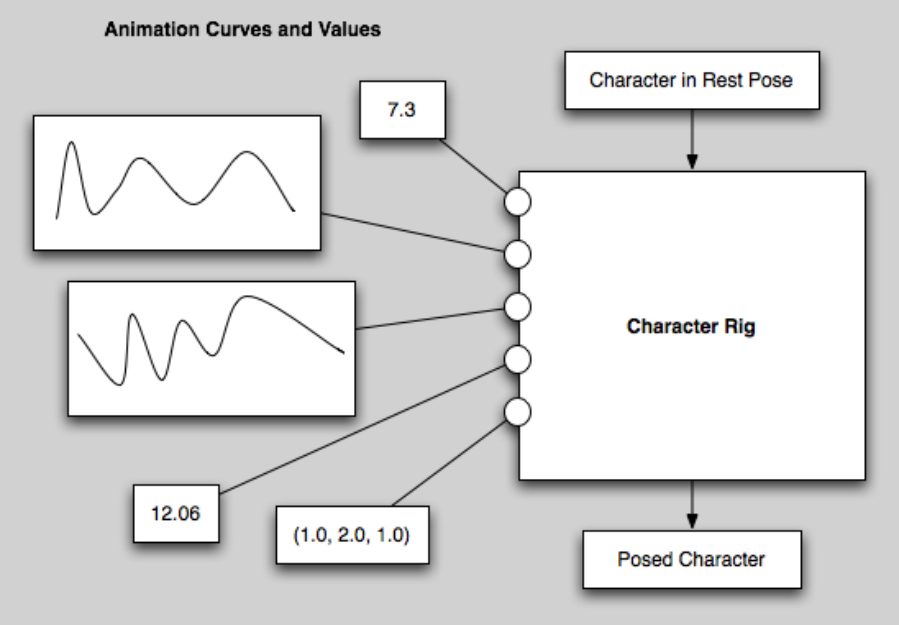

Figure 2: Based on a character in rest pose and various inputs, the character rig computers the posed character.

include here shape keys, which is a deformation specified as vertex displacements, which have been sculpted beforehand and are included as part of the rig. These deformers can be directly controlled by the animator, or automatically as part of the rig. The result is a mesh that has been deformed from its rest position into a new pose.

If some type of physics simulation is used, this usually happens after the geometric deformation applied above. Algorithms exist for muscle and fat simulation, collisions, and other effects. Depending on the complexity they are often computed offline.

Once we have the resulting mesh, more detail is often added to produce the final rendered mesh. Characters in animated films often use subdivision surfaces or NURBS, combined with displacement mapping for extra detail. This permits the animation and simulation to be done on lower resolution meshes, introducing the fully detailed model just before rendering.

3.3 Topics Covered

We will focus here on geometric methods for mesh deformation within a character rig. Skeleton setup, constraints and kinematics that drive such deformations will not be covered, nor will physics based methods that are applied after initial mesh deformation be treated here. Rather we will focus on geometric deform-

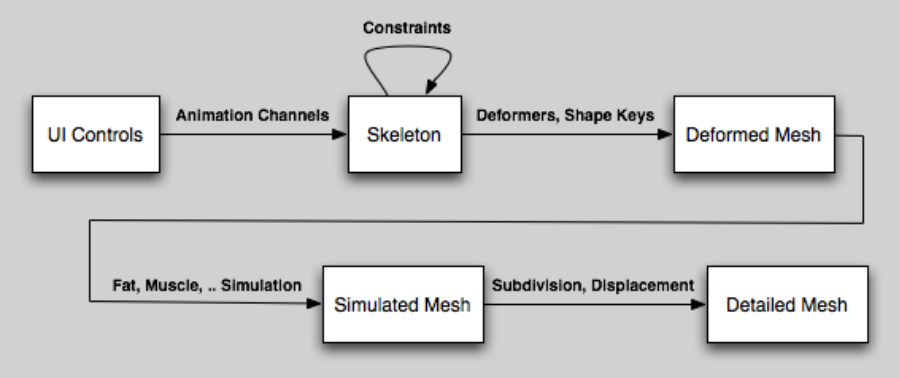

Figure 3: Successive stages in which an animated character is built up for the final image.

ers, starting from classical algorithms, investigating their limitations, and then providing solutions that give new possibilities and better results at realtime framerates.

4 Mathematical Toolbox

The algorithms in this text share a number of mathematical tools. Here we give an overview of the two most important ones: interpolation of transformations, and discrete differential operators and equations. This section gives an introduction to these, and holds as a reference for the implementation of the algorithms outlined later on.

4.1 Transformation Interpolation

Interpolation or blending is a central operation in character animation. The most common scenario is interpolation over time, from keyframe to keyframe, pose to pose. The fact that this interpolation can be done automatically is one of the most obvious advantages of 3D character animation over keyframe interpolation using pen and paper, where this has to be done manually. Even in 2D animation using computers, the problem is ill posed, as 2D keyframe drawings for example provide no direct information of their relation, or how the shape would look from a different viewpoint. Hence we can see that in 3D character animation, the ability mix and match poses, add extra deformations, etc, is one of the big advantages. We do note though that also in high quality 3D character animation much manual time is spent on the inbetween frames, to achieve effects like slow-in / slow-out, follow through or overlapping action [19].

Additionally within a single frame, the final mesh shape is a result of many channel values, specifying bone transformations, shape keys, each partly influencing the final position of a mesh vertex. In mesh skinning a deformed mesh vertex blends between multiple bone transformations, for shape keys a mesh vertex blends between different vertex key positions. Many algorithms in character animation involve interpolation of affine transformations, and careful use the of the correct algorithms is needed to achieve a good quality/speed balance. For this reason, we will give an overview of methods and issues related to this topic.

4.1.1 Linear Transformations in 3D

We will look at linear transformations, involving a 3x3 matrix and a translation vector. Such transformations can also be stored in a 3x4 matrix, or using homogeneous coordinates as 4x4 matrix with the last row is restricted to be [0 0 0 1]. Straightforward interpolation of such matrices does not yield good results, for example a direction element-by-element interpolation between the matrices will not preserve rigidity when interpolating between two rotations, so we will have to find more advanced methods. Linear 3D transformations can be decomposed into translation, scale/shear and rotation, commonly using polar decomposition [29]. The interpolation of such a transformation is then reduced to decomposition, separate interpolation of the three components and composition.

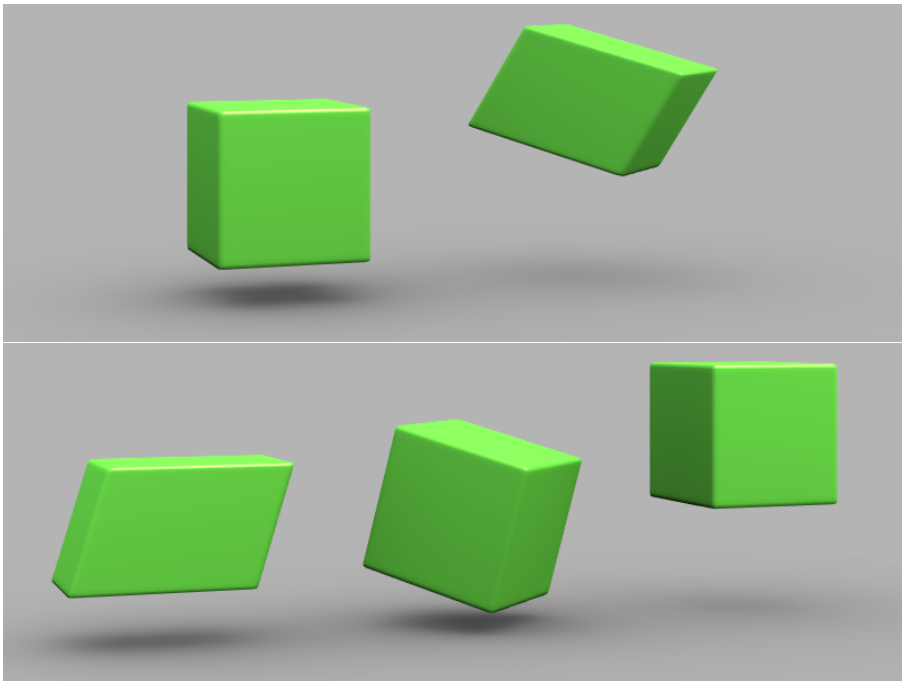

Figure 4: An affine transformation decomposed into scale/shear, rotation and translation.

Additionally, interpolation is often required to be not simply linear or at constant speed, but effects like ease in and ease out using smooth curves are used. We will not discuss these here, but simply assume to be interpolating two transformations M_1 and M_2 at constant speed, with t being the blending parameter. As interpolation operations are executed many times per frame in animation playback, next to quality, efficiency is also a concern to achieve interactive performance.

Interpolation of translation can be done by a simple linear blending of two vectors. Similarly, interpolating a scale/shear matrix can also be done by interpolating them element-by-element, with sufficiently plausible results [29]. Rotation is the most difficult, and different rotation representations and algorithms are often used instead of the matrix representation for improved results. We list here common rotations representations, with opinions on their strengths and weaknesses for our purpose. When it comes to mesh deformation and interpolation, the choice of representation is up to the programmer without much effect on the user interface, and so we can choose the most efficient representation.

4.1.2 Rotation Matrix

A 3x3 matrix with unit length rows and columns that are each mutually orthogonal represent a rotation. This representation is essential in decomposing and composing affine transformations. Direct interpolation of the matrix elements causes the resulting matrices not be be rotation matrices anymore, since the unit length and orthogonality of the rows and columns is clearly lost. This means the shape being transformed will lose rigidity and collapse.

While conversion to other representations is often used to get better interpolation, standard skeletal subspace deformation effectively does simple linear interpolation between matrices, being still the most common and simple way to do skinning. While it does result in artifacts like loss of rigidity and volume, the need for no complicated or slow computations makes this still popular.

4.1.3 Euler Angles

Euler angles are three values representing successive rotations around an axis. Multiple orderings are possible, for example XYZ, ZYX or YZY. Direct interpolation of these three values often yields unintuitive results, because each angle is interpolated separately. This does not take their interaction into account. Often it is desirable for the full rotation to take the shortest path over a sphere between the two rotations, and using euler angles the results are not guaranteed to be even close to that. Another disadvantage is that they suffer from the so called gimbal lock, where at either angle 0 or $\frac{\pi}{2}$ on one of the axes (depending on the rotation order), a degree of freedom is lost. Changing the value for that axis then has no effect. Alternatively, it means that the rotation at that point can be represented by an infinite amount of different values for that axis.

Nevertheless, the advantage is that for artists they are reasonably easy to understand, with each animation curve corresponding to a rotation around one axis. Animation packages often provide a workaround for the gimbal lock by offering multiple choices of axis rotation order. However, for skinning and shape interpolation we have not seen a good reason to use this representation.

4.1.4 Quaternions

A quaternion is defined as an extensions to complex numbers involving four rather than two values. Unit quaternions can be used to represent rotations. Assume a rotation around an axis $\mathbf{v}$ with an angle θ, then the quaternion q is:

$$q = (\mathbf{v}sin(\tfrac{\theta}{2}), cos(\tfrac{\theta}{2}))$$

The most common rotation interpolation algorithm in computer graphics, spherical linear interpolation (slerp) uses quaternions. This provides a rotation over the shortest path on a sphere with a constant speed, which is mostly commonly desired. It does involve comparatively slow trigonometric calculations, which makes it traditionally used for bones, that vary typically from tens to hundreds, rather the for example per vertex in mesh skinning, which vary from

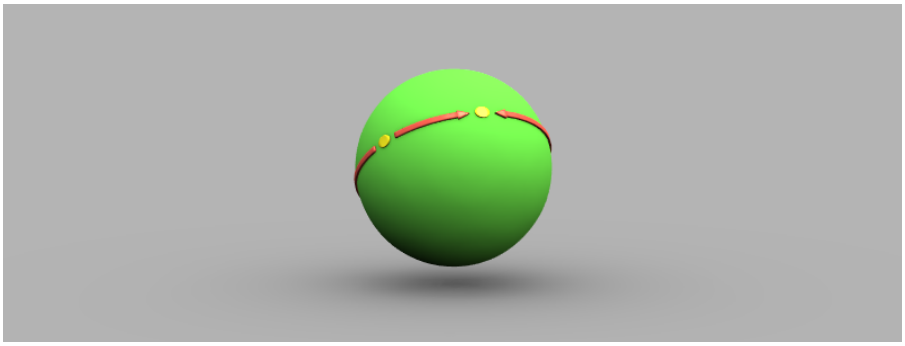

Figure 5: Two possible directions to rotate from one point to another, decided by the sign of one of the quaternions used for blending.

thousands to tens of thousands in a typical character rig. Using quaternion multiplication, inversion and exponentiation we can define slerp as [24]:

$$slerp(q_0, q_1, t) = q_0(q_0^{-1}q_1)^t$$

To ensure that the shortest path is taken, we must however do another check to ensure the shortest path instead of the long way around the sphere is used. This is done by taking the dot product of the two quaternions, and if it is negative, to use $-q_1$ instead of q_1 in the above formula. A number of variations to the slerp algorithm exist trading properties like constant speed for better performance.

