Technical Report

UCAM-CL-TR-723
ISSN 1476-2986

Number 723

Animation manifolds for representing topological alteration

Richard Southern

July 2008

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted February 2008 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

An animation manifold encapsulates an animation sequence of surfaces contained within a higher dimensional manifold with one dimension being time. An iso–surface extracted from this structure is a frame of the animation sequence.

In this dissertation I make an argument for the use of animation manifolds as a representation of complex animation sequences. In particular animation manifolds can represent transitions between shapes with differing topological structure and polygonal density.

I introduce the animation manifold, and show how it can be constructed from a keyframe animation sequence and rendered using raytracing or graphics hardware. I then adapt three Laplacian editing frameworks to the higher dimensional context. I derive new boundary conditions for both primal and dual Laplacian methods, and present a technique to adaptively regularise the sampling of a deformed manifold after editing.

The animation manifold can be used to represent a morph sequence between surfaces of arbitrary topology. I present a novel framework for achieving this by connecting planar cross sections in a higher dimension with a new constrained Delaunay triangulation. Topological alteration is achieved by using the Voronoi skeleton, a novel structure which provides a fast medial axis approximation.

Acknowledgements

Ideas are elusive, often only exposed through fruitful discussions. During the course of undertaking this research I have had countless interactions with members of the Computer Laboratory, researchers at Cambridge and others across the globe, each of which have helped to dislodge productive thoughts from within my jumbled head.

My supervisor, Neil A. Dodgson, has always offered his support throughout this odyssey, encouraging me to discover my own voice and providing indispensable advice where needed. My enormous thanks go to Malcolm Sabin for the incalculable assistance which he has provided in the development of the fundamental concepts of Animation Manifolds. I know of no other person with a comparable, almost supernatural, intuition for geometry and its practicalities.

Julian Smith, Tom Cashman and Ursula Augsdörfer have all provided assistance in tackling a variety of problems, from technical issues such as programming or mathematics problems, to discussions on grammatical correctness. I would also like to extend my thanks those with whom I have consulted: Patrick Campbell-Preston, Friedel Epple, Dominique Bechmann, Hang Si, Alan Blackwell and Tamil Dey. In addition, I would like to thank my examiners for their particularly constructive criticism.

This work would not have been possible without the generous financial assistance of the EPSRC, the Cambridge Commonwealth Trust and Clare Hall college.

A special thank you to the friends, loved ones and housemates (who are a combination both) who have showered me with kindness and support through the tough spells, especially as the end was nearing.

Finally this work is dedicated to my family, without whose unending support and motivation I would surely have lapsed into madness.

Contents

List of Figures

Chapter 1

Introduction

Computer animation has become indispensable in many forms of entertainment, from computer games to films, and is consequently a major driving force in computer graphics research. However there remain challenging questions relating to animation which have not yet been adequately answered.

1.1 Animation representations

The reader may be familiar with standard representations of animation sequences in the industry. An animation consists of two or more keyframes, specified at particular points in time, and rules for defining transitions between these keyframes. The keyframes should be exactly reproduced at the specified time instances in the animation sequence. Animation of surfaces is achieved by defining time dependent transformations for each vertex. An obvious limitation to these standard approaches is that vertices cannot be created or destroyed.

This limitation presents difficulties with modelling topological alteration during animations, such as separating a model into two disjoint parts or inserting holes in a surface while it is animating. Morphological changes in the animation are limited to surfaces which are topologically equivalent and have the same connectivity attributes. Methods exist to perform topological alterations, but these operations are currently impossible to integrate into a standard keyframe animation.

A second, more common problem facing animators is that severe extrusion or folding of geometry may cause self–intersections, folding and overlapping faces. These lead to poor visual fidelity, often caused by geometric and texture map distortion. An example of this is the polygonal bunching artifact (shown in Figure 1.1).

Avoiding this kind of artifact requires adaptive changes in polygon density through subdivision and simplification. However, these changes will require a break in the animation sequence, as the model is replaced with a denser or simpler level of detail.

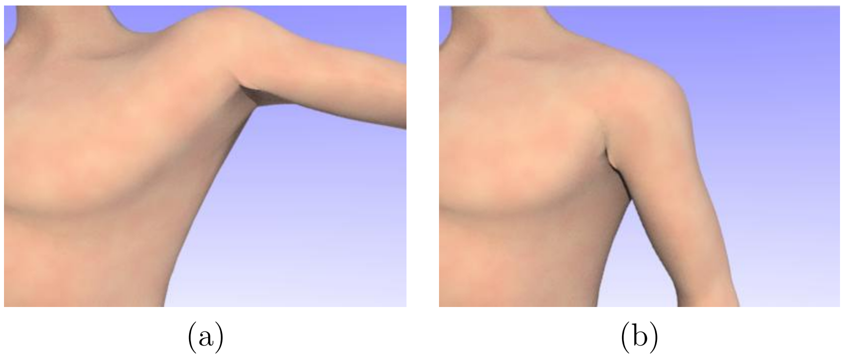


Figure 1.1: An artifact caused due to polygonal bunching. In (a), the character is initially modeled in a rest state with outstretched arms. However, when the arm is folded, there are two many polygons in the arm pit, causing polygons to bunch together, overlap and possibly self–intersect. This problem commonly arises in character modelling.

1.2 What is an animation manifold?

An animation manifold is a space-time representation of an animation sequence. A keyframe is a spatial representation of an animation sequence at a particular point in time. These are embedded into a unified space-time representation. The region between these keyframes is filled using a higher dimensional simplicial boundary representation.

Cross-sections extracted from this boundary representation at different time instances yield frames from the animation sequence. An animation manifold can be thought of as the boundary formed by stacking the continuous set of consecutive frames of an animation sequence on top of each other in the time dimension.

In Chapter 2 I formalise the definition of the animation manifold. In addition, I show how it can be simply constructed from a sequence of keyframes, and how it can be rendered with both a traditional raytracing approach or with graphics hardware. Chapter 3 covers related work in the field of space–time representations, with reference to applications in computer graphics.

I develop two approaches to altering the topology of an animated surface within the animation manifold:

- A user guided approach based on free form deformation, presented in Chapter 4.
- An automatic approach using the geometric properties of the shapes, presented in Chapter 8. This approach makes use of the constrained Delaunay algorithm of Chapter 5, the mesh refinement algorithm of Chapter 6 and the skeleton extraction method of Chapter 7.

The methods presented in this dissertation apply only to closed surfaces in 2D and 3D, specifically polygons in $\mathbb{R}^2$ and triangle meshes in $\mathbb{R}^3$.

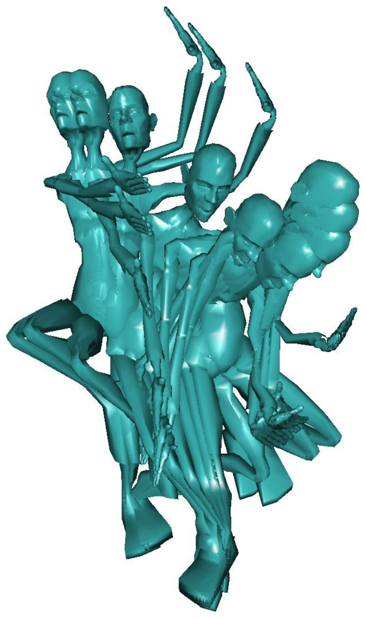


Figure 1.2: Emo's dance sequence from the short film *Elephants Dream* (`http://www.elephantsdream.org`). Several frames are rendered together, to give a visualisation of the animation manifold.

1.3 Applications of animation manifolds

By contrast with the lack of flexibility of traditional animation sequence representations, animation manifolds have some powerful features which them of particular interest in Computer Graphics. Faces and vertices can be created or destroyed at any point of the animation sequence, allowing these manifolds can represent transformations between levels of detail, and may represent the transformation between surfaces of differing topologies.

1.3.1 Topological alteration

Topological alteration is a challenging problem in any discipline, and particularly in computer animation. A standard animation object *cannot* alter its topology. Other representations are employed to model these changes, such as metaballs or constructive solid geometry, but these require the animation object to be approximated with one of these primitives before topological alteration can occur.

Animation manifolds can represent an animation sequence and can represent *any* topological alterations between closed manifold shapes. In this dissertation I will present two methods by which this can be achieved – a user guided surface deformation approach, and an automatic construction technique.

Altering topology by deformation

Surface deformation is commonly used in 3D graphics. While numerous paradigms exist for performing these operations, not all are applicable to editing higher dimensional manifolds.

In Chapter 4 I adapt the Laplacian mesh editing paradigm to a platform for user–guided topological modeling. Laplacian surface editing is unique in that it supports a large number of editing paradigms, is feature sensitive, and user edited results also have an “organic”, clay–like feel.

I adapt three surface deformation techniques to general dimensional simplicial deformation techniques by defining boundary and extrusion rules, and apply these to the deformation of animation manifolds.

Morphing between surfaces of arbitrary topology

The reconstruction of surfaces from cross–sections is a well studied topic, and is employed in many disciplines ranging from cartography to medical visualisation. The extrapolation of these techniques to animation manifolds by adding an extra dimension, however, is surprisingly difficult. This stems from the lack of an orientable structure — curves have a natural direction or ordering which makes connecting them relatively simple. Surfaces have no such discernable ordering.

In Chapter 8 I present a method which can automatically build morph sequences between surfaces of arbitrary topology using a Delaunay triangulation based space filling algorithm. Several components are required to first build the machinery for this approach. In Chapter 5, with Theorem 5.2.1, I provide a novel method for constructing a higher dimensional constrained Delaunay triangulation. In Chapter 7 I present a novel skeleton structure which serves as an approximation to the medial axis of an input surface based on the Voronoi diagram. This approach depends on the barycentric mesh refinement method in Chapter 6.

1.3.2 Varying polygonal density

A typical problem in animation is *adaptive polygon density.* Some typical animation operations, such as *extrusion* or *bunching* may need to adaptively insert polygons into the original shape in order to improve the shapes differential properties for lighting or texturing purposes. However it is not possible with a standard polygonal representation to smoothly and adaptively insert polyhedra in certain regions.

In contrast, an animation manifold is a simplicial representation of space and time, and therefore polygons and vertices can arbitrarily be inserted at any points in time or space using simple simplicial operations. Additionally, using local simplification tools the polygon density of an animation can be reduced. An application of polygonal regularisation is demonstrated in Section 4.5 in the context of deforming animation sequences. This technique draws upon the refinement algorithm of Chapter 6.

1.3.3 Prior technology

Triangle meshes are possibly the most studied surface representation in computer graphics, as well as the most used boundary representation in industrial application. Countless applications exist, ranging from surface simplification, remeshing and fairing to subdivision, deformation and parameterisation. Many, if not most methods which are applicable to triangle meshes in $\mathbb{R}^3$ are also applicable to tetrahedral meshes in $\mathbb{R}^4$, although it is reasonable to assume that not all of these are useful.