A simpler way to do quaternion blending is by component-by-component interpolation of the four values. While this does not guarantee constant speed, it does still guarantee the shortest path to be taken while keeping the speed nearly constant [17]. Additionally, we often wish to blend more than two rotations. More complicated algorithms with better properties exist, but for simplicity and speed, we simply do a linear blend of all quaternions and weights in our implementation.

4.1.5 Matrix Exponential/Logarithm

The matrix exponential map is another representation that can be used [12]. Taking the matrix exponential and logarithm provides a conversion to and from this representation, which is also a 3x3 matrix. Linear interpolation of this matrix element-by-element has good properties, however, it has more parameters than a quaternion and hence is of no benefit to us. Interestingly, this operation can also to done on a full affine transformation matrix including scale and shear, however this representation has singularities which quaternions and a separate scale/shear matrix avoid.

4.2 Differential Operators

Many of the algorithms presented here will involve the solution of Laplace's equation and related operators. In this section we will give a short summary of the relevant mathematical theory, which will later be expanded upon when we arrive at the algorithms that use them. For an in depth analysis of discrete differential geometry for computer graphics, we refer the reader to [2].

4.2.1 Laplace's Equation

The Laplace operator (the laplacian) in three dimensions, with cartesian coordinates x, y and z, is defined as:

$$\triangle\phi = \frac{\partial^2\phi}{\partial x} + \frac{\partial^2\phi}{\partial y} + \frac{\partial^2\phi}{\partial z}$$

The scalar function $\phi(x, y, z)$ is defined over a three dimensional volume. This operator can be used in two dimensions with an analogous definition, dropping the z component. Useful in our application, this operator can also be extended to work on surfaces, which yields the so called Laplace-Beltrami operator. Next to a three dimensional volume, we will use such an operator on the surface of a mesh. But regardless of the domain, these operators share many properties, and we will simply consider the definition over a three dimensional volume in this chapter.

Using the laplacian, Laplace's equation can be written as:

$$\triangle\phi = 0$$

In order for a unique solution to be found, boundary conditions are used to specify values at the boundary of the domain. For example if we wish to solve this equation in a three dimensional volume, the boundary of this volume could be given fixed values, and the internal values would then be derived from the solution of this equation.

Solutions to these equations are called harmonic functions. These functions are smooth away from the boundary, specifically C^∞ continuous. Such continuity is visually pleasing, and means there are no sudden sharp discontinuities in the result. For example in the design of smooth surfaces such continuity is often sought for, and if we are deforming or interpolating surfaces it is desirable to have smooth results as a default as well.

Closely related to this is Poisson's equation, where f is a scalar function over the domain, generalizing Laplace's equation:

$$\triangle\phi = f$$

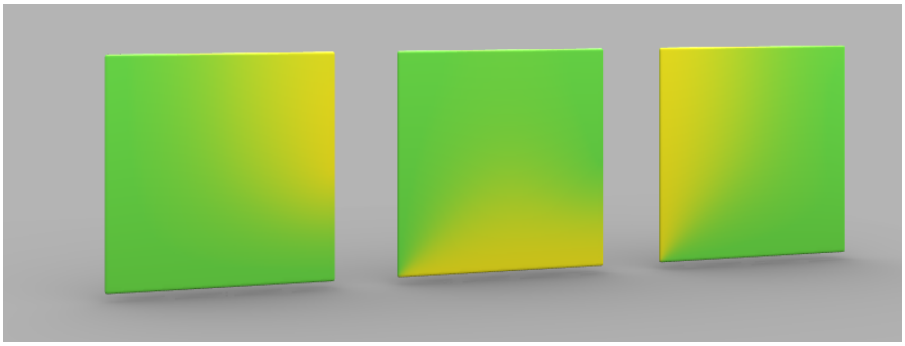

Figure 6: Solutions to Laplace's equation on a plane, with different boundary conditions.

4.2.2 Operators and Properties

We define a number of additional operators and properties that will be used later on in this text. Again considering a scalar function $\phi(x, y, z)$, the gradient is defined as:

$$\nabla\phi = (\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z})$$

The divergence is defined on a vector field $F(x, y, z)$ as:

$$\nabla \cdot F = \frac{\partial F_{\mathrm{x}}}{\partial x} + \frac{\partial F_{\mathrm{y}}}{\partial y} + \frac{\partial F_{\mathrm{z}}}{\partial z}$$

Their connection with the laplacian is:

$$\nabla \cdot (\nabla\phi) = \triangle\phi$$

Since these operators are quite abstract, let's look at an example from physics for an interpretation. Suppose we have a volume with heat sources and sinks. We can specify those sources and sinks as having a fixed temperature. These are the boundary conditions. The scalar function ϕ then is the temperature at each point in the volume. At a given point, the gradient of this function gives the direction in which heat increases most (i.e. towards the heat sources). The divergence of the gradient then indicates how much the given point in the volume acts as a source or sink of heat. If the divergence is zero everywhere (except at the specified sources and sinks), it means an equilibrium is reached, and without changes in the specified sources and sinks, the temperature will stay the same at all points. Since the divergence of the gradient is the laplacian, that means an equilibrium is reached when Laplace's equation is satisfied (figure 7).

Figure 7: A real life example of the heat equation.

4.2.3 Applications

These operators have many applications in physics. For example in fluid dynamics as part of the Navier–Stokes equations which are central to the theory, or in the heat equation describing the distribution of heat in a given region. In computer graphics they have also been shown to be useful for simulating these effects [32], but also in other areas such as rendering [16], mesh processing [31]and character animation [13], since efficient methods exist to solve such equations.

4.2.4 Discrete Operators

When applying these equations to numerical simulation in physics or computer graphics, the relevant functions and operators are discretized. The values of the functions then become defined at certain points on the domain rather than at every position over a continuous region, and the operators, rather than being defined as the limit over an infinitely small region, are defined as linear combinations of the values at these points.

For example on the surface of a mesh, we may define the function values at the vertices and interpolate their values on the triangles. In general, quantities may be defined on different elements of the mesh, like vertices, edges, triangles, tetrahedra or grid cells, but in this text we will solely deal with quantities defined on vertices connected by edges, and triangles between these edges. Applying the operator to a function then corresponds to multiplying a vector of these values by a sparse matrix. How exactly to create these different sparse matrices will not be discussed here with the exception of the laplacian operator whose implementation will be discussed in the next section, see for example [25] for an in depth overview.

For reference, in the formulas in this text we will assume the operators to

be discretized as sparse matrices as follows:

Operator	Symbol	Input	Output
Laplacian	$\triangle$	Scalar on Vertices	Scalar on Vertices
Gradient	∇	Scalar on Vertices	Vector on Triangles
Divergence	$\nabla\cdot$	Vector on Triangles	Scalar on Vertices

4.2.5 Discrete Laplacian

The continuous laplacian operator is defined as:

$$\triangle u = \frac{\partial^2 u}{\partial x} + \frac{\partial^2 u}{\partial y} + \frac{\partial^2 u}{\partial z}$$

Our goal here is to define a laplacian operator on a graph, with a value u at every vertex. As this is a continuous definition with infinitesimal numbers it needs to be discretized. In the case of finite element simulation on uniformly scaled grids, the discretization is particularly simple. Assuming the quantity u is defined at the grid vertices, the following formula defines the laplacian at vertex v_i with n neighbouring vertices v_j and $j = 1 \dots n$:

$$\frac{1}{n} \sum_{j=1}^{n} (u_j - u_i)$$

The collection of these for all vertices, each being linear, can be expressed in the form of a sparse matrix. The sparse matrix is defined such that multiplying a vector u of length n with it will give a vector with the result of the above formula. Linear equations involving large, sparse matrices can be efficiently solved using the appropriate solver, the choice of which we will detail in the section 8.

When dealing with regular grids, this simple definition is sufficient. When the field u is defined on an arbitrary mesh which does not have equally sized elements, we must use better weighting. While giving each neighbouring vertex a uniform weighting like in the definition above works, it does not take the shape of the elements into account. For example if we are solving the heat equation, we evidently need to take the distance between the points in space into account, to compute the correct equilibrium, so that nearby points will have more influence then points further away. From a mathematical point of view, we wish to preserve the properties of the continuous counterpart as much as possible.

For solving the laplacian on a manifolds mesh surfaces, the so called cotangent weights are the common choice. It has been shown that no single weighting scheme can preserve all properties of the continuous laplacian [35], however in our case we simply choose cotangent weights as they satisfy the properties we are most interested in. This leads to the formula for each vertex:

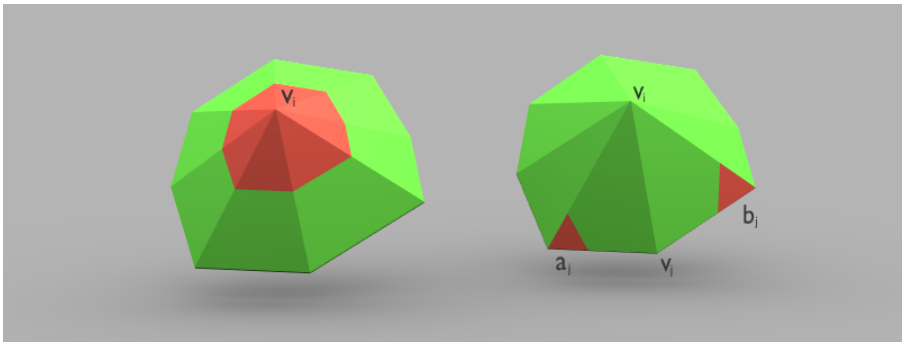

Figure 8: Voronoi area around a vertex on the left, and the angles used for the cotangent weights on the right.

$$u_i - \frac{1}{A_i} \sum_{j=1}^{n} w_{ij}(u_i - u_j)$$

The area A_i is the voronoi area around the vertex, see figure 8. To deal with poorly shaped triangles, we follow the rules outlined in [23] for the computation of this area. The weights $w_{ij} = w_{ji}$ are defined as:

$$w_{ij} = \frac{1}{2}(\cot \alpha_{ij} + \cot \beta_{ij})$$

The angles α_{ij} and β_{ij} are defined as in figure 8, at the triangle corners opposite the edge between v_i and v_j.

4.2.6 Boundary Conditions and Constraints

If we are going to solve Laplace's or Poisson's equation, some boundary conditions will need to be specified, in order for the solution to be unique. We will be using so called Dirichlet boundary conditions here, that is, some vertices in the graph will be given fixed values for u_i (other types of boundary conditions can specify fixed derivatives for example). For the laplacian of a single connected graph, we need to lock down at least one degree of freedom that exists in it. We can see this in the equation for a regular grid for example: if we add the same real number r to all the elements in the solution u, Laplace's equation is still satisfied. In practice we will always give one or more vertices a fixed value both to ensure this degree of freedom is removed, and to control the solution for our purpose. Another way to say this is that we add constraints to the solution.

Implementing this is quite simple. Given a laplacian matrix $\triangle$ constructed from a connected graph of n vertices, and provided a subset of graph vertices that need to be constrained. Suppose we wish to fix the value of vertex v_i in the graph to a value c_i with $1 \leq i \leq n$. Given a right hand side vector f we have a Poisson equation.

$$\triangle\phi = f$$

We remove the row and column i from the matrix resulting in $\triangle'$, and remove the element i from the vector f resulting in f'. Assume $\triangle'_i$ is the column i of $\triangle$ with the element from row i removed, we get:

$$\triangle'\phi = f' - f_i\triangle'_i$$

This effectively eliminates the vertex v_i from the system and compensates for it in the right hand side of the equation. This procedure can be repeated for each constrained vertex.

5 Shape Interpolation

A typical character rig includes shape keys, a collection of vertex displacements from the rest shape. Some other terms used are morph targets or key shapes. In character setups, they can be controlled by animation channels or automatically as part of a mechanism in the rig. For example shape keys corresponding to facial expressions like close eyelids, smile, frown, can be created without using bones, where there is no obvious correspondence to a bone in the skeleton, and instead be directly sculpted. Additionally, animators might specify vertex positions as keyframes to achieve just the right pose that the rig does not (easily) permit, or if it is faster to animate vertex positions than setup a full rig for a character or object that only needs a minor amount of animation.

5.1 Applications

5.1.1 Expressions

The most typical application is for facial expressions. This goes from small specific deformations like "open mouth" or "left eyelid close", to full expressions like "happy", "sad", "evil grin", .. .