In Chapter 2 I present two methods for rendering animation manifolds based on more traditional surface based techniques. In Section 2.5 I generalise raytracing and spacial partitioning to the higher dimensional context, while in Section 2.6 I adapt real–time rendering techniques for isosurface extraction from tetrahedral meshes to the animation manifold domain. In Chapter 4 I develop a Laplacian mesh editing system for animation manifolds in $\mathbb{R}^4$, and apply the approximating subdivision method of Schaefer et al. [2004] to smooth the results in Figure 4.6.

Chapter 2

Building and rendering an animation manifold

An animation volume is a geometric structure which naturally encodes an animation sequence within a certain time interval. In this chapter, I present some basic principles of constructing and rendering animation manifolds.

2.1 Animation manifold definition

An animation manifold is an manifold consisting of n-simplices embedded in $\mathbb{R}^{n+1}$, with one of the spatial dimensions representing time. In this thesis, n is typically 2, indicating that the animation is a two dimensional contour sequence, or 3, where each frame of the animation is a surface[1].

An animation manifold is typically represented by structure $\mathcal{A} = \{P, F\}$, consisting of the set of points P in $\mathbb{R}^{n+1}$ and a homogeneous set of n-simplices F indexing P.

In addition a animation manifold has the following properties:

- The number of connected components can be easily deduced.
- An extracted contour iso-surface is *always orientable*.
- An iso-surface may have a *boundary* only if $\mathcal{A}$ has one.
- Self-intersection may only occur if there is self-intersection within $\mathcal{A}$.

2.2 Building an animation manifold

The problem of connecting 2D iso-contours into an n-manifold in $\mathbb{R}^3$ is well studied [Bajaj et al., 1996, Barequet et al., 2003] — these methods will be discussed in Chapter 8. The problem of connecting 3D iso-surfaces is more challenging, made simple when there are exact vertex correspondences. This is the case with keyframe animation.

[1]I have not considered $n > 3$, but higher dimensionality would represent an additional transformation axis, allowing for transitions between animating shapes. This is an area for future work.

Given m triangle meshes $\mathcal{M}_i = \{P_i, F_i\}$, $i = 1 \ldots m$, with $|P_i| = |P_j| = k, |F_i| = |F_j| = l$ for all $i, j \in m$. Define $\mathbf{P} = \cup_i P_i$ and $\mathbf{F} = \cup_i F_i$. The operation **oneRing**(p) returns the indices of the facets adjacent to a vertex p.

```
int $numVerts = k(m-1)$
int $numPrisms = l(m-1)$
int $numTets = 3 \times numPrisms$
vector<bool> $prismComplete[numPrisms]$
Set all $prismComplete$ to $false$
vector<int> $prismVerts[numPrisms]$
Set all $prismVerts$ to $-1$
for ($i = 1$ to $m$) do
    for ($j = 1$ to $numVerts$) do
        $idx = j + i \times numVerts$
        foreach ($f \in$ oneRing($\mathbf{P}[idx]$)) do
            if (not $prismComplete[f]$)
                if ($prismVert[f] < 0$)
                    $prismVert[f] = idx$
                    Create a tet from this triangle connected to $idx + k$
                else
                    Create tets from remaining non-lifted verts
                    $prismComplete[f] = true$
                end if
            end if
        end foreach
    end for
end for
```

Figure 2.1: The algorithm for constructing an animation manifold from consecutive keyframes with point correspondences.

My algorithm for constructing an animation manifold from consecutive keyframes with direct vertex correspondences is given in Figure 2.1. It makes use of combining a "tent-pegging" approach, where a single vertex is lifted to form a tent, followed by a prism filling approach which completes incomplete prisms. This converts a sequence of surface meshes in $\mathbb{R}^3$ into a single surface in $\mathbb{R}^4$.

The resulting higher dimensional 3-manifold $\mathcal{A}$ exactly reproduces keyframes from the input sequence at the point in time at which they were embedded. Transitions between these frames are *linear* as the edges of the tetrahedral mesh are straight. As is clear from the above algorithm, the number of resulting tetrahedra is $3l(m-1)$, where m is the number of keyframes and l is the number of faces in each mesh. Although this algorithm applies only to constructing a 3-manifold from an input triangle mesh, it can easily be generalised to the lower dimensional case. Figure 2.2 shows part of a short animation sequence created from 50 keyframes.

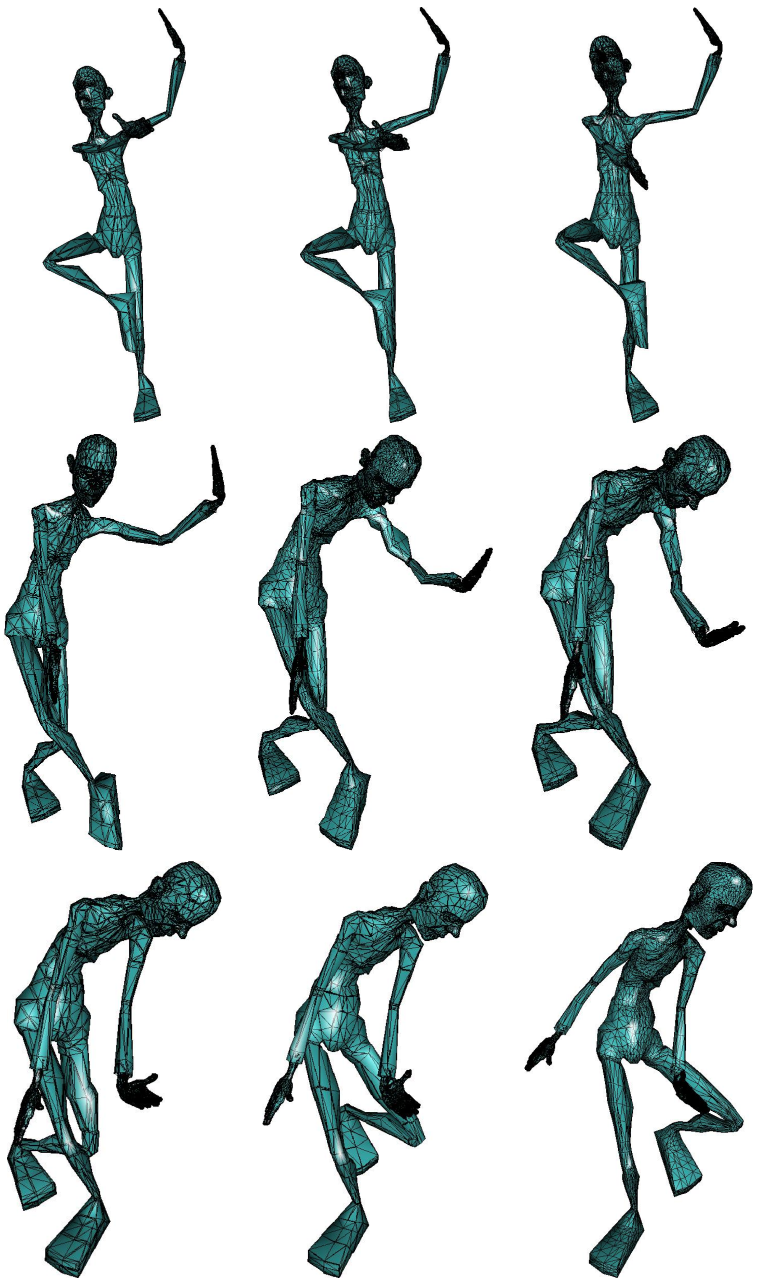


Figure 2.2: Several frames from Emo's dance sequence from the short film *Elephants Dream*. Notice from the overlayed mesh wireframe that the polygon count changes between consecutive frames. The thickening of the outstretched arm in the middle frame is an artifact caused by the linear interpolation between frames.

2.2.1 The boundary

The algorithm given in Figure 2.1 will have an open boundary at the start and the final frame. While this is not an issue when rendering the sequence using the methods described in Section 2.5 and 2.6, methods which apply to triangular meshes, such as those for deformation, subdivision and simplification will require special treatment of the boundary.

The boundary can be trivially *closed* by connecting the first and last frames to a single vertex which lies outside of the animation interval, but since most simplicial methods perform best on regularly sampled geometry, this itself is problematic.

In general, I instead leave the boundary open, and either rely on existing boundary conditions or generate new ones (see Chapter 4) as and where necessary.

2.3 Computing smooth vertex normals

A surface normal (a vector orthogonal to the surface tangent) is typically extracted in $\mathbb{R}^2$ and $\mathbb{R}^3$ using a **cross** or **outer** product. In $\mathbb{R}^3$ this can be written with the commonly used formulation

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \tag{2.1}$$

where $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ are orthogonal unit vectors defining a coordinate system such that a vector $\mathbf{a} = a_x\mathbf{e}_1 + a_y\mathbf{e}_2 + a_z\mathbf{e}_3$.

Unfortunately the outer product does not generalise to all dimensions, and only holds for a *normed division algebra.* An algebra A is a *normed division algebra* if there is a norm $||.||$ such that

$$||xy|| = ||x||||y|| \text{ for all } x, y \text{ in } A,$$

where x, y are members of some vector space. Only real numbers, complex numbers, quaternions and octonions comply with this property.

For this reason, the *wedge product*

$$\bigwedge(\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^n) = \begin{vmatrix} v_1^1 & \cdots & v_{n+1}^1 \\ \vdots & \ddots & \vdots \\ v_1^n & \cdots & v_{n+1}^n \\ \mathbf{e}_1 & \cdots & \mathbf{e}_{n+1} \end{vmatrix} \tag{2.2}$$

is used, where $\mathbf{e}_i, i = 1, \ldots, n+1$ are the orthogonal unit vectors. Note that this formulation is very similar to the cross product except that the orthogonal unit vectors are shifted to the bottom row of the matrix, which affects the orientation of the resulting vector. Similar to the vector cross product, the magnitude of $\bigwedge(\mathbf{v}^1, \ldots, \mathbf{v}^n)$ is the hypervolume enclosed by vectors $\mathbf{v}^i, i = 1, \ldots, n$, and can be used to normalise the resultant normal vector. The wedge product is used in conjunction with the inner product in *Clifford algebras.*

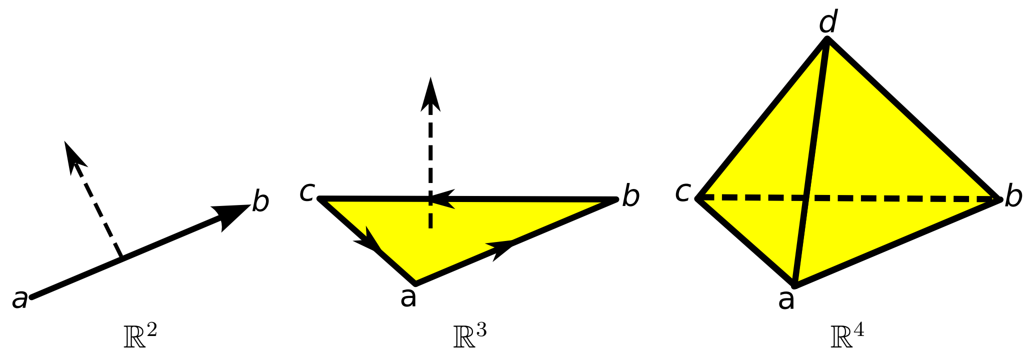


Figure 2.3: Consistent simplex orientation. The dotted arrow indicates the normal vector to the simplex in each case. A normal vector in $\mathbb{R}^4$ cannot be visualised on a 2D sheet of paper.