5.1.2 Corrective Shapes

Another application is to correct or improve deformations generated in other ways. For example simple skeleton deformation leads to certain artifacts, which might be countered by having a shape key activated when this happens, correcting for the error. Or if skin self intersects due to fat colliding, a shape key can then deform the mesh to avoid intersection. These shapes can be driven by the character rig, based on bone angles for example. This avoids slower physical simulations or other more advanced skinning methods. It also gives artists more control over the exact deformation, for example if a specific cartoony deformation is desired.

5.1.3 Example Based

When a database with character deformation examples is available, these can be used to ensure the character deforms similarly when animated. Scanning technology makes it possible to acquire example poses, which can be used to drive the deformation realistically by learning from these examples, see for example [4].

5.2 Pose Space Deformation

A well known framework to automatically drive shape keys is pose space deformations (PSD) and its variations [21, 18]. The workflow for riggers is as follows. Using a character that is already bound to a skeleton, it is put into a pose that needs a correction or improvement. The animator then sculpts the shape key

in that pose, and repeats this process for different poses. When the character is animated, it will automatically hit those shape keys as the bones are posed with similar angles, and smoothly falloff as it gets further away from that pose. But not only bone angles can be used to drive shape keys, any animation channel including custom values defined by the rigger like "eyelid close" could be taken into account, providing an intuitive way to add controls to the rig, specifying deformation by example.

The method is based on scattered data interpolation with radial basis functions, providing an efficient method to interpolate data from varying poses smoothly. This algorithm may be used with different shape interpolation methods as discussed in the next sections. There we will however assume interpolation weights for each shape key to be given per vertex or triangle, whether they are computed using pose space deformation or any other method.

5.3 Object Space Interpolation

The simplest way to implement blending between multiple shape keys is blending the vertex positions in object space. This works well as long as the deformations are not too large, and for facial expressions this is often the case. However under rotations the mesh can 'collapse' or produce other nonsensical inbetween results. Additionally, deformations may combine poorly, adding many deformations on top of others is problematic then. Note that for these to work correct with bone deformations, they must be applied first, followed by the bone deformations, since they are not invariant to rotation and scale.

5.4 Tangent Space Interpolation

An alternative is to blend between the deformations in a local space at the vertex, formed by the normal and two tangent directions (tangent space). The vertex displacement is then transformed into tangent space, interpolated and transformed into object space again. Such methods avoid some of the issues of simple linear interpolation. This means for example that the if the base mesh is rotated, the deformation will still be correct. However, this representation still doesn't recognize rotations for example, and will cause shrinking when interpolating between a base shape and a shape that is rotated. It provides more flexibility in the ordering of deformations add some additional computation cost, but by itself does not improve interpolation and extrapolation quality.

5.5 Differential Interpolation

A different way to encode the transformations is per triangle instead of per vertex. For each triangle we can encode the difference in the transformation from one shape key to another. It turns out that when disregarding the translation component and using only a 3x3 matrix per triangle, the shape key can be fully reconstructed, except for the global translation. This means that we can interpolate and extrapolate affine transformations rather than just translations.

Two approaches for this have been proposed, poisson mesh interpolation [37] and deformation gradients [33], which have later been shown to be equivalent [9]. Here we will follow the exposition in the last paper shortened to the parts most relevant to the implementation. While these methods are significantly slower than the above mentioned ones, they provide superior interpolation and extrapolation results.

Consider scalar functions ϕ_i over the surface of the mesh, one for each vertex v_i with $i = 1...n$, and position p_i. It is defined to be 1.0 at the vertex v_i and 0.0 at all other vertices, with the value interpolated over edges and triangles, such that each point on the surface can be retrieved as:

$$p(\mathbf{x}) = \sum \phi_\mathrm{i}(\mathbf{x}) \cdot \mathbf{p}_\mathrm{i}$$

In other words, the values of $\phi_\mathrm{i}(x)$ for the three vertices p_i in a triangle give the barycentric weights for the point x in that triangle, and these will reproduce the original position when used as weight to interpolate the vertex positions. We can then take the gradient of this function to get:

$$\nabla p(\mathbf{x}) = \sum \nabla \phi_\mathrm{i}(\mathbf{x}) \cdot \mathbf{p}_\mathrm{i}^\mathrm{T}$$

Note that $\mathbf{x}$ is a three dimensional vector and that we have three vertices with non-zero values of ϕ_i in each triangle, so the gradient becomes a 3x3 matrix G_j constant for each triangle $j = 1..m$. We now still need to know how to compute $\nabla\phi_\mathrm{i}$ for each triangle t_j. Let us consider a coordinate frame formed by that triangle, with axes:

$$[\mathbf{p}_1 - \mathbf{p}_3, \mathbf{p}_2 - \mathbf{p}_3, \mathbf{n}]$$

Here $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ are the vertex positions of the triangle, and $\mathbf{n}$ is the normal of the triangle. Since we know $\phi_\mathrm{i}$varies linearly over the triangle (they are barycentric weights), and if we compute the derivatives w.r.t. to an $\mathbf{x}$ defined in this coordinate frame, the solution becomes particularly simple. Since ϕ_i is linear, the derivative is constant and we may compute it with:

$$\nabla\phi_\mathrm{i} = (\phi_\mathrm{i}(1,0,0) - \phi_\mathrm{i}(0,0,0), \phi_\mathrm{i}(0,1,0) - \phi_\mathrm{i}(0,0,0), \phi_\mathrm{i}(0,0,1) - \phi_\mathrm{i}(0,0,0))$$

So the value of $\nabla\phi_\mathrm{i}$ for the three vertices is:

$$\nabla\phi_1 = (1,0,0)$$

$$\nabla\phi_2 = (0,1,0)$$

$$\nabla\phi_3 = (0,-1,-1)$$

Rewriting this in matrix form, G_j can now be computed as:

$$G_\mathrm{j} = \left(\mathbf{p}_1^\mathrm{T}, \mathbf{p}_2^\mathrm{T}, \mathbf{p}_3^\mathrm{T}\right) \cdot \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{pmatrix}$$

Now the next step is to bring in a shape key with vertex positions $\mathbf{p}'_\mathrm{i}$. The so called deformation gradient T can now be computed as the transformation from the gradient G_j to G'_j:

$$G_\mathrm{j} T_\mathrm{j} = G'_\mathrm{j}$$

$$T_\mathrm{j} = G_\mathrm{j}^{-1} G'_\mathrm{j}$$

So to recap, what have we computed? With G_j, we have the gradient of the vertex positions with respect to a position on the surface, which is constant on each triangle. The deformation gradient then is the change in this gradient for a particular shape key, which is a 3x3 transformation matrix. That means we have a new representation which we can manipulate in interesting ways that would otherwise be very difficult to do. For example, we can now interpolate these matrices T rather than vertex positions. Or we can remove the scale/shear component from that matrix to make a deformation rigid.

But, we're not there yet, while we now know how to compute deformation gradients from vertex positions, how do we go the other way around to retrieve vertex positions? We can think of these deformation gradients transforming each triangle individually, with the result that the vertex positions between triangles do not match up anymore The solution is then to devise a way to stitch all those transformed triangles back together such that the transformation of each triangles is preserved as much as possible. It turns out we can do this by solving Poisson's equation!

The gradient $\nabla p(\mathbf{x})$ as defined here corresponds to a gradient operator on the function $p(x)$ over a mesh surface [25]. Note how the quantities correspond with the discrete gradient operator on a mesh as defined in section 4. There the operator took a scalar as input and resulted in a vector as output. Here we take a vector as input and output a matrix. The link is that what we do here is basically applying the operator to each scalar component in the vector and then get three vectors back, used as columns in the matrix.

For simplicity, we will now deal with sparse matrices and vectors for the full mesh. The vector $\mathbf{p}'$ contains the vertex positions we wish to compute, and the vector $\mathbf{G}'$ consists of gradient matrices, which have been computed in some way, by manipulating the original gradient matrices to achieve some result. Ideally we could solve the following equation:

$$\nabla \mathbf{p}' = \mathbf{G}'$$

Remember that the gradient is a discrete operator that corresponds to a sparse matrix. If we can now invert the sparse matrix ∇, we could retrieve the positions. However, this matrix is not invertible in general (it's not even square), and as a result we will resort to a least squares solution that gets the result as close as possible. We could write this as:

$$\nabla^{\mathrm{T}} D \nabla \mathbf{p}' = \nabla^{\mathrm{T}} D \mathbf{G}'$$

The diagonal matrix D is used for weighting of the least squares solution by triangle area. Another way of getting to this result is by applying the divergence operator to both sides, which gives a Poisson equation that is fully equivalent to the above formula:

$$\triangle \mathbf{p}' = \nabla \cdot \mathbf{G}'$$

Hence, we have arrived at the solution. The laplacian is computed as in section 4, the divergence is computed as $\nabla^T D$. In practice we will solve this equation three times with a different right hand side, once for each spatial dimensions. A single vertex must be constrained to a given position, as noted in section 4, which in this case corresponds to a global translational degree of freedom.

The algorithm now proceeds as follows. We compute the gradient matrices for all shape keys, convert them to a quaternion and a scale/shear matrix, and cache them for later use. Then we interpolate those gradient matrices per triangle based on the current animation channel values of the shape keys. For a given mesh, we detect the connected components, of which there might be multiple, and construct a separate laplacian matrix for each. For each component, we plug the resulting matrices into the Poisson equation, and constrain an arbitrarily chosen vertex to location $(0, 0, 0)$. Next we solve for the new positions, caching the solution of the matrix factorization for later reuse (see section 8 for more details). Now we position the components, by transforming them such that the averaged vertex position center remains in the same place.

We provide this interpolation method as an alternative to simple object space interpolation of vertex positions. Additionally, for each shape key, we offer the extra option to remove scaling or rotation from the shape key. This is useful to make a shape key rigid for example. Results of the implementation are shown below.

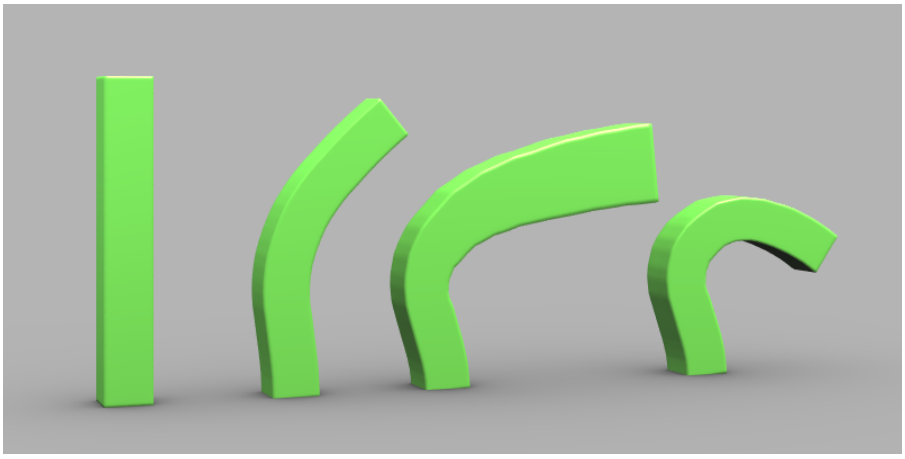

Figure 9: From left to right: original mesh, a given shape key, extrapolation of the shape key in object space, and the differential method. Note how simple object space blows up the mesh and does not provide an intuitive results, while the differential method gives in a more expected results.

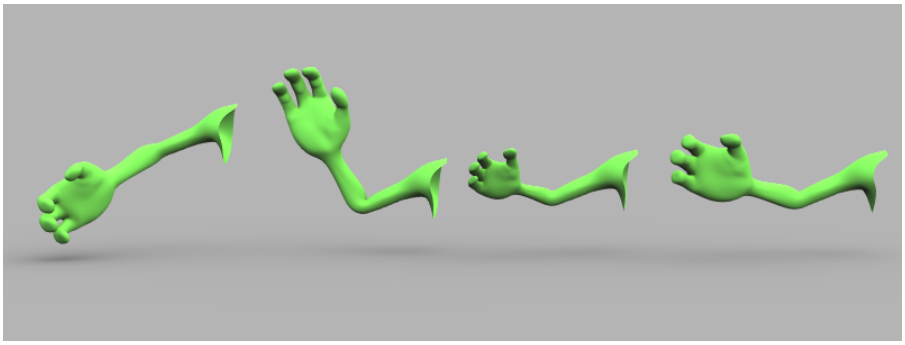

Figure 10: From left to right: original pose, shape key, interpolation in object space, and differential interpolation. Note how the object space interpolation loses rigidity and shrinks inbetween the two poses.

6 Skeleton Based Deformation

6.1 Skeletal Subspace Deformation

The standard method to do skeleton deformation is using skeletal subspace deformation (SSD) [24]. Consider a vertex with position v influenced by n bones from the skeleton with weight b_i and transformation relative to the rest transformation M_i in object space for $i = 1...n$. Then the new vertex position is:

$$v' = \sum_i b_\mathrm{i}(M_\mathrm{i}v)$$

$$\sum_i b_\mathrm{i} = 1$$

So, we are transforming the vertex position by each transformation and then interpolating the result. The per vertex bone weights may be computed automatically, based on the distance from the bone for example, though in many cases these will be manually edited by the user for optimal results. We will discuss automatic weight computation in a later section.