2.3.1 Smooth vertex normals

Smooth vertex normals are central to real-time shading algorithms such as Phong and Gouraud. Smooth vertex normals which are orthogonal to the tangent at a vertex can be approximated by a weighted sum of the normals of the facets incident on the given vertex:

$$\mathbf{n}_{\text{vertex}} = \frac{\sum_i \alpha^i \mathbf{n}^i_{\text{facet}}}{||\sum_i \alpha^i \mathbf{n}^i_{\text{facet}}||} \tag{2.3}$$

A simple and effective weighting scheme [Akenine-Möller and Haines, 2002] is to use the *area* of facet i as α^i, thus favouring larger incident facets. Note that by the properties of the cross and wedge products, $\alpha^i \mathbf{n}^i_{\text{facet}}$ is simply the unnormalised result of these products. This simple formulation generalises to higher dimensions, giving us a method to compute smooth surface normals at vertices from the surrounding hyperfacets.

2.3.2 Extracting lower dimensional normals

For the smooth shading of isosurfaces extracted from our manifold we need to determine vertex normals in $\mathbb{R}^3$. We can determine the lower dimensional normal by *forgetting* a component of the vector and re-normalising. Formally, we define a normal extraction matrix $\mathbf{M}_i = [\mathbf{e}_1 \ldots \mathbf{e}_{i-1}\, \mathbf{e}_{i+1} \ldots \mathbf{e}_{n+1}]^T$, where $\mathbf{M}_i$ is the $n+1$ by n matrix of basis vectors excluding e_i. The unnormalised surface normal corresponding to dimension i is given by $\mathbf{n}_i = \mathbf{M}_i \mathbf{n}$. Note that the normalisation factor for each "forgotten" component can be precomputed.

2.4 Extracting facets from simplices

All the algorithms which I present depend heavily on the consistent ordering of simplices. I use a right hand ordering convention. This is shown in Figure 2.3. A simple heuristic for this orientation is that to increase the dimension of a simplex the new vertex is always inserted in the positive direction of the normal to the old simplex in the original dimension.

Given a simplex oriented with our formalism above, we can then define an operator which extracts outwardly facing facets. This operator behaves similar, in an abstract way, to a matrix *determinant*. The i_{th} $(n-1)$-simplex facet of an input n-simplex is extracted by removing the i_{th} component, and flipping the first two components if i is *even*. For example, simplex $[A, B, C, D]$ becomes $[B, C, D]$, $[C, A, D]$, $[A, B, D]$ and $[B, A, C]$.

2.5 Ray tracing Animation Surfaces

Glassner [1988] first introduced the idea of ray tracing higher dimensional primitives. An animation manifold is a manifold typically embedded in $\mathbb{R}^4$, rather than the simply constructed boundary shapes used by Glassner [1988]. In spite of this distinction, the approach used here remains largely the same.

Raytracing is a standard approach for creating an image of a virtual scene by tracing in reverse the path a ray of light would follow to a virtual camera lens. This approach allows complex lighting effects to be used in order to improve visual realism.

Central to any ray tracing algorithm is the calculation of intersection of rays and surfaces, the single largest bottleneck to any ray tracing implementation. A simple ray tracing algorithm is given in Figure 2.4, but consists of a very large number of ray-surface intersections per ray.

This algorithm can be easily extended to ray tracing manifolds in $\mathbb{R}^4$. The ray simply becomes a 4-vector by including a time component. Consecutive frames of the sequence are rendered by incrementing the time component. Mixed dimensional complexes are also supported by using the same ray vector and ignoring the time component of the ray. This provides a useful facility for creating static environments for the animation sequence.

A standard method for improving ray tracing performance is to define a spatial hierarchy [Glassner, 1984]. Fortunately these methods generalise easily to higher dimensions. I use a general dimensional spatial partitioning based on the quadtree structure, called a hypertree. Simplices of the animation manifold are assigned to their incident cells as a preprocess. Each 4D ray is intersected with these cells — full intersection tests are only necessary between the ray and simplices in the intersected cell. This approach dramatically reduces the number of intersection tests.

While this structure generalises to any dimension, there are considerable memory limitations. Memory usage in the worst case is of the order of l^{2^n} where l is the number of levels in the tree, and n is the dimension. For this reason, a maximum tree depth must be enforced. Alternatively an out of core solution is also possible. A spatial partitioning algorithm is essential to an efficient implementation of any higher dimensional ray tracer. An example of a rendered animation sequence is shown in Figure 2.5.

2.6 Real-time rendering of Animation Surfaces

The rendering of Animation Surfaces is equivalent to methods to extract isosurfaces from tetrahedral volumes, where the isovalue become the static axis. In most circumstances

```
for (each pixel)
    Generate $ray$ from eyepoint passing through this pixel
    $near\_t = \infty$
    $near\_object = \emptyset$
    foreach ($object$ in $scene$)
        $t$ = intersection($ray$, $object$)
        if ($t < near\_t$)
            $near\_t = t$
            $near\_object = object$
        end if
    end foreach
    if ($near\_object = \emptyset$)
        Fill this pixel with background colour
    else
        Shoot a ray to each light source to check if in shadow
        if ($object$ is reflective)
            Generate reflection ray, recurse
        if ($object$ is transparent)
            Generate refraction ray, recurse
        Use $near\_object$ and $near\_t$ to compute colour
    end else
end for
```

Figure 2.4: A simple ray tracing algorithm. A *ray* is intersected with each *object* in the *scene*. The resulting colour is accumulated for each pixel.

this axis is t, although for visualisation other axes may be made the static axis.

Figure 2.6 gives all cases which arise from isosurface extraction. The vertices are ordered such that inserting them into a triangle strip primitive will yield a counter-clockwise ordered triangulation.

This algorithm can be implemented directly in graphics hardware with little difficulty, as rendering each cell is easily parallelised (see Reck et al. [2004]). Note that by using an alternative component for the isosurface code (for example, y instead of t) it is possible to generate a visualisation of the objects sweep through time. We show a simple example of this in Figure 2.7. This gives an animator a powerful tool by which the results of the animation may be visualised.

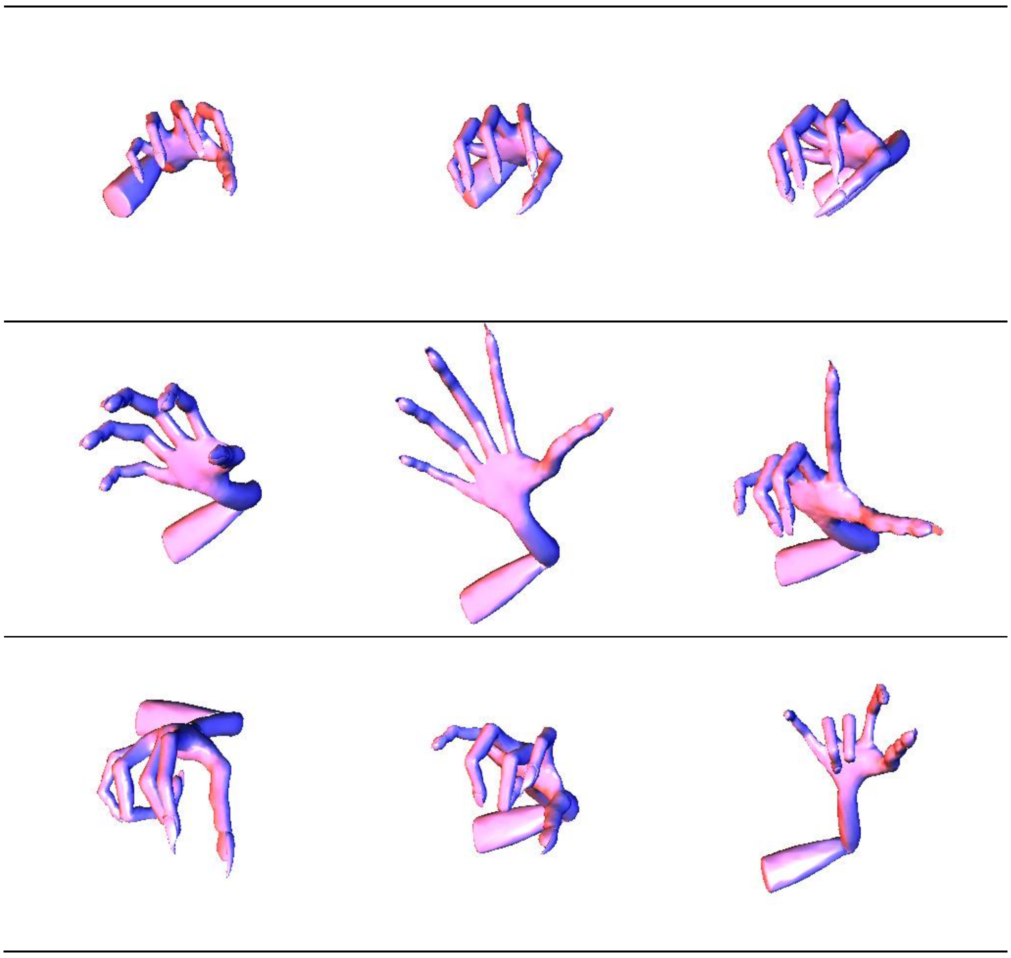


Figure 2.5: A raytraced sequence of a hand. The hand animation was converted to an animation manifold from a sequence of models, each extracted at regular time steps from from an animation sequence. The animation sequence was acquired from the BlenderNation model repository (`http://www.blendernation.com`). The model was available at time of submission.