Even given good weights, this method tends to give certain artifacts. As we can see in the following pictures, the mesh quickly collapses and loses volume when the bones are rotated to angles more than about 45 degrees relative to each other. Looking at an example with a rotation of a bone twisting around its own axis, we can see why this has been named the "candywrapper effect", see figure 11. Users can work around this by using corrective shape keys, or by introducing more bones around the collapsing joint. Most animation packages or games still use this interpolation method with these artifacts, and indeed require the user to solve the problem. Here we will look at automatic algorithms to counter these effects.

6.2 Improved Transformation Blending

There are different reasons for these artifacts, one being related to the transformation blending. A simple rewriting of the SSD equation reveals the problem:

$$v' = (\sum_i b_\mathrm{i}M_\mathrm{i})v$$

We can see that it is equivalent to interpolation of transformation matrices component-by-component, which does not preserve rigidity as explained in section 4. The solution to this problem is to use a better but slower interpolation method. We can decompose the matrices M_i into translation, rotation and scale/shear. Since this is a per vertex operation, we need to use a quick method for interpolation of rotation, and so we use linear interpolation of quaternion vectors, which provide a shortest path interpolation with constant speed, as

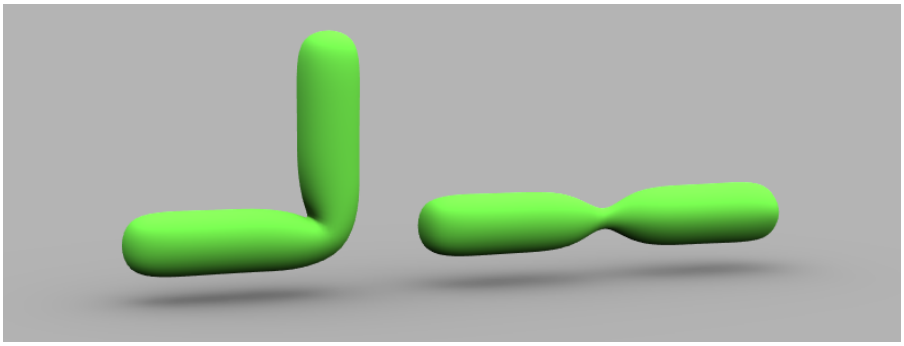

Figure 11: Two examples of loss of volume when using SSD. A simple 90 degree rotation of a bone on the left demonstrates what could be a collapsing elbow, and the same bone on the right twisted shows even worse artifacts, known as the candywrapper effect.

explained in section 4. Scale/shear uses element-by-element interpolation of the 3x3 matrix. It is tempting to simply use linear interpolation of the translations too, but this does not work well.

Note that while bones are usually only rotated and not translated relative to their parent, the child bones are translated indirectly by rotation of the parent in object space. Since the matrix M contains the relative transformation in object space, the translation component is the center of rotation around which the bone rotates. Linear interpolation between the centers of rotation does not work well at all. The child bones will rotate around a different center, and this is were the issue is. If we consider a blend between the transformation of a single parent bone and a child bone, we're blending transformations with different centers of rotation. In such a case we must carefully select the center of rotation, otherwise the result will drift away from the expected position.

Recently a method [17] was proposed for dealing with this issue using dual quaternions, that provides a good trade off between quality and speed. A dual quaternion in the context of 3D transformations, consists of two quaternions, one for rotation which is identical to a usual rotation quaternion, and one for translation. Crucial for this representation is that it non-linearly interpolates the center of rotation to match the non-linear interpolation of rotations. The maths that prove this are quite involved, suffice it to say that the formula result from these derivations is still almost as efficient as simple SSD. We can see many artifacts are removed compared to simple SSD skinning, as show in figure 12 and 13.

Such transformation blending does have a disadvantage compared to standard SSD, due exactly to the fact that rigidity is preserved. At an angle of 180

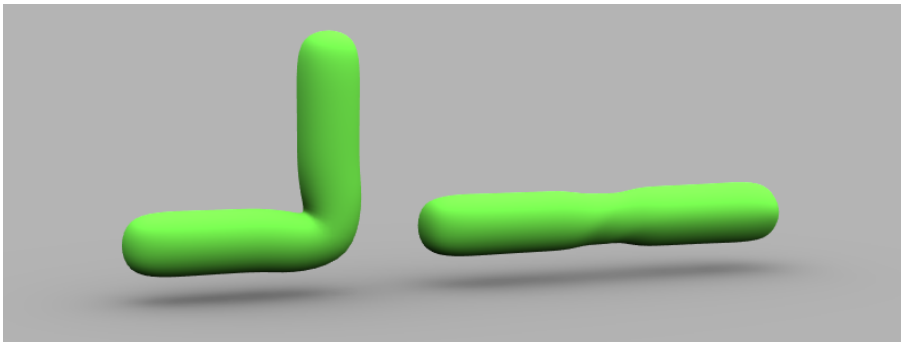

Figure 12: The same deformation as in figure 11, but using dual quaternions. For this simple case, nearly all the volume loss is gone.

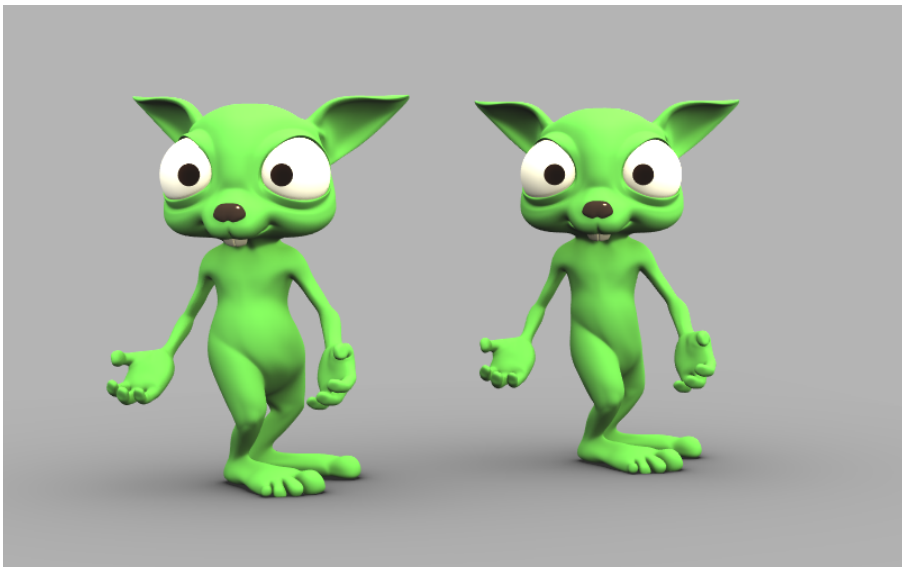

Figure 13: The same pose with dual quaternions and SSD created by twisting the spine to make the character look sideways, note the unintended loss of volume for SSD.

degrees, there is no single shortest path to take, the rotation could go any way around. A small change in rotation can then cause a completely different shortest path, and so will give a completely different result. In this case the SSD result collapses onto a single point, but still gives a continuous results without popping. We do note that for typical skeletons, such a joint configuration is highly implausible anyway, and that we have only observed this in contrived cases (it's clearly impossible to rotate a joint like an elbow more that 180 degrees backwards). An interesting solution could involve a rotation representation and blending that actually includes information on which way the bone was turned, and even how many times it was turned around its axis, although we predict such a method could be quite incompatible and difficult to implement in existing animation systems. Another solution is to resort to some sort of simulation, which will take the previous frame into account and so will take the shorter path from frame to frame.

6.3 Using Curves

A curve based approach is often used for modelling the spine which consists of many bones, but we can generalize this idea to more joints. We can think of this as improving transformation blending by doing it at the bone level. Consider an area of the mesh being influenced by two connected bones. We can create inbetween bones that interpolate the transformation of the original bones using better transformation blending, without the per vertex cost, since the result of this affects many vertices. The per vertex weights then need to be changed to refer to those smaller bones. An application of this idea for joints like elbows or knees, is to think of the bone joints being control points of a bezier spline or other smooth curve, and subdivide the bones into smaller ones according to this according to the curve. Next to improved quality, curve control parameters give interesting possibilities for users to tweak the look of the deformation as well.

6.4 Differential Interpolation

Just like for shape interpolation, we can take a differential approach using deformation gradients [36]. This is done by making per triangle bone weights (we simple average the weights of the three vertices), and interpolating the bone transformations there. We then reconstruct for the vertex positions just like shape interpolation. In fact, we only need to do a single reconstruction, since we can combine shape interpolation and skeletal deformation in a single step.

Let us look at how this works. First we compute the gradient matrices for the base vertex positions, denoted as G_j for each triangle $j = 1 \dots m$. This can be done only once for the first time and cached for later reuse. Next we go over all bones influencing this triangle through the weights of its vertices. We can do weighted interpolation of their matrices using any of the methods described in section 4, here we used linear quaternion blending. This will result in a matrix M'_j for each triangle. This then results in:

$$G'_j = G_j M'_j$$

From these new gradient matrices, we can retrieve the new vertex positions as described in section 8. An interesting extensions to this algorithm would be to add volume preservation. One solution is to apply the algorithm to tetrahedra instead of triangles, a faster alternative is to extend the existing graph with extra edges to take volume into account.

6.5 Volume-Preserving Deformations

An interesting new skinning method was proposed in [5]. Suppose we have a spatial deformation function f. In the case of SSD for example, this spatial function for each vertex is simply the result of the matrix interpolation. If the divergence of the associated velocity field, i.e. the gradient of f is zero everywhere, then the deformation can be proved to be both fold-over free (no self intersection) and volume preserving [5]. This can be exploited as follows. We can think of the transformation from bone rest transformation to the deformed transformation as a continuous motion, with the gradient of f being the velocity. The paper presents a method to make the gradient divergence free, and then integrates using this new gradient towards the final position, with user adjustable parameters for how much volume should be preserved. While this method involves velocities, it is still purely kinematic, and a few integration steps are computed for each vertex in every frame starting from the rest position. The computation is still relatively efficient.

6.6 Using Dynamics

Here we have only considered skeletal deformation methods that are purely kinematic in nature. That is, from a given skeletal pose at a single frame, the result can be computed directly. By using dynamics including motion over time and forces, we can get more physically based results. For example self collision can be avoided explicitly if collision is integrated into the dynamics simulation. Physics based approach are often added offline after kinematic animation is finished as such methods are typically slower. Another issue is that such approaches are more difficult to control for animators. Part of this is due to some of the inherent complexity that is involved in physics simulation, another part is due to the fact that editing the pose at one frame has an effect on following frames, and so control is not as local as it is for purely kinematic approaches. Still, dynamics can solve many issues that purely kinematic approaches have to work around. Kinematics and dynamics are not mutually exclusive, since a kinematic deformation is often the first step after which dynamics are added.

6.7 Computing Vertex Weights

Manually editing vertex weights is a time consuming task, and we wish to at least automatically compute a set of weights that give good deformations. These

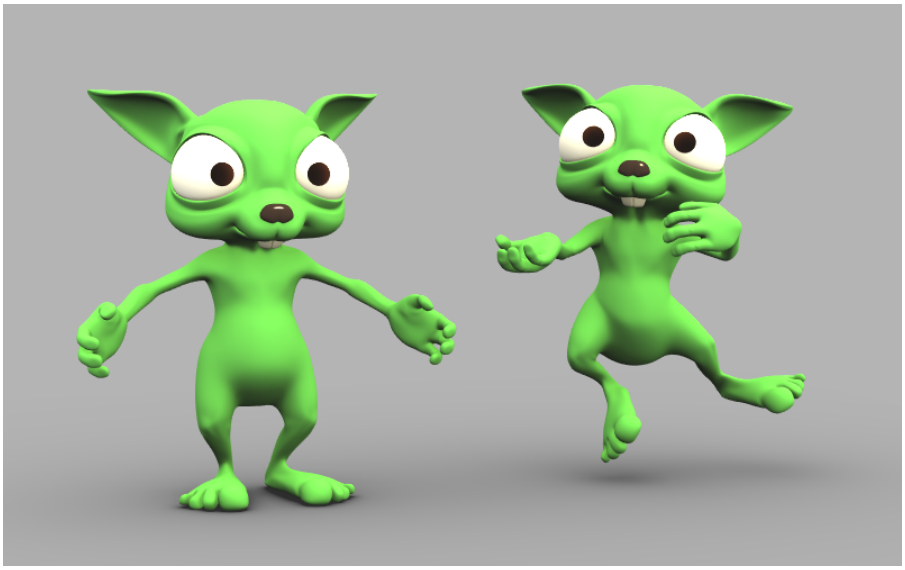

Figure 14: Rest pose and new pose using the computed weights.

can then be further changed by the artist to fix remaining issues and change the look. One such automatic method is enveloping, where each bone has a surrounding ellipsoid. If the vertex falls within the volume of the ellipsoid of a bone, it is influenced by that bone. The weights are then computed based on the distance of the vertex to the bone. If multiple bones influence a vertex, the weights are normalized to sum to 1.0. This method and other heuristics might be improved by smoothing out weights, or removing "outlier" weights on vertices that are directly connected to other vertices with weights. We will use a method that arguably gives much better results than such heuristics without the user having to tweak any settings for the algorithm.