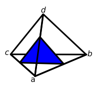
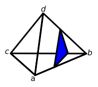
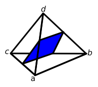
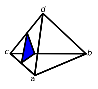
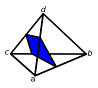
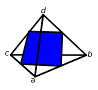
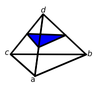
abcd	**split edges**	**abcd**	**split edges**	
$- - - -$	———	$+ + + +$	———	
$- - - +$	$\overline{ac}$, $\overline{ab}$, $\overline{ad}$, –	$+ + + -$	$\overline{ab}$, $\overline{ac}$, $\overline{ad}$, –	
$- - + -$	$\overline{ab}$, $\overline{bc}$, $\overline{bd}$, –	$+ + - +$	$\overline{bc}$, $\overline{ab}$, $\overline{bd}$, –	
$- - + +$	$\overline{ad}$, $\overline{ac}$, $\overline{bd}$, $\overline{bc}$	$+ + - -$	$\overline{ac}$, $\overline{ad}$, $\overline{bc}$, $\overline{bd}$	
$- + - -$	$\overline{bc}$, $\overline{ac}$, $\overline{cd}$, –	$+ - + +$	$\overline{ac}$, $\overline{bc}$, $\overline{cd}$, –	
$- + - +$	$\overline{ab}$, $\overline{ad}$, $\overline{bc}$, $\overline{cd}$	$+ - + -$	$\overline{ad}$, $\overline{ab}$, $\overline{cd}$, $\overline{bc}$	
$- + + -$	$\overline{ab}$, $\overline{ac}$, $\overline{bd}$, $\overline{cd}$	$+ - - +$	$\overline{ac}$, $\overline{ab}$, $\overline{cd}$, $\overline{bd}$	
$- + + +$	$\overline{bd}$, $\overline{ad}$, $\overline{cd}$, –	$+ - - -$	$\overline{bd}$, $\overline{cd}$, $\overline{ad}$, –	

Figure 2.6: A table of all possible isosurface cases for a single tetrahedron. Each vertex is evaluated to determine if it is greater than (+) or less than (−) the given isovalue. This table indicates which edges need to be split for each case and in what order.

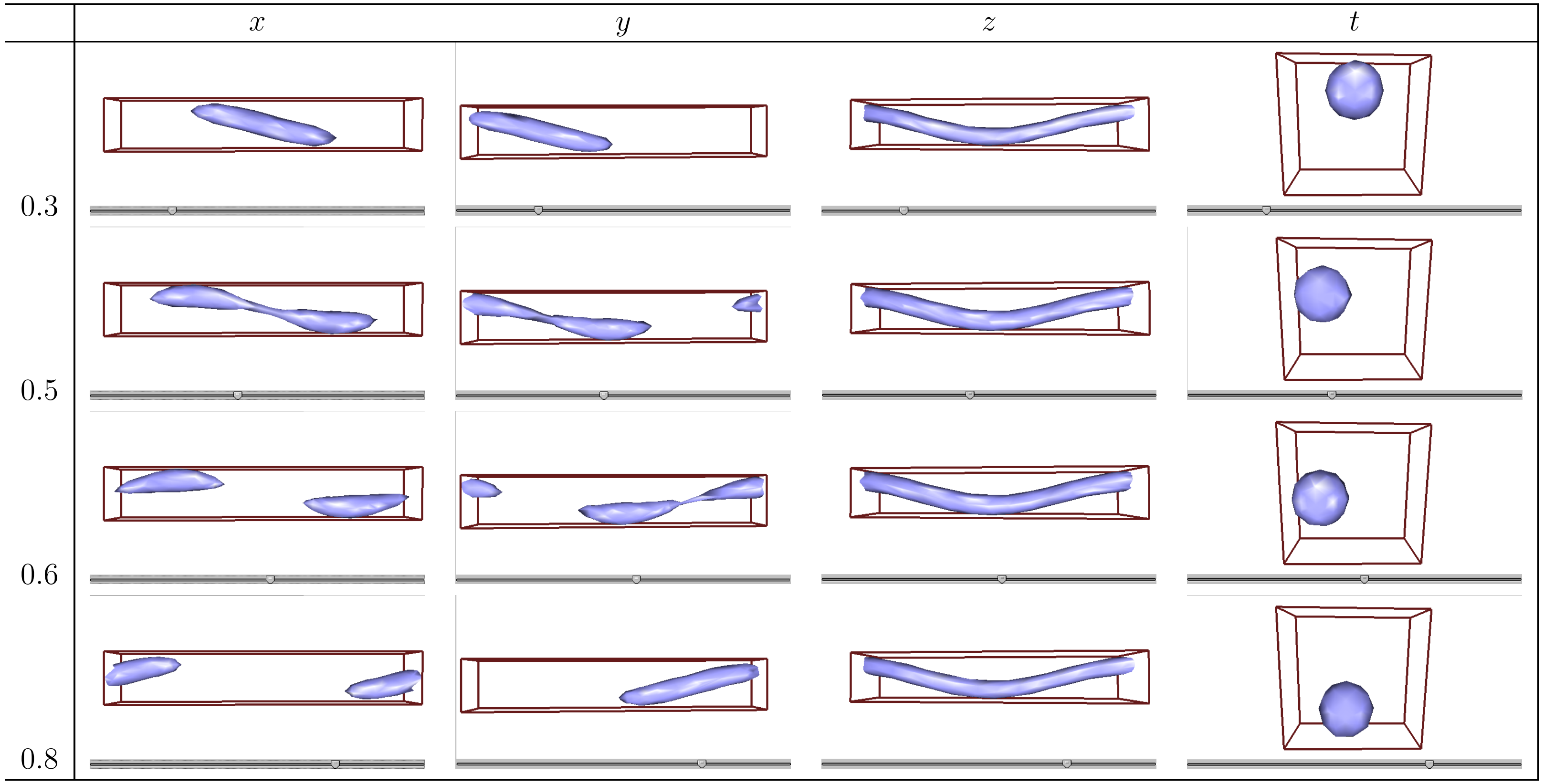


Figure 2.7: A ball revolves in the x and y plane over time t. All four dimensions are visualised by holding a component constant. The horizontal axis represents the component which is constant, while the vertical axis is the value of that component ($0 < t < 1$). Note that since z is not altered in the sequence, what is rendered is a tube with a varying diameter. The right column (t) corresponds with the actual animation sequence. In all remaining columns, t increases from left to right.

Chapter 3

Background

The concept of unifying time and space into a single construct dates back to late 19^{th} century authors and philosophers. It was first defined with reference to theoretical physics by Herman Minkowski [Lorentz et al., 1952], and has since become a fundamental building block in special and general relativity, cosmology and speculative theories such as string and M-theory.

Surprisingly, few applications of space–time to Computer Graphics and Animation exist, given that it is conceptually a natural representation for an animation sequence. In particular, only three methods surveyed use a tightly coupled space–time representation, and no existing method of representing animation sequences makes use of a simplicial representation. In this Chapter, I will outline prior work in the application of space–time for computer graphics, with particular reference to space–time representations used in the literature.

3.1 Applications of space–time

While Minkowski space–time has enormously influenced models of our universe, any sciences which capture time series data have benefited by unifying the concepts of space and time. For example, Magnetic Resonance Imaging (MRI) data is 4-dimensional, and a significant body of medical imaging literature is devoted to the segmentation and automatic identification of features, such as cardiac motion [Shen et al., 2005] and morphological changes in the brain [Shen and Davatzikos, 2004].

Computer animation can be thought of as a basic application of space–time. Early cartoons were rendered by using keyframes at certain time instances, and the frames in between filled with some form of interpolation (called tweening). Simulating realistic animation of characters and physical objects with tweening remains a challenging problem even today, with various different forms of interpolation and interface options available [Foley et al., 1997]. Traditionally spline paths and complex user interfaces are used to allow the user to plot a smooth path for each vertex in the animation sequence.

Automatic constraints based path planning is of interest to both the computer animation industry and robotics communities, and improving the performance of these systems has been an active area of research. The revolutionary Luxo junior animation was based on the work of Witkin and Kass [1988], in which the authors introduce a method to define an animation automatically by specifying constraints relating not only to the skeletal and physical characteristics of the animating body and desired result, but also to energy used and physical resources which are available. The spacetime animation is the result of the solution of a non-linear optimisation problem. Since this paper there have been a raft of performance improvements, most recently by Guenter [2007].

Raytracing of animation sequences is also an obvious field which benefits from a unified spacetime. Glassner [1988] improved the spatial partitioning algorithms for raytracing by generalising the octree [Glassner, 1984] and "slab" [Kay and Kajiya, 1986] partitioning algorithms to higher dimensions. By building these data structures based on the entire animation sequence and embedding these into a single higher dimensional data structure, ray-tracing can be dramatically improved since the partitioning need not be recomputed at every step. In Section 2.5 I use a similar approach to partition space in the form of the *hypertree*, but the animation manifold is itself truly a $4D$ structure, and so differs from the primitives used in Glassner's scenes.

Finkelstein et al. [1996] introduced a space-time representation of video sequences which each frame at some temporal resolution as a quadtree into a binary tree. Each leaf node represents the highest temporal resolution of some region in the video, and while intermediate frames contain interpolated resolutions. This method allows the authors to simulate effects such as motion blur, or to render the video at differing spatial or temporal resolutions. Klein et al. [2002] embed each frame of a video sequence into a space-time volume and render the result using a time-varying kd-tree [Bentley, 1975]. This representation allows the authors to explore a range of artistic yet contextually unaware non-photorealistic rendering techniques.

Additionally time-based simulations have been employed in physical simulations, such as with deformable objects [Debunne et al., 2001] and in cloth, water, smoke and other phenomena [Nealen et al., 2005]. These problems will typically employ time–series integration to define smooth transitions between frames, using diffusion, deformation or one of a number of physical interactions. For a survey of these techniques, the reader is referred to the report of Nealen et al. [2005].

3.2 Space–time representations

In almost all space-time representations in Computer Graphics space and time are *loosely coupled* meaning that keyframes and temporal data are contained within different data structures. This allows the keyframes to easily be extracted.

Aubert and Bechmann [1997] create simple space–time objects by extruding or evolving surfaces in $\mathbb{R}^3$ into the time dimension. Using this methodology, interesting topological alterations can be achieved using standard deformation tools.

The shape representation used is defined by Brandel et al. [1998] for use with their STIGMA system, which utilises the *generalised map* or *G-map* structure of Leinhardt

[1989]. The modeller connects shapes by using special connections across the time dimension called *darts* which may split or converge in order to permit changes in polygon density or topology between frames. Smoothing paths between elements in the G-map requires special behaviour to be encoded into the individual darts. Additionally, the STIGMA package provides a novel user interface by which space–time deformations can be specified. Construction of these structures is difficult and time consuming, and the results of their approach are therefore limited to transitions between relatively simple platonic shapes.

In comparison, an animation manifold is a simplicial n-complex in $\mathbb{R}^{n+1}$, allowing the use of an interpolating or approximating subdivision to produce C^1 paths in the animation sequence — a comparatively simple operation. Deformation is also made easier as the animation manifold is a homogeneous structure. Most importantly, constructing the animation manifold between arbitrary shapes is *automatic*, as shown in Chapter 8.

However, the G-map has the advantage that all edges spanning time are "straight", so transitions may be linear or smooth. In contrast an Animation Manifold is a simplicial complex which may have edges running obliquely in time. While it is difficult to visualise the result on surfaces in $\mathbb{R}^4$, oblique edges in time will typically have the same rendering, deformation and subdivision artifacts which arise in triangular meshes in $\mathbb{R}^3$ which are representing a parameterised surface, such as a quadrilateral grid. These are similar to the artifacts encountered in Carr et al. [2006].

Turk and O'Brien [1999] generalise the thin plate energy minimisation problem to higher dimensions as an N-dimensional scattered data interpolation problem. Given a set of constraints which take the form of points and normals on some surface, an energy minimising implicit representation interpolating these points is constructed by solving a linear system of equations constructed from local radial basis functions.

An application of this generalised surface construction algorithm is to the construction of morph sequences by embedding planar cross-sections at points in time, sampling vertices and normals from these surfaces, and constructing and solving the linear system.

This approach has several unique properties:

- The input for the algorithm need not be in the form of planar cross-sections in t. The authors demonstrate a method of surface design using cross-sections in t and y to produce better surfaces.
- Additional dimensions can be added to the system based on other input shapes. These *influence shapes* can be used to create unique blend axes. This is similar to the approach of Fausett et al. [2000].
- Extracted cross sections are smooth and non-linear, as is the transition between the shapes.

The method yields visually pleasing results with an elegant solution. Additionally it has the property that it can deal with arbitrary input topologies. However, it has several shortcomings:

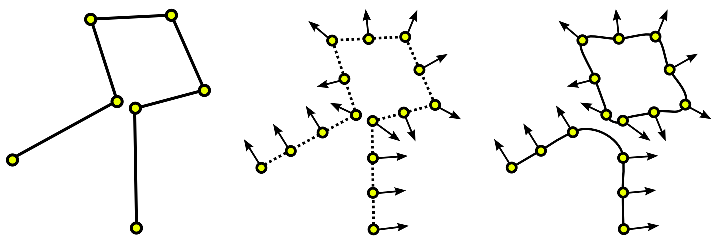


Figure 3.1: An example of incorrect surface generation from scattered data interpolation. A connected surface (a) is given as input to a variational interpolation method, such as that of Turk and O'Brien [1999]. In (b), the surface is sampled with normals. Note that a local feature size is smaller than the sampling density. The result in (c) incorrectly separates the shape into two.

- Like all methods of scattered data interpolation, if the sampling density of the manifold does not take into account the minimum feature size, the surface may separate into pieces (see Figure 3.1). Our method and the methods of Klimmek et al. [2007] and Brandel et al. [1998] will exactly reproduce the connectivity of the input geometry.
- The resulting manifold is not a simplicial complex, and cannot be deformed or edited by standard simplicial techniques.

Fausett et al. [2000] introduce a functional language which can be used to specify a shape representation. A shape transformation is achieved by defining transitions between these functional representations. An array of these transitions can be produced, allowing for multi-dimensional transitions between shapes in a manner similar to Turk and O'Brien [1999]. Due to the restrictive nature of the functional language as shape specification, only relatively simple shapes and shape approximations are supported.

Shamir and Pascucci [2001] and Kircher and Garland [2005] derive multiresolution space-time representations of animating meshes. Shamir and Pascucci [2001] builds a directed acyclic graph from multiple resolutions of keyframes based on how the mesh changes between keyframes. By separating temporal deformations into differing frequency bands, deformations can be transfered onto different levels of detail in the hierarchy. Kircher and Garland [2005] define a multiresolution representation of a deforming sequence which takes into account the deformations occurring over time. Later in Kircher and Garland [2006] the authors exploit this structure and a signal processing approach to allow detail to be embedded into an existing animation sequence.

3.3 Summary

A unified framework for representing space and time has applications in any field of science involving time series data, including seismology, medical imaging and theoretical physics. Of the methods surveyed, I am only aware of three other systems in computer graphics which tightly couple time and space: the variational interpolation method of Turk and O'Brien [1999], the generalised map representation for space–time deformation of Brandel et al. [1998], and the video cube methodology of Klein et al. [2002]. No space-time simplicial representations have been introduced.

The animation manifold is unique in that it is a tightly coupled *simplicial complex* and a *boundary manifold*. A large body of work exists for simplification, signal processing, editing, rendering and subdividing these structures. This makes animation manifolds sufficiently flexible to be suitable to a wide variety of modelling tasks.

Unfortunately animation manifolds also inherit some of the problems associated with standard surface representations. Artifacts arise during subdivision across feature lines (see Figure 10.1). Additionally the tight coupling of time and space make it difficult (but not impossible) to modify the spatial features of a single keyframe without influencing consecutive frames.

Chapter 4

Deforming animation manifolds

Topological alteration is difficult to model in an animation sequence. Most animation packages make use of surface representations, such as NURBS, triangle meshes or subdivision surfaces. Topological alteration of surfaces, however, can only be represented by one of two methods:

- The most common approach uses metaballs [Wyvill et al., 1986], which require the object to be converted into an approximate "blobby" shape.
- Constructive solid geometry is less commonly used, where boolean operators are used to define relationships between different geometric primitives.

In either case, the surface must be changed from the initial shape representation into another form for the topological alteration to occur. This adds approximation error and increases the complexity of applying the operation.

In the application STIGMA [Brandel et al., 1998], the authors showed that by editing a 4D representation it is possible to model complex topological modifications, such as modifying topological genus or the number of connected components. In Figure 4.1 a splitting operation can be modelled using a deformation operation on the higher dimensional structure.

Recall that the animation manifold is a n-manifold embedded in $\mathbb{R}^{n+1}$. Free form deformation is a powerful tool for building and modifying shapes using a simplified editing interface with intuitive behaviour. Laplacian mesh editing is a free form deformation tool which is based on discrete signal processing. As I will show, the Laplacian approach generalises easily to higher dimensional manifolds such as the animation manifold. I will use this technique to deform animation manifolds. The goal of this chapter is to reconcile the simplicity of free form deformation, with the power of 4D modeling, allowing animators to represent animation and topological alteration in the same consistent medium.

However, there are several limitations to standard Laplacian editing which need to be addressed.

- Laplacian mesh editing typically encodes features in an anisotropic manner, causing surface detail to be distorted during deformation. I show that the dual Laplacian

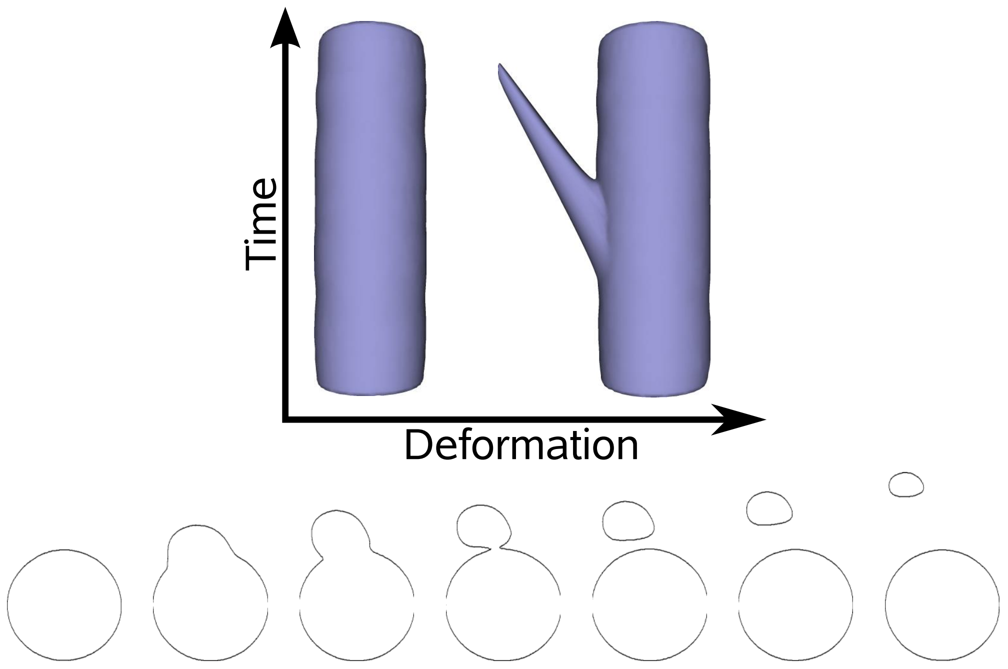


Figure 4.1: Deformation can be used to modify the topology of an animating shape. A branch is extended from the tube, which represents the animation sequence in 2D of a small circle splitting off from the larger circle.

method of Au et al. [2006] best preserves surface details in an isotropic manner, and generalises to higher dimensions.

- Laplacian methods do not deal adequately with surfaces that have a boundary. Animation manifolds will typically have a boundary at the start and end keyframes. In Section 4.4 I present both primal and dual boundary conditions for Laplacian mesh editing.
- As Laplacian editing is a discretised signal processing approach, the shape and quality of the triangulation will affect the predictability of the editing behaviour. However, a deformation in the form of an extrusion will cause triangles to elongate. I present a simple mesh regularisation step which is applied after deformation.

4.1 Background

Deformation spans a large range of methodologies, from the manipulation of handles or bounding shapes, to sketch-based interfaces based on local geometry. Recently there has been significant interest in deformation methods based on local differential surface properties, and for this reason we focus our work on these techniques. These can be generally divided into multiresolution approaches [Zorin et al., 1997, Kobbelt et al., 1998, Guskov et al., 1999, Botsch and Kobbelt, 2004] and Laplacian based techniques [Taubin, 1995, Sorkine et al., 2004, Lipman et al., 2005, Sheffer and Krayevoy, 2004, Au et al., 2006].

Multiresolution approaches decompose the input surface into varying levels of resolution, or *frequency bands.* Deformation is performed by editing one (or more) of these frequency bands. Resolving the multiresolution hierarchy reconstructs the higher frequency detail features. However, a regular mesh decomposition often requires remeshing the input surface such that multiple inter-dependent levels of resolution can be deduced.

Laplacian editing has recently (since 2002) received considerable attention due to its ability to simulate local feature and global shape editing in a more intuitive manner. In a typical Laplacian system, a user defines surface geometry to be marked as *anchors* which are fixed, and *handles*, which can be manipulated by the user. Unmarked geometry is then allowed to freely deform as the handle is manipulated. It is due to this flexibility that I use a Laplacian editing framework for performing deformations on animation manifolds.

Special mention should be made of the methods of Angelidis et al. [2004] and von Funck et al. [2006] as these approaches *implicitly* preserve the enclosed surface volume. The vector field deformation approach of von Funck et al. [2006] in particular preserves local features. Implementing these approaches with animation manifolds is an area for future work.

4.2 Laplacian surface editing

In this section I derive the basic Laplacian deformation operator in $\mathbb{R}^n$. For a full overview of the various methods applicable to surface editing, see the literature survey of Sorkine [2005].