The method can be interpreted in terms of the heat equation. To compute the weight for a bone, we give it a temperature 1.0, and we set the temperature for all other bones to 0.0. Then the heat equation can give us the heat in equilibrium over the volume inside the mesh, which is a smooth harmonic function. Solving this equation for each bone, we get per vertex SSD blending weights for that bone. Rather than deriving the formulas from the heat equation as in [6], here we will use the Laplace equation with boundary conditions for consistency with the harmonic coordinates algorithm explained the next section.

For simplicity and speed, we do not solve the equation over a 3D volume, but instead on the surface of the mesh. This means we avoid embedding the mesh and bones in a 3D grid, or constructing a tetrahedral mesh. The disadvantage is that the weights will not be smooth between vertices that are nearby in 3D space but not over the surface. This would be useful for disconnected compo-

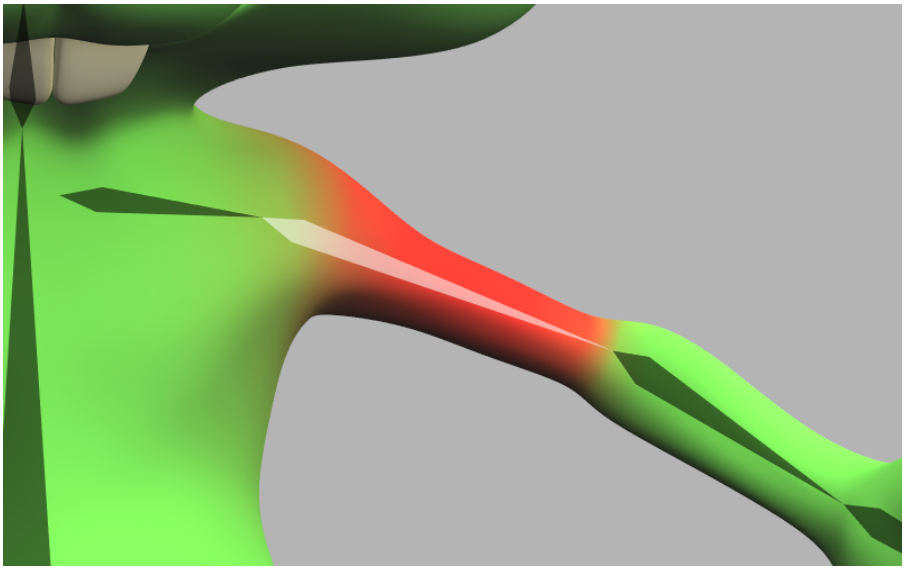

Figure 15: Resulting weights for a left upper arm.

nents, for example eyes in a head might get disconnected during deformations because the weights at the eye sockets do not match well.

We will solve for the weights of each bone independently, solving one instance of Laplace's equation each time. Assume we are computing the vector of vertex weights b_j for bone B_j with $j = 1..m$. We have to compute the weights for the vertices v_i for $i = 1...n$. For each bone, we first detect which vertices are *visible*, meaning there are no triangles between the bone and the vertex, so as to avoid bones effecting spatially nearby but unrelated parts of the mesh. We do this by tracing a ray from the vertex to the closest point on the bone. This means (number of bones)×(number of vertices) rays must be traced, so for efficiency we reuse a raytracing acceleration structure, as was already available for rendering. Note that this simple visibility detection using only a single ray and direct line of sight will give a fairly rough initial guess, including gaps. This is no problem however, as the solution of Laplace's equation will fill in the gaps and smooth out the result. The original paper used a three dimensional distance field as available from a skeleton fitting method their algorithm is part of, however using the existing raytracing code made the implementation considerably smaller, and gives nearly identical results.

The laplacian is defined over the graph of mesh vertices, with the solution at each vertex being influenced by its surrounding vertices. Here we virtually extend this graph, by adding one extra vertex for each bone, connected to the mesh vertices *visible* from that bone. Since for best results the laplacian matrix should use weighting, so must we define compatible weights for these extra bone

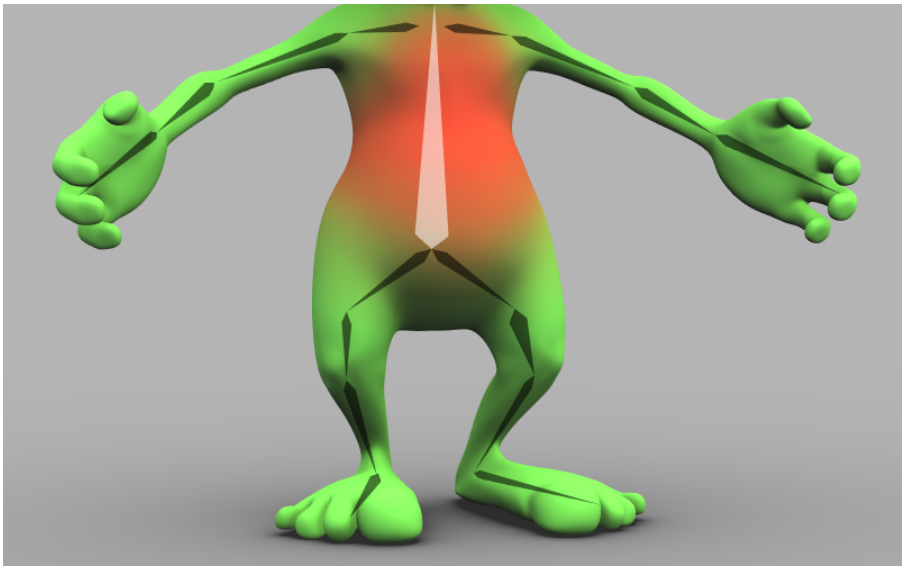

Figure 16: Resulting weights for a spine bone.

vertices. We simply weight the connection between the vertex associated with bone B_j and the mesh vertex v_i inversely proportional to the square of the distance d_{ij} to the closest point on the bone:

$$w_{ij} = \frac{1}{(d_{ij})^2}$$

Now assume $\triangle$ to be the extended laplacian operator including these extra bone vertices. We now solve the following equation, under the boundary conditions that these extra bone vertices have fixed weights, being either 1.0 or 0.0 depending on which bone we are solving for.

$$\triangle b_j = 0$$

In practice, we may rewrite this as a Poisson equation using the laplacian operator from the original mesh without extra vertices, folding the boundary conditions into the right hand side, as explained in section 4. Doing this gives us the heat equilibrium equation used in [6]. The result is a harmonic function for each bone, which have a number of desirable properties [13], being:

- They are C^∞ over the surface of the mesh, at least as much as the discretization allows, and so will give smooth weights.
- The weights are all within the range 0..1. This follows from the property that harmonic functions reach their extrema at the boundaries, and since these are 0.0 and 1.0 here, all values are in this range.

- The weights at each vertex sum to 1.0. [13]
- The weights are local, decreasing as vertices get further away from the bone. This follows from the fact that a harmonic function reaches its extrema at the boundaries. The maximum 1.0 is reached at current bone vertex, and it decreases to the minimum 0.0 at all the other bone vertices.

Compared to enveloping, we do not need to define a bone's radius of influence. If some region of the mesh does not have associated bones, the influence of the nearest bones will be automatically extrapolated into this region, leaving no parts of the mesh behind when deforming.

We can see that the solution of these equations give good results, see figure 14, 15 and 16. Solving for these harmonic weights is clearly slower than enveloping, but efficient methods to solve laplacian systems exist, and this is a one time cost only, saving the user much time fixing up weights. Especially combined with dual quaternions or differential interpolation, we feel that these weights give a good initial deformation that requires few manual corrections, but still can be further tweaked by the user, whereas usually the user would have to start by fixing the automatically generated weights, and only after that customize them for the particular character they are deforming.

7 Other Deformation Methods

7.1 Overview

Various other methods exist to deform meshes besides skeletons based deformation. The classical example is freeform deformation (FFD) [26], which places a grid around the mesh. This grid has a number of control points that when moved have local effect the mesh. Their influence can be higher order, with higher order offering smoother results but less direct control.

The initial versions only allowed regular grids, but this has been extended to allow more flexible control point arrangements to fit the mesh better. An example of this is. Additionally, a wide range of other deformations exist, like curve based deformation, WIRE, .. [24].

These methods are commonly available in animation packages, and they are used as part of rigs in various ways, to improve on SSD, add muscle bulging, facial manipulation, and so on. We have implemented a recent method that is quite flexible in the placement of control points and cage structure, balancing the weighting from different control points smoothly, using again Laplace's equation. It can also handle multiple deformers, and ensure strict non-overlapping control of the mesh such that the deformers do not 'fight' and require counter-animation.

7.2 Harmonic Coordinates

A recent paper by Pixar presented a method to extend lattice deformation to arbitrary volumes [13]. Such methods are not completely new, as for example mean value weights [14] before demonstrated how smooth vertex - control point weights could be computed inside a cage mesh. The big advantage of such a method is that the cage can be modelled to fit the character, rather than having to use multiple overlapping lattices. The mean value control point weights can be computed using simple, closed form equations. This method however has some unacceptable disadvantages for character animation, namely that the influence of control points to a vertex solely depends on the direct distance from the vertex to the control point. This means that spatially nearby but unrelated control points will still influence the vertex, for example a control point for one the legs would influence the other leg too. Additionally the weights may become negative, resulting in unintuitive behavior as moving a control point will move some vertices in the opposite direction.

The harmonic coordinates method solves these issue by solving Laplace's equation, rather than relying on closed form solutions, providing both non-negative weights and local influence relative to the cage. Later, an alternative method called positive mean value weights has also been proposed [22], with similar properties and also without a closed form solution, but faster GPU accelerated computation. Even more recently, another closed form method [?] has been proposed that does suffer from negative weights and non local influence, but that interestingly is shape preserving, yielding purely rigid deformations. We note that using deformation gradients, it is also possible to turn any deforma-

tion into a rigid one, by extracting the rotation component from the deformation gradient and solving for the vertex coordinates, albeit at higher computational cost.

7.2.1 Formulation

Like the weight computation method for bones described in the previous chapter, this method takes advantage of the properties of harmonic functions, leading to the name *harmonic coordinates*. Conceptually the method is very similar, with two main differences:

- The equation is solved on a 3D volume surrounding the mesh rather than the mesh surface. This results in a higher computation time, but on the other hand makes it work well for disconnected components. Additionally, the solution becomes independent of the particular mesh or its deformation, so the cage can be reused for different resolution models or even different characters.
- Rather than bones we must find weights for control points.

We assume a closed cage mesh surrounding the character, with m vertices that we will call control points. We want to compute for each position p within the cage, harmonic coordinates or weights $h_j(p)$ for the control points C_j with $j = 1...m$. When used for interpolating the control point position at some p, they must yield that position:

$$p = \sum h_j(p) C_\mathrm{j}$$

Analogous to bone weights, we will solve one instance of Laplace's equation for each control point C_j. We start by defining the weights on the boundary of the cage. The scalar function $h_j(p)$ is 1.0 at the corresponding control point C_j and 0.0 at all the other control points. On the triangles between the control points, the weights are linearly interpolated like barycentric coordinates. Having fixed the weights on the boundary, we must now compute the weights inside the cage.

Some properties of harmonic functions such as C^∞ smoothness and non-negativity have been discussed in the chapter on computing vertex weights for bones. These hold here as well, and we mention two more that are relevant for the cage based formulation, the proof of which can be found in [13].

- The above mentioned formula $p = \sum h_j(p) C_\mathrm{j}$ holds for all points p inside the cage. For bone weights this is not generally possible, since the bones are inside the character instead of surrounding it, and so the mesh vertex positions can not interpolate the bone transformations. Here however it means that the vertex positions will not *pop* when the vertex positions are reconstructed from the control points and associated weights, but rather stay in exactly the same place.

- The weights are a strict generalization of barycentric weights, i.e. if the cage would consist of a single triangle, the weights would be identical.