Laplacian mesh editing is effectively an application of *signal processing* on surfaces meshes, and was first introduced by Taubin [1995]. To derive the Laplacian matrix L we consider the differential properties of the geometry. A simple method of measuring the differential properties is to consider the basic hat function about each of the m vertices of the input manifold $\mathcal{M}$:

$$\boldsymbol{\delta}_i = v_i - \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} v_j$$

where $\mathcal{N}_i$ are the indices of vertices within some neighbourhood of i (typically the 1-ring). The vector $\boldsymbol{\delta}_i$ is the difference between the vertex and some mean of its neighbourhood, and is an approximation of the discrete mean curvature at vertex v_i. $\boldsymbol{\delta}_i$ encodes a *local feature.*

A more general form of this equation is

$$\boldsymbol{\delta}_i = v_i - \sum_{j \in \mathcal{N}_i} w_j v_j, \tag{4.1}$$

$$\sum_{j \in \mathcal{N}_i} w_j = 1, \; w_j > 0 \text{ for all } j \in \mathcal{N}_i$$

This can be formulated into the linear system $\overline{L}V = \overline{\Delta}$, where $\overline{L}$ is a square $m \times m$ matrix, V is the $n \times m$ matrix containing the vertex positions, and $\overline{\Delta}$ is an $n \times m$

matrix containing as its i^{th} row the vector $\boldsymbol{\delta}_i^T$. Intuitively, this formulation encapsulates coordinates of the input geometry in a *relative* sense.

By itself, $\overline{L}$ is ill-defined, with a rank of $n-1$. Intuitively this is because it does not "fix" the geometry at any absolute position in space. By appending a row indicating one *absolute* vertex position to matrix $\overline{L}$ and $\overline{\Delta}$, for example of the form

$$L = \begin{bmatrix} \overline{L} \\ e_i^T \end{bmatrix}, \text{and}$$

$$\Delta = \begin{bmatrix} \overline{\Delta} \\ v_i \end{bmatrix},$$

where e_i is the i_{th} unit basis vector, the system $LV = \Delta$ is *anchored* to the location of v_i, and has a unique solution. If multiple anchors are specified and shifted, the result is a deformation of the input geometry. The system is overconstrained, and can ordinarily be resolved in a least squares sense by using the pseudo-inverse, $V = (L^T L)^{-1} L^T \Delta$.

This formulation can be used for free form deformation by distinguishing between anchored rows of L. Some retain a fixed position during editing, which are called *anchors*, while others can be transformed (translated, scaled or rotated) by the user using a user interface, which are called *handles*. This is demonstrated in Figure 4.3.

A special property of the matrix L is that it remains unchanged during deformation, and as $L^T L$ is symmetric, sparse and positive definite, $L^T LV = L^T \Delta$ can be factorised with Cholesky factorisation[Gould et al., 2005]. Deformation is then performed by changing the absolute positions of handles in Δ, and solving the pre-factorised system $L^T LV = L^T \Delta$.

Several methods have been used for choosing the weight w_j in Equation 4.1. Taubin [1995] employed an equal weighting $w_j = 1/|\mathcal{N}_i|$ for all $1 \leq j \leq |\mathcal{N}_i|$, but empirical evidence [Meyer et al., 2003] suggests that, on triangle meshes, the cotangent operator of Pinkall and Polthier [1993] yield the best results. The cotangent operator, a generalisation of the Laplace–Beltrami operator, is given by

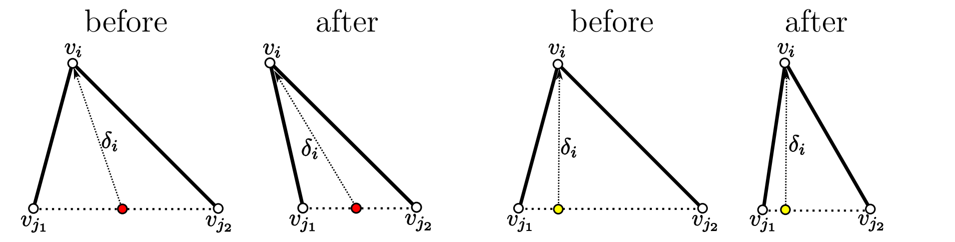
$$w_j = \cot \gamma_j + \cot \beta_j$$

where γ_j and β_j are the angles in triangles about i opposite the edge ij. Desbrun et al. [1999] showed that the cotangent operator implicitly minimises the vector $||v_i - \sum_{j \in \mathcal{N}_i} w_j v_j||$, and therefore minimises the *tangential drift* caused during deformation (see Figure 4.2).

There are two noticeable shortcomings of the basic Laplacian editing framework:

- *Transformation invariance*: Translation and rotations applied to handles of the Laplacian system will cause local features to become distorted, as these features are defined on the global domain. A transformation invariant Laplacian must encode local features in a local frame.

- *Volume preservation*: In many applications, it is desirable that the edited manifold behaves like clay, in that the enclosed volume is preserved. Laplacian editing does not implicitly preserve the enclosed volume.



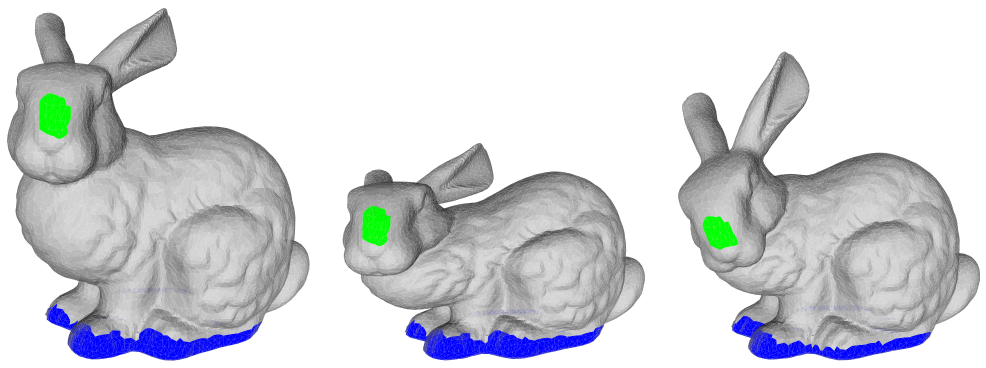
(a) Even weights of Taubin [1995] (b) Cotangent weights of Pinkall and Polthier [1993]

Figure 4.2: Here I demonstrate why cotangent weights better preserve differential properties than even weights. In each case, a deformation is performed which contracts the base v_{j_1} and v_{j_2}. In (a) the vector $\boldsymbol{\delta}_i$ has an implicit tangential component, causing the deformed surface to not only distort the local feature, but also increase the tangential component. In (b), $\boldsymbol{\delta}_i$ is minimal, at a normal to the base shape, and has no tangential component. This figure is adapted from Au et al. [2006].

4.2.1 Transformation invariance

Several approaches have been presented for building a Laplacian with rotational invariance. Lipman et al. [2005] build local coordinate frames which are invariant under rotation by projecting local vertices into a local tangent plane. Sheffer and Krayevoy [2004] define pyramid coordinates from the local coordinate frames, and use an iterative approach to update the vertices giving them rotational invariance.

Both methods suffer from *tangential drift* as the tangent deduced from the local region represents a region of the surface which may not necessarily be entirely convex or concave — due to the mixed curvature behaviour, features can "slide" during deformation, and the iterative solution of Sheffer and Krayevoy [2004] may not converge.



(a) (b) Cotangent weights (c) Dual Laplacian

Figure 4.3: In (a) the base of the bunny is painted as anchors (in blue) and part of the head is painted as handles (green). A similar deformation is applied in both (b) and (c), where the handle points are dragged downwards. In (b), the bunny appears to be squashed as basic cotangent weights are not invariant under transformations. In (c) the bunny still retains it's "bunny-like" attributes, such as its pointy ears. Restoring the features of the bunny in (c) takes ≈ 20 iterations with the method of Au et al. [2006].

The method of Au et al. [2006] is both *transformationally* and *rotationally* invariant, and will be described in greater detail. Like Sheffer and Krayevoy [2004] a local coordinate frame is defined by finding the projection of the input vertex v_i into the plane of its neighbours. To avoid the drifting problem, this is instead built from the graph *dual*, a process defined by Taubin [2001]. The dual graph of a triangle mesh defines a vertex at the centroid of each facet, and as each triangle has three neighbours in the primal mesh, each vertex in the dual mesh has exactly 3 neighbours, except possibly on the boundary. Each vertex of the dual mesh therefore defines a unique plane and normal component.

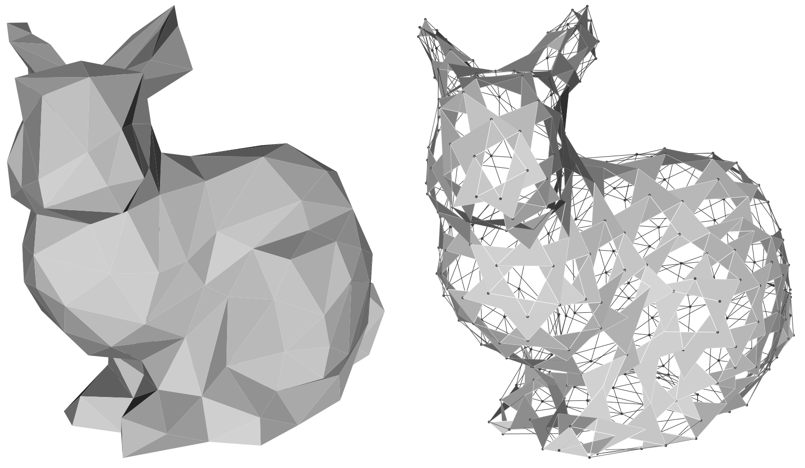


Figure 4.4: In (b) the triangle representing the local tangential frame of each dual vertex of (a) is shown. These triangles are used to define a local rotationally invariant frame for each vertex.

The dual matrix operator D computes face–vertices using a simple weighting scheme. If $f_{i,k}$ is the index of the k^{th} vertex of face i, then D is deduced from $v'_i = \sum_k \frac{1}{n} v_{f_{i,k}}$. The weights w_{ij} and the associated feature vectors $\boldsymbol{\delta}_i$ are deduced by finding the minimum projected distance from v_i to its associated dual facet. Note that there are exactly n vertices in the dual facet, so there is no ambiguity (and hence no tangential component) in $\boldsymbol{\delta}_i$. This gives us the dual Laplacian matrix L', while the actual Laplacian is then given by $L = L'D$.

An iterative procedure is then employed to restore local features after deformation by computing the local normal, adjusting the values of $\boldsymbol{\delta}_i$ accordingly, and resolving the system. This approach is basically a non-linear Gauss-Seidel method, which converges under most geometric conditions. The dual Laplacian is compared with a standard cotangent weight Laplacian in Figure 4.3.

4.2.2 Volume preservation

Zhou et al. [2005] use an explicit approach to achieve volume preservation on deformed meshes. They define an internal volumetric structure, and define two differing Laplacian meshes — the outer mesh is defined using a cotangent weight Laplacian, and an inter-

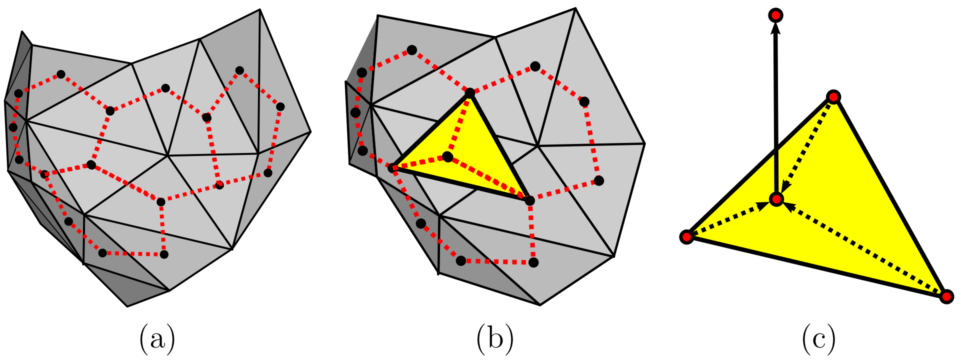


Figure 4.5: The derivation of the dual graph Laplacian. Vertices of the dual graph in (a), shown in red, each have a valence of 3. This property allows us in (b) to define a unique tangential plane (shown in yellow) for each dual vertex v_i. In (c), the dual Laplacian coordinates are defined by the minimal distance of the input vertex to the local tangent plane.

nal mesh Laplacian. The weights for the volumetric Laplacian are formulated with the following *quadratic programming* problem:

$$\min_{w_j} \left(||v_i - \sum_{j \in \mathcal{N}_i} w_j v_j||^2 + \lambda \left(\sum_{j \in \mathcal{N}_i} w_j ||v_i - v_j||^2 \right) \right) \tag{4.2}$$

$$\text{such that} \sum_{j \in \mathcal{N}_i} w_j = 1 \text{ and } w_j > \xi.$$

The volumetric graph Laplacian is derived from two energy terms:

- the first minimises the length of the vector $\boldsymbol{\delta}_i$, effectively minimising the distance from the point predictor to the actual point, and
- the second favours weights inversely proportional to the edge lengths.

ξ and λ are user controlled parameters to alter the shape of the Laplacian. If λ is zero, this is the equivalent to the cotangent weights of Desbrun et al. [1999] in the case of a manifold in $\mathbb{R}^3$. ξ is present to prevent a degenerate solution, and enforces positive, non-zero weights.

Only the first term is a quadratic — if it is dropped we are left with a linear programming problem, quickly solved by setting $w_j = 1/||v_i - v_j||$, the reciprocal edge length.

As it stands, 4.2 is a quadratic programming problem for every vertex prior to solving the system $L'V = \Delta$ in order to deduce the weights.

It can be rewritten in the quadratic programming form

$$\min_{\mathbf{w}} f(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T Q \mathbf{w} + \mathbf{c}^T \mathbf{w}$$

with the each entry of matrix Q given by $[Q]_{ij} = 2v_i^T v_j$ and

$$[\mathbf{c}]_j = \lambda ||v_i - v_j||^2 - 2v_i^T v_j.$$

As Q is positive semi-definite, the problem is convex, and is therefore solvable using a number of off-the-shelf solvers.

4.2.3 Adaptive subdivision

Adaptive subdivision is a method to smooth a surface in isolated regions based on some criteria. Typically the dihedral angle between adjacent faces is used as a measure of the sharpness of a feature, but features can also be ranked according to their importance to the viewer. A generalised approach to adaptive subdivision was made available by Sovakar and Kobbelt [2004].

Subdivision has also been applied to deformation by Gain and Dodgson [1999] as a means to detect surface self intersection. A similar approach was also used by Angelidis et al. [2004]. An important property of the approach of Gain and Dodgson [1999] is that the *deformed surface itself is not subdivided*, rather the original mesh is subdivided, and the deformation applied to it. This approach reduces degeneracies which can arise when the deformed surface is used as a base mesh for subdivision.

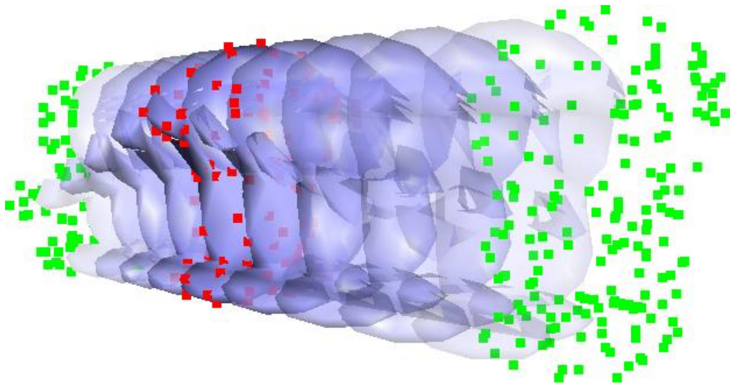
4.3 Deforming animation manifolds with Laplacian editing

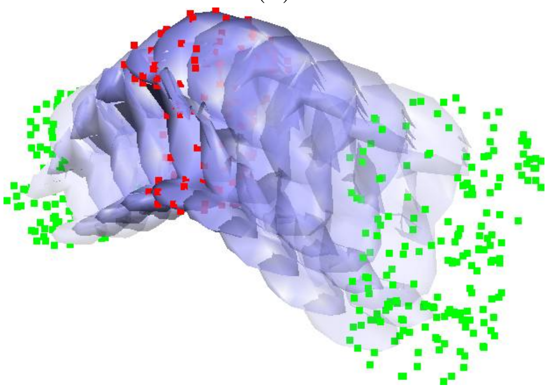
A simple example of editing an animation manifold is given in Figure 4.6. An animation manifold is constructed by the algorithm in Figure 2.1 by timeshifting several copies of the same model. Green points indicate anchors, and red points indicate handles. In (a), anchors were shifted to raise both arms of the character in the middle frame. In (b) the handles are lifted (leaving the constraints in place) causing a smooth deformation of the animation sequence in the z direction. Multiple frames are shown with transparency for visualisation purposes (a process I call “ghosting”). The volume was subdivided once using the method of Schaefer et al. [2004] to smooth out the resulting geometry and animation path. Note that deformations on the animation manifold are implicitly propagated in time.

Laplacian editing of the animation manifold may appear straightforward, but several unique characteristics differentiate it from the lower dimensional equivalent:

- It typically has an *open boundary.* When it is constructed from connecting keyframes, there is no data succeeding the first keyframe, or proceeding the final keyframe. The manifold could be closed by attaching the first and last keyframes to a single vertex, but I have found that this impacts negatively on both the editing behaviour and the subdivision surface.

- The concept of *regularity* of a triangulation of a manifold in $\mathbb{R}^4$ is more difficult to define than in the lower dimensional case. In $\mathbb{R}^3$ a triangulation can fit together as equilateral triangles in a hexagonal grid, but no such convenient shape exists in $\mathbb{R}^4$. Semi-regular octahedral configurations have been introduced [Schaefer et al., 2004] but these do not have even edge lengths. For this reason, the equal weights of Taubin [1995] will cause unwanted distortion on tetrahedral grids.


(a)


(b)

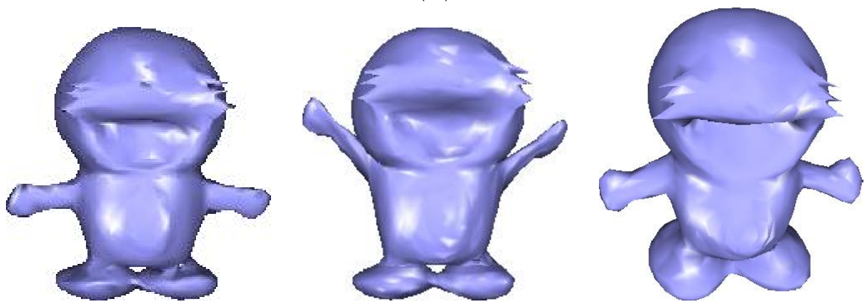


Figure 4.6: A simple animation sequence of a character jumping, lifting and lowering its arms. These images were generated using the equal weights $w_j = 1/|\mathcal{N}_i|$ and as a result some distortion can be expected due to the natural irregularity of a tetrahedral mesh.

For the above reasons, only two existing Laplacian deformation techniques could potentially yield good results when editing of animation surfaces. The Volumetric Graph Laplacian (VGL) defined in Zhou et al. [2005], which solves for local weights as a quadratic optimisation problem, should yield similar near optimal Laplacian editing results as the cotangent weights of Pinkall and Polthier [1993]. Additionally the VGL does not need any special constructions around the manifold boundary.

The dual graph Laplacian method of Au et al. [2006] is attractive in that it generalises easily to higher dimensions, and it naturally preserves features due to its implicit rotationally and translationally invariant construction. An unresolved issue, however, is how the dual graph can be constructed at the manifold boundary.

4.4 Boundary conditions

Other authors have not yet defined rules for deforming triangle meshes with a boundary. In fact, most methods of Laplacian editing will fail on meshes with a boundary due to the dependence on the one-ring neighbourhood to deduce the weights w_j of each vertex in Equation 4.1. Some examples of these deficiencies are:

- There is no rule to define cotangent weights across the boundary. To deduce w_j, $j \in \mathcal{N}_i$ there need to be facets on either side of the edge $v_i v_j$.
- There is no rule to define the matrix dual of Au et al. [2006] about the boundary. Each triangular face requires three neighbours in order to define the neighbourhood plane.
- The even weights of $w_j = 1/n$ of Taubin [1995] can be used across mesh boundaries, but these will be highly susceptible to tangential drift, shown in Figure 4.2 even in the perfectly regular case. A simple example of this is given in Figure 4.7.

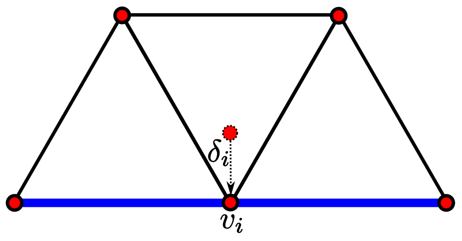


Figure 4.7: In this figure, the blue edges indicate a boundary of an otherwise perfectly regular and flat mesh region. Using an even weighting method to define $\boldsymbol{\delta}_i$ in Equation 4.1 will result in it lying within the plane of the mesh region. Therefore $\boldsymbol{\delta}_i$ *only* has a tangential component and guarantees tangential drift.

A boundary can be strictly enforced by fixing boundary vertices as *anchors*, forcing (in a least squares sense) the boundary to be maintained. However, this will likely be too restrictive to be practical. I will derive boundary conditions for primal and dual Laplacian editing in the following sections.