7.2.2 Solving

To solve Laplace's equation we construct a three dimensional grid, embedding the cage. The resolution k of this grid will influence the accuracy of the result. A higher resolution will yield more exact results, but the number of grid vertices scales cubically with the the resolution, such that we will have to solve for the solution at k^3 grid vertices. The graph for the laplacian matrix consists of all grid vertices that are within the cage, and intersections of the grid edges with the cage triangles. Grid vertices out of the cage are not used. It is the values at the internal grid vertices that we wish to find for each control point C_i. The intersections of grid edges with the cage triangles are given fixed values, interpolated from the control point vertices with barycentric interpolation in the triangles.

Again, proper weighting for the laplacian is required. The edge between two vertices in the graph is given a weight inversely proportional to the distance between them. Between internal grid vertices the weight is constant for each dimension, while between an internal grid vertex and an intersection with the cage the distance will vary.

We then solve Laplace's equation with the weighting and boundary conditions as described above, the result of which is the weight for each vertex. Note that up to this point, we have weights at all grid points, but we have not used the character mesh yet. So, the solution is independent on the character mesh, only depending on the cage.

Next we have to compute the values for each vertex of the character. We do this through a lookup in the grid, finding the grid cell in which this vertex lies. Then we use bilinear interpolation of the weights at the 8 nearest grid vertices. Special care must be taken near the cage, when not all 8 grid vertices are within the cage, and hence were not solved for. In this case we normalize the weights to sum to one, although this violates the property that the weights should reproduce the original position. A better method could be used, such as using mean value weights for the convex polyhedron made up by the grid vertices withing the cage and the intersection points of edges with the cage.

7.2.3 Dynamic Binding

For integration with other deformation tools, a problem still remains. When the vertices are deformed by another operation before it, the control point weights no longer yield the original vertex position, which causes poor deformation results. A solution is to keep the weights for the whole volume around, and when deforming, lookup the weights again with bilinear interpolation in the grid. This is still efficient, and the solution can be compressed a lot by not storing weights below a certain small threshold, which keeps the number of control point weights

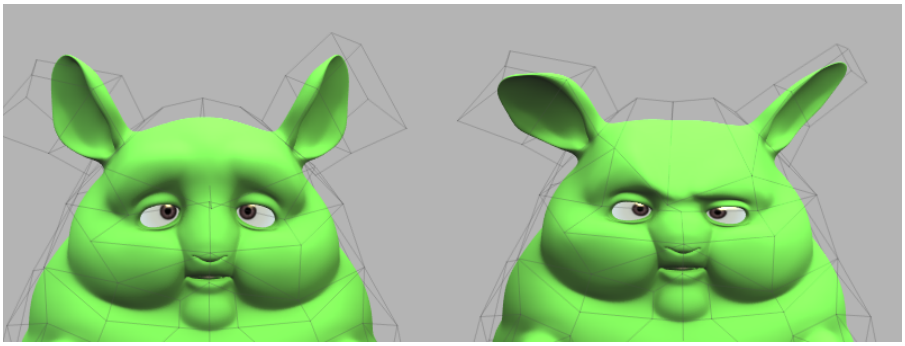

Figure 17: Facial deformation using harmonic coordinates.

per grid vertex reasonably low, as control points have a local influence. While the weights can be compressed a lot, the resulting grid at a high resolution still requires considerable amounts of memory. The disadvantage is that now vertices are required to stay within the cage even during animation, which may be hard to ensure in practice. An useful improvement would be a way to smoothly extend these coordinates outside of the volume, and this is provided by positive mean value coordinates, however how far these are to be extended volume must still be known a priori.

7.2.4 Results

Running the solver takes up a few seconds for cage resolutions of $2^4..2^5$ and a few minutes for resolutions of $2^6..2^7$ with cages of up to a few hundred vertices, the latter being sufficient for full characters. The solving is a one time operation after which the weights are stored. We solve Laplace's equation using a sparse direct solver, as explained in section 8.

This deformation has been tested in two ways, as a method to do high level facial deformations, and as a method to do skeletal deformation on' characters. The facial deformation cage provided interesting possibilities for squash and stretch in facial deformations, and as can be seen in figure 17. But the combination with skeletal deformation was particularly effective for 'fat' characters, where the skeletal bones do not match the mesh as well as a thinner character. The character was setup such that the cage is deformed by the skeleton, with the cage in turn deforming the character mesh. This provided a way for the influence of bones to be smoothed out, giving less self intersections and smooth higher order influences of bones, where as direct use of a skeleton would have required many corrective bones or shape keys. An example of such a cage an character can be seen in figure 18. As can be seen, the cage is used for the full body deformation. The detailed parts of the mesh are rigged to be deformed with regular skeletal deformation as the resolution required for solving at such small scale compared to the rest of the body would have been too high. Since these areas fit well to a skeleton this was not so much a problem, but an adaptive

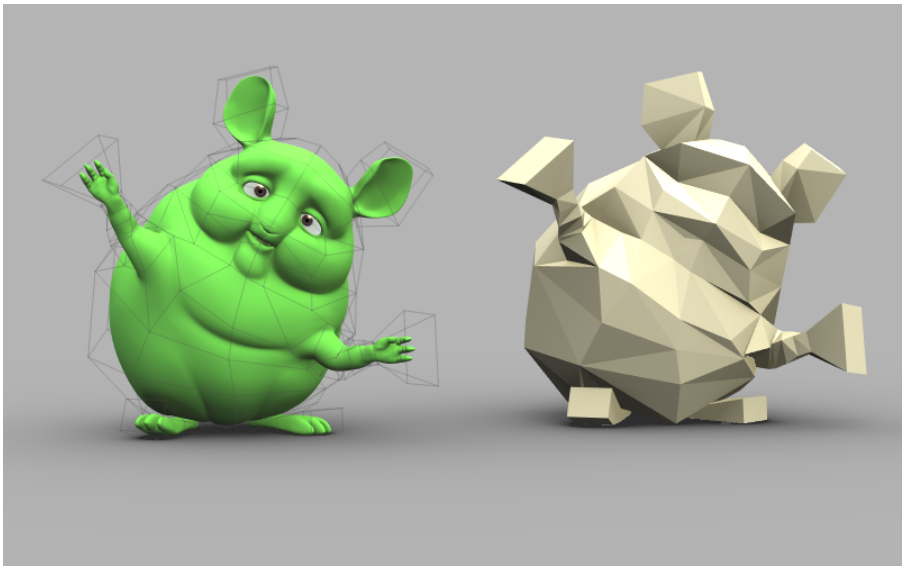

Figure 18: Character deformed by a the cage shown on the right.

grid would make it possible to also use cage deformation for these areas.

7.3 Inverse Deformation Tools

When editing skeletons, both forward kinematics and inverse kinematics are usually used. Instead of specifying for each bone how it is rotated, we can instead specify the position of each joint, and then automatically solve for the bone rotations. We can make the analogy in mesh deformation tools. Skeletal deformation or freeform deformation are more like forward kinematics, in that they control the vertex positions bone by bone or control point by control point. For example bones might be driven by inverse kinematics too at the skeletal level, but we can also do an operation analogous to inverse kinematics on the mesh itself. This is typically done by specifying the location for a subset of the vertices, and then automatically solving for the other vertex positions, preserving the original shape of the mesh as much as possible.

Recently a wide range of algorithms that tackle this problem have been proposed, for example [31, 27, 39], each trying to get better quality results with better performance. At this moment these algorithms are fast enough to run at interactive rates. How such tools integrate in a character animation pipeline however is not obvious. These algorithms offer great flexibility to the animator, with the possibility to edit the pose at the vertex level, and without the need to setup a skeleton. On the other hand, a character rig typically abstracts the mesh away by offering higher level controls to the animator. Hence it is interesting to

see how such tools can integrate with a classical character rigging setup. Four such applications in this context are:

- Animation of characters or props for which it is faster to use these mesh deformation tools than making a rig. If the character is only visible in a small part of the animation, or if a prop requires only simple animation, there is little point in developing a full rig, and these tools provide a quick way to articulate a character.
- Creation of shape keys by the rigger. Since mesh deformation tools are not limited to skeletal structures and require little setup time, they can be used to sculpt facial poses, or any other deformations that would be integrated into the rig.
- Extra control for the animator. On top of the controls provided by the rig, and extra layer of deformation could be used that allows the animator to get the shape just right, or to achieve an uncommon shape that was not taken into account when creating the rig.
- An alternative way to control the animation channels. For example in first the bone joints might be manipulated, while in another next on might manipulate mesh vertices. A two way coupling is then useful, so that manipulating the vertices would update the bone rotations and vice versa. The cascading solver proposed in [28] provides an integrated way of solving inverse kinematics on both a mesh and skeleton.

7.3.1 As Rigid As Possible

While a great variety of algorithms have been proposed and more advanced algorithms exist, we here describe a method [30] that is particularly simple, and allows us to reuse parts of the implementation of previous algorithms. The method attempts keep the deformation as rigid as possible relative to the original mesh. Such methods are necessarily non-linear and require multiple iterations to get to a solution. The input to the algorithm is a starting position of all vertices, and a new position for a subset of the vertices, as specified by the user. We will refer to this subset of vertices as constrained vertices. Our goal is then to find the positions of the other vertices, keeping the mesh as rigid as possible.

The methods works by preserving the rigidity of cells. Here each cell C_i consists of a vertex v_i at position p_i and the edges it is connected to. Let the neighbouring vertices be denoted v_j. Now assume that the vertex positions respectively at rest and after a deformation are denoted as p_i and p'_i. Then we can say the deformation of a cell is rigid, if a rotation matrix R_i exists such that the following sum is zero:

$$\sum_j w_{ij} \parallel (p'_i - p'_j) - R_i(p_i - p_j) \parallel$$

This means there is a rotation matrix that when applied to the original edges yields the deformed edges (disregarding translation). In practice we wish to minimize this quantity, while keeping the matrix R_i as rigid as possible. The weights w_{ij} give a different weight to each edge, the proper choice of which will be discussed later.

We now give an algorithm to minimize this quantity for all cells in the mesh, while obeying the constraints provided by the constrained vertex positions. Both R_i and p'_i will be solved for. This is done in multiple interleaved iterations, first computing the rigid rotation matrices R_i for each, then computing the p_i that obey these rotations as much as possible, then again the matrices R_i from the new positions, .. until convergence. Since this is an iterative algorithm, we need an initial guess of the vertex positions and cell rotations. There are different options, for example an estimate could be made based on a simpler linear deformation, though here we simply assume the current mesh shape to be the starting point.

We will not repeat the full derivation from [30] here, but rather the formulas most relevant to implementation. Computing the rotation matrix for a cell C_i is as follows. Given original vertex positions p and current vertex positions p', we compute the covariance matrix S_i from the vertex positions.

$$S_i = \sum_j w_{ij}(p_i - p_j) \cdot (p'_i - p'_j)^T$$

The rigid matrix R_i then follows from the singular value decomposition of the covariance matrix $S_i = U_i \Sigma_i V_i$.

$$R_i = U_i V_i^T$$

Next we plug these matrices into the right hand side of a Poisson equation to get the vertex positions. The elements in the right hand side vector f are computed for each vertex v_i as:

$$\sum_j \frac{w_{ij}}{2}(R_i + R_j) \cdot (p_i - p_j)$$

We then solve:

$$\triangle p' = f$$

The constrained vertices correspond to the boundary conditions on this system, and the solution gives the new vertex positions.

We have implemented this method as a way to sculpt example poses that may be integrated into the rig, not such much as an animator's tool. Any number of vertices may be constrained explicitly or implicitly by the fact that they are set as hidden. While the user is moving, the solver is continuously doing iterations, to keep the tool responsive without waiting for the full solution to be converged.

8 Implementation Issues

8.1 Non Manifold Meshes

While the definition of contangent weights is over a manifold surface, in practice we may still encounter non-manifold meshes, which might have three faces using one edge for example. In this case, we have extended the contangent weights for this purpose without much mathematical motivation, as follows. Remember the cotangent weights as specified in section :

$$u_i - \frac{1}{A_i} \sum_{j=1}^{n} w_{ij}(u_i - u_j)$$

$$w_{ij} = \frac{1}{2}(\cot \alpha_{ij} + \cot \beta_{ij})$$

Rather than computing the voronoi area and weights per vertex, we accumulate them per face, which generalizes to non manifold meshes, and avoids us having to use a mesh representation with vertex to face connectivity information. We loop over each face and do:

- Compute voronoi area divided by two for each vertex of the face.
- For each vertex in the face:
  - Compute cotangent of the angle.
  - Add cotangent multiplied by the voronoi area, to the corresponding matrix elements of the two opposite vertices. One of the vertices gets a negated weight, depending on the face orientation.

Construction of the laplacian matrix in this way is equivalent and extends to non-manifold meshes. Support for such non-manifoldness can be exploited by the artist to achieve better results. For example simple volume preservation may be preserved by creating internal faces, as used in [38] for example. This comes at little extra solving time, however they must be manually created by the user.

Our solver currently ignores loose edges, which would be easier to create than internal faces. Additionally, automatic creation of such internal edges could provide a simple solution for volume preservation with little extra solving time and effort by the user.

8.2 Multiple Disconnected Components

Since meshes do not necessarily exist as a single connected component, which we have assumed before for the most part, we have to take this issue into account. Given that the laplacian is translation independent, we must constrain at least on vertex per component. So we detect disconnected components explicitly and

solve a separate linear system of equations for each of them. This however has the disadvantage that components are in no way related, and will not influence each other, which isn't always desirable. Eyes should preferably stay in their sockets, and rings should stay on fingers, even if they are modelled as separate components. One solution is to push the problem to the user, and let them make sure that these components are connected in the mesh or manually adjust the weights, which is what we assume in our implementation. Another possibility would be to connect these components automatically in some way, although it is probably still desirable for the user to be able to specify which ones should be connected and which ones not, without having to resort to changing the mesh model as is required now.

8.3 Linear System Solving

Solving sparse systems of linear equations is a well studied problem with many applications. Given that A is a sparse matrix, i.e. a matrix that has only a few non zero elements per row or column and a vector b, we wish to find the vector x such that:

$$Ax = b$$

Various approaches exist, the relevant ones being [8]:

- Conjugate gradient: an iterative method that can be implemented in terms of sparse matrix-vector multiplications, which can be parallelized well.
- Multigrid: also an iterative method, that requires the domain to be available in a number of coarse-to-fine resolutions. By solving first at the coarse resolution level, the solution can be reached quicker at the fine resolution since the error can be diffused faster at the coarse resolution. Again, multigrid methods tend to be well parallelizable. Multigrid typically scales better than conjugate gradient methods, as error diffusion becomes slow at high resolution for the latter.
- Sparse Direct Solvers: these factorize the matrix A into upper and lower triangular matrix, examples of which are LU or Cholesky factorization. The solution is then computed by back and forward substitution, which computes the solution row after row. An advantage is that the factorization can be reused for the next solve. This can make solving again faster than other methods, since factorization is the slowest step, and back and forward substitution can be implemented efficiently. However, the latter algorithms are hard if not impossible to parallelize efficiently on current systems.

We use the sparse direct solver SuperLU for our implementation, which at least in this comparison [8], can be seen to have the fastest solution time once the matrix is factorized. The disadvantage is that this step cannot take advantage

of multiple threads. As can be seen in the statistics from the referenced paper, SuperLU is still 8x faster on meshes with 10k vertices, which corresponds to the resolution of the meshes we have encountered in the production of an animated short movie.

A bigger problem showed up in the implementation of harmonic coordinates. While the SuperLU solvers was very fast for reasonable grid resolutions, for higher ones like $2^8 \times 2^8 \times 2^8$, corresponding to 16777216 grid vertices, the matrix factorization did not fit in memory even on a system with 8 GB ram, causing heavy disk swapping and very slow performance. We estimate a multigrid solver would scale better to such a resolution, also allowing the solution to be solved only local in the region near the control point. With the need for such large resolutions, adaptive grid resolution where it is required would probably be the best solution, which can be used also with sparse direct solvers.

8.4 Using the Graphics Card

Fast linear (re-)solvers exist on the CPU, but the vast number of floating pointer operations per second that current graphics cards offer are interesting to improve performance even further. For example games commonly do mesh skinning on the graphics card in vertex shaders, to improve performance and avoid data transfer. We would like to do the same thing for differential skinning methods. Solving a linear system of equations with a parallel processor is more complicated however.

The most successful linear system solving method implemented on the GPU are conjugate gradient and multigrid [7][10]. Methods like cholesky factorization may also be implemented on the GPU [15]. However backward and forward substitution are difficult to parallelize. The computations for each row/column depend on the previous one. Parallelizing the computations within each row appears to require too much communication overhead between threads to be efficient.

And this is where the problem lies. Ideally we would like to do the same as the CPU, factorize the matrix in advance and solve again quickly afterwards. But conjugate gradient and multigrid need to start from scratch each time. Even with the vast amount of FLOPS on the GPU, the extra computations required are not (yet) offset by the increased computational power. Nevertheless, given the increase in the number of cores rather than per-core performance in both CPU's and GPU's, we expect multigrid methods to become faster over time.

We note that some algorithms [39, 34] have successfully used linear system solving on the GPU, by storing the full inverse matrix as a dense matrix. Clearly this scales poorly to high resolution meshes, but as long as the number of vertices is low enough (up to 1k in the referenced papers), a multiplication with such a dense matrix is still very fast due to the high bandwidth and FLOPS in the GPU.

8.5 Mesh Size and Multiresolution

The methods in this text that depend on solving sparse linear systems cannot run in realtime (24-25 fps) as part of a larger rig with very high resolution. The character we used all had 10k or less vertices, which we believe is not an uncommon amount for animated movies. It would however be interesting to apply multiresolution solving methods to optimize both the cases that already fast and to allow more complex meshes.

We do not though that the use of subdivision surface and displacement mapping are very common in animation productions, and that they allow both faster performance and leave less vertices to manipulate for the rigger. Hence in some way multiresolution is already a part of many animation pipelines anyway, as very high resolution are difficult to manipulate, which works out in favor of these more computationally intensive skinning methods.

If there are more vertices in the mesh than there is detail in the deformations, it is also possible to solve the problem for a subspace of the original mesh. That is, we can use a lower resolution representation that is linked to the original high resolution mesh in some way, found for example by clustering vertices or faces which deform similarly in a set of example poses. The solution of the low resolution model can then be transferred to the high resolution, see for example [11, 34].

8.6 Order of Deformations

To get correct results, the order of deformations matters. For example in a classical setup, shape keys will be applied before skeletal deformation. This is because the simple interpolation of vertex displacements in object space for shape keys is not invariant to rotation of the mesh, and so if it would be applied after SSD, they would displace in the wrong direction compared to the surface. If the displacements are stored in tangent space or if we use deformation gradients, this is less of a problem, and the order can be switched more easily, although this will still give different results.

8.7 Editing Deformed Meshes

It is often useful to edit a mesh while it is deformed. For example corrective shape keys could be edited while the mesh is deformed by an armature, to immediately see the results. Depending on the ordering of deformation operations, this may pose a problem. If shape keys are applied before some other deformation, and we are transforming the vertex coordinates as if they were undeformed, we will get unintuitive results. A deformation that rotates the mesh will cause moving the vertices in one direction them to go in another, which has been dubbed *crazyspace* since it how vertices will move is unpredictable for the user. If we want the result of the deformations to correspond exactly to what the user is editing, we need to have an inverse deformation function. That is we need to be able to compute original vertex coordinates x such that given deformed

coordinates y are the result. If f is a deformation function, that means we need to compute:

$$x = f^{-1}(y)$$

For typical SSD, also with dual quaternions or curves interpolation, or harmonic coordinates with static binding the solution is simple. Each vertex is deformed by a $4x4$ matrix in a given pose, so all we need to do is invert that matrix. For FFD or dynamically bound harmonic coordinates there is however no trivial inverse, in fact these functions are not bijective in general, and so multiple solutions might be possible. A solution to this problem is to iteratively minimize $\|f(y) - x\|$ in some generic way, for example using Powell's method [20].

Our implementation only provides correct deformed editing for the simple linear cases without using a generic minimization methods, which were found to be too slow when applied to full meshes. With some effort this implementation could be optimized to only do computations for vertices that are actually being edited, and a faster and more stable optimization method rather than the simple Newton-Raphson method we tested could be implemented.

9 Conclusion

9.1 Results

The animated short movie *Big Buck Bunny* [3] used some of the algorithms and implementation discussed in this text, and all algorithms were implemented in the 3D animation software Blender [1]. The automatic method to compute harmonic vertex weights was used as the first weight assignment, which was improved by manual painting later. It was found that this method permitted adding many bones more easily for example for facial animations, whereas much manual painting made this a less appealing option before. Dual quaternions were use for transformation blending in skeletal subspace deformation on all characters, requiring fewer corrective shape keys. Cage based deformation based on harmonic coordinates was used for the body of the "fat" characters, and for facial deformation on most characters. A setup with a skeleton controlling the cage deforming the character, quickly gave better results than the manually created corrective bones and shape keys that were laboriously added before.

9.2 Future Work

We have demonstrated how differential equations can be used both to compute influence weights, and to yield deformations that are either shape and volume preserving depending on the domain in which the equations are solved. We believe many improvements are still possible in terms of scalability and performance. Multiresolution solving, adaptive resolution and GPU acceleration would make these methods more interactive and scalable to complex characters. It should also be possible to construct an adaptive volumetric laplacian graph that can guarantee volume preservation and at the same time solve the issue of translation independence between loose components, while avoiding the high grid resolution required by harmonic coordinates.

Further, we think physical simulation methods would be the logical next step to improve upon this system, where the interesting challenge is to keep them interactive and easily controlled by the user, while still automating deformations that would be difficult to animate manually.

Figure 19: Relatively 'extreme' poses for the bunny using harmonic coordinates. The bunny deformations were especially difficult given the fat and stumpy arms, resulting relative little flexibility, which gave many self intersections with regular skeletal deformation. Harmonic coordinates smooth out the deformations in such problematic areas.

Figure 20: Demonstration of mesh deformation in various shots. The squash and stretch in the faces is accomplished with harmonic coordinates, the skeletal deformation for the character is done with dual quaternions. The blue chinchilla is completely deformed with harmonic coordinates, except for the fingers.

References

[1] Blender, http://www.blender.org/.

[2] Discrete differential geometry: an applied introduction. *ACM SIGGRAPH Courses*, 2006.

[3] Big Buck Bunny, http://www.bigbuckbunny.org/, 2008.

[4] Brett Allen, Brian Curless, and Zoran Popović. Articulated body deformation from range scan data. *ACM Transactions on Graphics*, 21(3), 2002.

[5] Alexis Angelidis and Karan Singh. Kinodynamic skinning using volume-preserving deformations. In *ACM Symposium on Computer animation*, 2007.

[6] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. *ACM Transactions on Graphics*, 26(3), 2007.

[7] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroeder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3), 2003.

[8] Mario Botsch, David Bommes, and Leif Kobbelt. Efficient linear system solvers for mesh processing. In *IMA Conference on the Mathematics of Surfaces*, volume 3604, 2005.

[9] Mario Botsch, Robert Sumner, Mark Pauly, and Markus Gross. Deformation transfer for detail-preserving surface editing. 2006.

[10] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent number cruncher - a gpu implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, 2008.

[11] Kevin G. Der, Robert W. Sumner, and Jovan Popović. Inverse kinematics for reduced deformable models. In *ACM SIGGRAPH*, 2006.

[12] F. Sebastian Grassia. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3), 1998.

[13] Pushkar Joshi, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. Harmonic coordinates for character articulation. *ACM Transactions on Graphics*, 26(3), 2007.

[14] Tao Ju, Scott Schaefer, and Joe Warren. Mean value coordinates for closed triangular meshes. In *ACM SIGGRAPH*, 2005.

[15] Jin Hyuk Jung. Cholesky decomposition and linear programming on a gpu. *Scholarly Paper, University of Maryland.*

[16] Michael Kass, Aaron Lefohn, and John Owens. Interactive depth of field using simulated diffusion on a gpu, 2006.

[17] Ladislav Kavan, Steven Collins, Jiri Zara, and Carol O'Sullivan. Skinning with dual quaternions. In *SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2007.

[18] Tsuneya Kurihara and Natsuki Miyata. Modeling deformable human hands from medical images. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2004.

[19] John Lasseter. Principles of Traditional Animation Applied to 3D Computer Animation. *ACM SIGGRAPH*, 21, 1987.

[20] J. P. Lewis. Pose space deformation notes. 2000.

[21] J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformations: A unified approach to shape interpolation a nd skeleton-driven deformation. In *SIGGRAPH*, 2000.

[22] Yaron Lipman, Johannes Kopf, Daniel Cohen-Or, and David Levin. Gpu-assisted positive mean value coordinates for mesh deformations. In *Eurographics symposium on Geometry processing*, 2007.

[23] Mark Meyer, Mathieu Desbrun, Peter Schroeder, and Alan Barr. Discrete differentialgeometry operators for triangulated 2-manifolds, 2002.

[24] Rick Parent. *Computer animation: algorithms and techniques.* Morgan Kaufmann Publishers Inc., 2001.

[25] K. Polthier and E. Preuss. Identifying vector fields singularities using a discrete hodge decomposition, 2002.

[26] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *ACM SIGGRAPH*, 20(4), 1986.

[27] Alla Sheffer and Vladislav Kraevoy. Pyramid coordinates for morphing and deformation. In *Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, 2004.

[28] Xiaohan Shi, Kun Zhou, Yiying Tong, Mathieu Desbrun, Hujun Bao, and Baining Guo. Mesh puppetry: cascading optimization of mesh deformation with inverse kinematics. In *ACM SIGGRAPH*, 2007.

[29] Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. In *Proceedings of Graphics Interface*, 1992.

[30] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Proceedings of the fifth Eurographics symposium on Geometry processing*, 2007.

[31] Olga Sorkine, Yaron Lipman, Daniel Cohen-Or, Marc Alexa, Christian Rössl, and Hans-Peter Seidel. Laplacian surface editing. In *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, 2004.

[32] Jos Stam. Stable fluids. In *ACM SIGGRAPH*, 1999.

[33] Robert W. Sumner and Jovan Popović. Deformation transfer for triangle meshes. *ACM Transactions on Graphics*, 23(3), 2004.

[34] Robert Y. Wang, Kari Pulli, and Jovan Popović. Real-time enveloping with rotational regression. In *ACM SIGGRAPH*, 2007.

[35] Max Wardetzky, Saurabh Mathur, Felix Kalberer, and Eitan Grinspun. Discrete laplace operators: no free lunch. In *Symposium on Geometry Processing*, 2007.

[36] Ofir Weber, Olga Sorkine, Yaron Lipman, and Craig Gotsman. Context-aware skeletal shape deformation. *Proceedings of Eurographics*, 26(3), 2007.

[37] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Mesh editing with poisson-based gradient field manipulation. In *ACM SIGGRAPH*, 2004.

[38] Kun Zhou, Jin Huang, John Snyder, Xinguo Liu, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Large mesh deformation using the volumetric graph laplacian. *ACM Transactions on Graphics*, 24(3), 2005.

[39] Kun Zhou, Xin Huang, Weiwei Xu, Baining Guo, and Heung-Yeung Shum. Direct manipulation of subdivision surfaces on gpus. *ACM Transactions on Graphics*, 26(3), 2007.

10 Samenvatting

10.1 Character Rigging

Character rigging of articulatie is het proces waarbij een statisch model omgezet wordt in een model dat geanimeerd kan worden, met controls voor de animator. In plaats van bijvoorbeeld individuele vertex posities te manipuleren, wordt een structuur aan het model toegevoegd met controls voor lachen, het verplaatsen van handen of het buigen van spieren.

Het maken van character rigs kan een full time job zijn in de productie van een animatie. De rig moet eenvoudig te gebruiken zijn en de irrelevante details verbergen van de animator, wat belangrijk is aangezien naast het animatie programma, de character rig in feite de user interface is waar hij dag in dag uit mee werkt.

Een character rig neem een aantal geanimeerde waarden en de originele mesh als invoer, en produceert dan een vervormde mesh in de pose gemaakt door de animator. In dit werk zullen we methodes bespreken voor het vervormen van een mesh, van zijn rust pose tot vervormde pose met puur geometrische/kinematische methoden, zonder animatie van skeletale structuur of fysische simulatie en rendering te beschouwen. Uitgaande van bestaande, klassieke methode tonen we aan hoe deze verbeterd kunnen worden zonder grote aanpassing in de character rig setups.

10.2 Mathematische Tools

Affiene transformaties zijn een belangrijk element in het vervormen van modellen. De interpolatie, extrapolatie en manipulatie ervan zal worden gebruikt in de verschillende algoritmen die we bespreken. Correcte interpolatie vereist een decompositie van deze transformaties in verschillende componenten: rotatie, schaling en translatie. Individuele interpolatie van deze componenten levert beter resultaat op in enkele algoritmen die we bespreken. Vooral de interpolatie van rotaties is belangrijk, en deze kan bijvoorbeeld beter met quaternions uitgevoerd dan door direct matrices te interpoleren.

Een andere minder voor de hand liggende mathematische methode die we zullen gebruiken zijn differentiaal vergelijkingen. Meer specifiek vergelijkingen met de Laplaciaan, Gradient en Divergentie operators op 3D volumes en mesh oppervlakken. Deze operators en bepaalde vergelijkingen waarin zij voorkomen hebben nuttige eigenschappen, zoals C^∞ smoothness. Voor het oplossen van zulke vergelijkingen in de praktijk zullen we de operators discretizeren, bijvoorbeeld over de vertices of driehoeken in een mesh, of over een 3D grid met cellen. Dit levert dan een verzameling van lineaire vergelijkingen op die efficiënt kunnen opgelost worden. Verder van belang zijn ook de boundary conditions, waarmee constraints kunnen worden toegevoegd worden om de oplossing te manipuleren.

10.3 Shape Interpolatie

Interpolatie van verschillende vooraf gemaakte deformaties (shape keys) van een mesh wordt veel gebruikt in character rigging, voor verschillende doeleinden. Dit gaat van volledige gezichtsexpressies tot het buigen van spieren of correcties van deformaties met andere algoritmen. Deze shape keys worden vaak manueel door de animator gecontroleerd, maar kunnen ook automatisch geactiveerd worden als onderdeel van de rig, gebaseerd op de huidige pose van het skelet.

Eenvoudige interpolatie van de vertex posities wordt gewoonlijk gebruikt. Een alternatief is interpolatie in tangent space, waardoor het resultaat onafhankelijk wordt van de globale rotatie van het model. Wij bespreken een derde methode die in plaatse van vertex translaties, de deformatie encodeert in affiene transformaties per driehoek. Op deze manier kan een rotatie daadwerkelijk als een rotatie worden geïnterpoleerd, wat de interpolatie en extrapolatie van grote deformaties meer intuïtieve resultaten geeft. De resulterende vertex posities verkrijgen vereist het oplossen van een differentiaal vergelijking, dewelke resulteert uit de formulatie van het probleem in termen van kleinste kwadraten. Deze methode is significant trager dan eenvoudige lineaire interpolatie van translaties, maar kan interactief worden uitgevoerd door gebruik te maken van het feit dat de vergelijkingen snel opnieuw opgelost kunnen worden eens zij eenmaal zijn opgelost.

10.4 Skelet Gebaseerde Deformatie

Deformatie van een mesh met een skelet bestaande uit individuele bones is de standaard methode voor het animeren van modellen. De typische methode die wordt gebruikt is skeletal subspace deformation (SSD), maar deze lijdt aan het ineenzakken van modellen en verlies van volume rond de verbindingen tussen twee bones. Een eenvoudige analyse van het probleem toont aan dat SSD neerkomt op lineaire interpolatie van affiene transformaties, en als zodanig kan een methode die deze transformaties correct interpoleert rekening houdend met rotaties een betere oplossingen geven. Een recente methode, dual quaternions, biedt een efficiënte oplossing op het probleem van het kiezen van een centrum van rotatie, wat tot nu toe het gebruik van betere transformatie interpolatie onpraktisch maakte. Verder is het eveneens mogelijk curves te gebruiken voor betere interpolatie, of interpolatie per driehoek uit te voeren met de methode uit de vorige sectie om de problem van SSD op te lossen.

Voor SSD en aanverwante methoden is het nodige de invloed van de bones op elke vertex te specifiëren. Dit gebeurt gewoonlijk eerst automatisch waarna deze manueel aangepast worden. Wij formuleren een methode gebaseerd op de oplossing van een Laplace's vergelijking over het oppervlak van de mesh. Deze resulteert in gewichten die zorgen dat de invloed van bones smooth met elkaar blenden. Voor elke bone wordt de vergelijking opgelost door de 'temperatuur' van deze bone op 1.0 te zetten en alle andere bones op 0.0, zodat deze temperatuur zich vanuit die bone verspreid over de mesh. Dit wordt gedaan door deze als boundary conditions van de vergelijking te specifiëren. Er wordt van

raytracing gebruik gemaakt om te bepalen welke bones welke delen van de mesh direct beïnvloeden. Het resultaat is dat de mesh automatisch aan een skelet kan gebonden worden, met een goede standaard deformatie, waarin weinig manuele correcties gemaakt moeten worden.

10.5 Andere Deformatie Methoden

Freeform deformation is een veelgebruikte methode voor het vervormen van meshes, waarbij een cage wordt geplaatst rond een model. De manipulatie van de controlepunten van de cage resulteert dan in deformaties van het model. We bespreken een veralgemening van deze methode gebaseerd op differentiaal vergelijkingen, genaamd harmonic coordinates. Deze methode maakt het mogelijk de cage arbitraire vormen en controlepunten te geven, in plaats van restricties op te leggen op de vorm. De invloed van verschillende controlepunten of zelfs overlappende cages wordt automatisch gebalanceerd door de oplossing van de differentiaal vergelijking.

De methode discretiseert Laplace's vergelijking over een 3D grid binnen de cage. De invloed van elk controlepunt wordt individueel berekend met het oplossen van de vergelijking door de gewichten op deze punten op 1.0 te zetten, en 0.0 op alle andere punten. De oplossing geeft dan de waarden op de tussenliggende punten. Eens de oplossing berekend is op alle punten in de cage kan voor elke vertex in de mesh de invloed van de controlepunten berekend worden. Er kan aangetoond worden dat als we deze gewichten gebruiken voor het interpoleren met de posities van de controlepunten, de oorspronkelijke positie gereproduceerd wordt van alle punten in de kooi. Bij het vervormen van de cage wordt de nieuwe positie van een vertex dan berekend door eenvoudigweg met de berekende gewichten en nieuwe posities van de controlepunten te interpoleren.

Het resultaat is dus een alternatief voor freeform deformatie dat flexibeler is, en gebruikt kan worden om een cage te maken rond volledige character meshes, in plaats van het typische combineren van overlappende reguliere grids. Er kunnen bijvoorbeeld flexibele gezichtsuitdrukkingen gemaakt worden met deze methode, door een cage te modelleren rond het hoofd van het model. Ook kan bijvoorbeeld de volledige mesh vervormd worden met freeform deformatie, door een skelet de cage te laten vervormen, die dan weer de finale mesh vervormt.

In de laatste jaren heeft ook een een type deformatie methode veel aandacht gekregen, waarbij de mesh gemanipuleerd worden door bepaalde vertices van de meshes te manipuleren die dan indirect de rest van de mesh mee vervormen, op een natuurlijke wijze. Dit is in feite inverse kinematics voor meshes. We hebben een specifieke methode geïmplementeerd die nauw gerelateerd is met de methodes die we reeds besproken hebben. De methode is gebaseerd op het behoudt van rigiditeit in de mesh. Een affiene transformatie wordt berekend per vertex door de vervorming van de omliggende vertices te in acht te nemen, en deze transformatie wordt dan telkens rigide gemaakt. Om de vertex posities te constrainen wordt Laplace's vergelijking met boundary conditions gebruikt, terwijl deze tevens de rigide transformaties zo goed mogelijk probeert te re-

specteren. Op deze manier worden om beurt de transformatie rigide gemaakt en de posities van de tussenliggende vertices opgelost, tot convergentie. Deze methode blijkt vooral nuttig voor het maken van shape keys die dan in een rig gebruikt kunnen worden, aangezien ze niet efficiënt genoeg is voor het realtime oplossen als onderdeel van een volledige character rig.

10.6 Implementatie Details

De implementaties van de algoritmen die we besproken hebben een aantal aspecten gemeen. Het oplossen van alle differentiaal vergelijkingen wordt met een sparse direct solver uitgevoerd, gebaseerd op LU decompositie van sparse matrices. Er wordt geen gebruik gemaakt van de grafische kaart, en we verwachten dat met het gebruik van multiresolutie verwachten de GPU benut kan worden en de vermelde algoritmen geoptimaliseerd kunnen worden.

Voor het oplossen van de differentiaal vergelijkingen over mesh oppervlakken, worden ook non-manifold meshes en meerdere losse componenten ondersteund. Non-manifold meshes worden ondersteund door de standaard Laplaciaan operator over mesh oppervlaken te veralgemenen.

10.7 Conclusie

De geanimeerde kortfilm Big Buck Bunny maakt gebruikt van enkele algoritmen en implementaties besproken in deze tekst. De automatische vertex gewichten voor skeletale deformaties werden gebruikt als vertrekpunt voor alle modellen, waardoor minder manuele correcties nodig waren. Dual quaternions werden gebruikt voor skeletale deformatie op alle modellen, wat resulteerde in minder manuele correcties voor het verlies van volume bijvoorbeeld. Harmonic coordinates werd gebruikt voor de deformatie van zwaarlijvige karakters, aangezien dit minder scherpe deformaties geeft en zo zelf-intersecties vermijd. Tevens werden deze gebruikt voor gezichten voor het maken van expressies en squash and stretch deformaties.

Een belangrijke verbetering die aan het systeem gedaan zou kunnen worden is op het vlak van performantie, door het gebruik van multiresolutie en GPU acceleratie. Voor de modellen in Big Buck Bunny die 1000 tot 10000 vertices hadden was dit niet nodig, maar, voor meer gedetailleerde modellen, complexere rigs en meerdere karakters in een scene zijn performantie verbeteringen handig. Verder zou het expliciet voorkomen van verlies van volume, en efficiënte fysisch gebaseerd simulaties de resultaten verder verbeteren, indien deze op interactieve snelheid kunnen werken voor complexe rigs.