4.4.1 Primal Laplacian boundary conditions

To define a solution to the boundary editing behaviour, we must first ask what the desired result of editing the boundary is. Prior literature suggests that for best results, $\boldsymbol{\delta}_j$ should encode feature information in a direction that is at a normal to the local surface, i.e. without any unwanted *tangential information*. This is achieved by *minimising* the value of $\boldsymbol{\delta}_j$.

For this purpose, I adapt the first term of the volumetric graph Laplacian of Zhou et al. [2005], finding the weights w_j by solving the quadratic programming problem

$$\min_{w_j} \left(||v_i - \sum_{j \in \mathcal{N}_i} w_j v_j||^2 \right) \tag{4.3}$$
$$\text{such that} \sum_{j \in \mathcal{N}_i} w_j = 1 \text{ and } w_j > \xi.$$

For well-formed meshes we can assume that the boundary is relatively small, so this expensive operation can be used as an alternative to the cotangent weighting at boundary regions of the mesh.

This approach could also be applied to meshes which are non-manifold, and may also be developed into a method for deforming an arbitrary *polygonal soup* assuming that a means to identify some local connectivity is available.

4.4.2 Dual Laplacian boundary conditions

Taubin [2001] and Au et al. [2006] construct the dual mesh only from closed manifolds. Dealing with open manifolds requires a significant modification to the construction of the local Laplacian. In this section I propose a solution to this problem by creating a *ghost* face across the boundary edge of a face f by rotating f about the midpoint of the boundary edge by π. The dual vertex of this ghost face is incorporated into the Laplacian system.

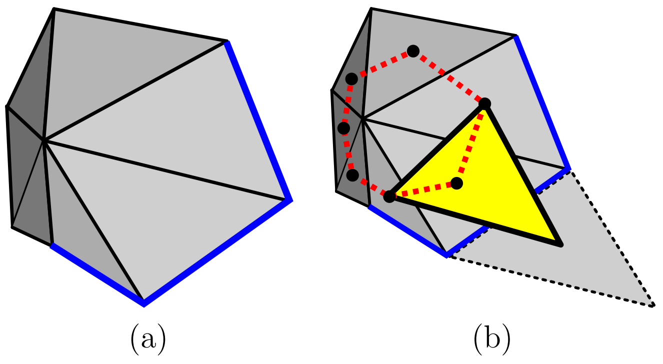


Figure 4.8: Boundary conditions for the derivation of the dual mesh Laplacian. In (a) a facet lies on a boundary (marked in blue). In (b), a dual tangential plane (in yellow) is derived by creating a ghost dual vertex from the ghost of the face on the boundary.

Given an $(n-1)$-simplex of the input manifold $\mathcal{M}$ with at least one sub-simplex on the boundary of $\mathcal{M}$, its dual is vertex v'_i. The ghost dual vertex is defined by $\bar{v}'_i = \sum_{k=1}^{n-1} \frac{2}{n} v_k - \frac{1}{n} v_l$ where v_l is the vertex in the simplex opposite the boundary sub-simplex[1].

Note that appending the new ghost vertices to the dual matrix operator D will cause the rank of the modified Laplacian $L'D$ to be decreased by one for each of dual vertices. This is significant, as it is possible for a system to be defined with fewer anchor and handle vertices than there are constraints. This will result in an *underconstrained* system, which is undesirable. This situation can be dealt with in a number of ways:

- The ghost vertices can be made anchors. This will change the editing behaviour significantly as the boundaries will remain fixed. However, the matrix will be overconstrained.
- The vertices surrounding existing anchors or handles can be *grown* to neighbouring vertices until sufficient constraints exist for L to be overconstrained. This will slightly modify the editing behaviour.
- The user may simply be informed that insufficient anchors and handles have been specified.

Note that all of these solutions should be thought of as user preferences rather than rigid mathematical solutions.

4.5 Adaptive refinement of deformed geometry

Laplacian mesh editing, like all simplicial deformation techniques, allows simplices to be extruded as they are deformed. The Laplacian operator, however, is particularly susceptible to poor mesh sampling, as it is constructed from vertices in the one-ring of each vertex. This can result in unexpected mesh editing behaviour if there is a large variation in the length of edges surrounding a vertex.

In this section, I discuss a simple method which can be used to approximately regularise the facet shape in the deformed mesh in order to ensure that deformation behaviour is predictable. To this end, I define mesh regularisation as the process which adaptively reduces the variation in edge lengths by splitting edges which are considered to be too long.

Before a technique for facet regularisation can be devised, we need to consider some useful properties of Laplacian mesh editing:

- Sorkine and Cohen-Or [2004] showed that the connectivity of a mesh naturally encodes surface information. The vectors $\boldsymbol{\delta}_i$ encode local feature detail, with $\boldsymbol{\delta}_i$ small indicating that a vertex location deviates little from its predicted position based on the surrounding mesh connectivity. By iteratively setting all $\boldsymbol{\delta}_i$ small to

[1]This can be derived for a triangle ABC — the ghost vertex across edge BC is given by $A' = B+C-A$, and the face dual is therefore given by $\frac{2}{3}(B+C) - \frac{1}{3}A$.

zero, the method of Sorkine and Cohen-Or [2004] performs a mesh compression akin to similar signal processing approaches.

- Laplacian mesh deformation has several applications, from sketch based interfaces to detail transference from objects with equivalent connectivity. We focus on the intuitive mesh editing approach of Botsch and Kobbelt [2004]. With this technique, anchors and handles are selected by the user, and then deformed using some rigid body transform. Unless drastic rotational transformations are specified, translational deformations will result in the lengthening of edge vectors *in the direction of the translation.*

I exploit these two properties in defining a method for adaptively regularising a mesh after mesh deformation.

Here I present two basic approaches to mesh regularisation which I have developed. The first approach is applied to the mesh *after* deformation, while the second splits edges *during* deformation. I refer to these approaches as *offline* and *online* regularisation respectively.

4.5.1 An offline approach

An offline approach to mesh regularisation is one which is applied once the deformation has been completed. Given an initial mesh $\mathcal{M}$, a deformed mesh $\mathcal{M}'$ and a deformation operation $def : \mathcal{M} \longrightarrow \mathcal{M}'$ the offline regularisation algorithm is as follows:

```
do
    foreach (edge in $\mathcal{M}'$)
        if ($error(edge) > \varepsilon$)
            Append the tuple $[\beta, edge]$ to splits
    end foreach
    Apply splits to $\mathcal{M}$
    Compute $\mathcal{M}' = def(\mathcal{M})$
while ($splits \neq \emptyset$)
```

Each edge of the deformed mesh $\mathcal{M}'$ is evaluated according to some quality criteria (represented by the function *error*). If the edge fails this test, a split operation is appended to a list of valid splits. Once all edges have been assessed, these split operations are applied to $\mathcal{M}$. The deformation is then reapplied to the split complex. This entire process is applied until no split operations were required. Note that convergence requires the split operation to reduce the *error* function.

The *split* tuple and process of applying splitting operations to a manifold is described in Chapter 6. This simple algorithm has a number of unknowns:

- An error metric *error* and an associated tolerance value ε need to be defined, and
- The barycentric location of the split vertex β must be defined.

An error measure

There are several different methods which have been used in other applications to decide which edge to split in order to regularise the mesh:

- The dihedral angle across the edge could be used [Sovakar and Kobbelt, 2004] which would smooth across sharp edges. In this case, the edge itself would not be split, but rather the neighbouring edges of the two facets on the sharp corner. This approach does not regularise the uniformity of the triangles in the mesh and is therefore not appropriate.
- The edges could be split adaptively based on view or user dependent criteria, such as edges on the silhouette of the object. This approach is not applicable, as it is view and user independent.
- The shape of the input triangle could be used. This approach is common in remeshing algorithms which attempt to reduce triangle degeneracy by remeshing, but requires remeshing the surface.
- The length of the edge could be used.

For this application, the edge length is most appropriate, since it will regularise the shape of long thin triangles in the direction of deformation. We use the Manhattan distance of each edge as a measure, and for ε use double the mean of the Manhattan edge lengths in the mesh. For implementation purposes, the edge vectors of $\mathcal{M}$ are encoded into a sparse matrix form for quick computation of the Manhattan edge lengths.

Vertex location

In deciding on the location of the new vertex I use the finding of Sorkine and Cohen-Or [2004] that the connectivity information of a mesh natural encodes geometric information. By assigning a vertex v_i with even weights $w_j = 1/|\mathcal{N}_i|$ and setting the associated $\boldsymbol{\delta}_i = \mathbf{0}$, the position of the deformed vertex will effectively have a local feature size of zero.

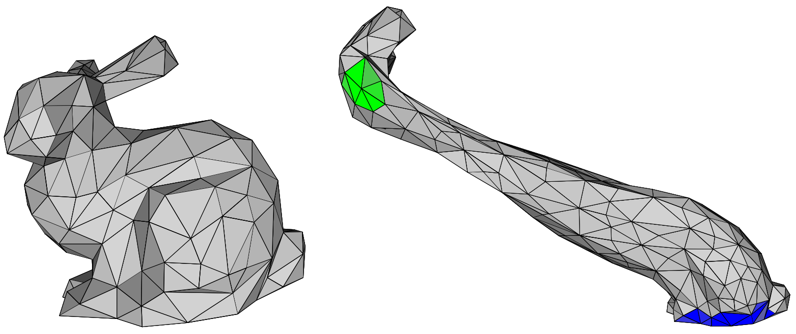


Figure 4.9: In this simplified bunny model, the head is dramatically extruded. Using the offline approach described in Section 4.5.1 edges are adaptively split after deformation has finished to regularise the triangle shape.

Edge splitting will result in a mesh that is not *regular*, in that it increases the number of irregular valence vertices in the mesh. In the surface case, this means that this approach will significantly increase the number of valence 4 vertices in the mesh. Regularity is a useful property in some surface applications, such as subdivision and mesh smoothing.

4.5.2 Online mesh regularisation

A user may wish to see vertices being added to $\mathcal{M}'$ as it is being deformed. Given the Laplacian system takes the form $A^T Ax = A^T b$, it is possible to define an adaptive subdivision matrix operator S which is applied to the solution x at run-time. The connectivity could then also be adaptively updated. The connectivity property of Sorkine and Cohen-Or [2004] cannot be used in this case without recomputing the matrix A and re-factorising.

There are two main problems with this approach. Deducing which edge to split, using a simple technique such as that described in Section 4.5.1, adds a potentially unreasonable overhead to the deformation operation, which we would like to keep as interactive as possible. Recent work by Bóo et al. [2001] describes a hardware accelerated adaptive subdivision implementation, which would dramatically speed up the application of the subdivision operator, but applying this to general dimensional manifolds is still an open problem.

Additionally general dimensional interpolating subdivision rules for manifolds have not yet been developed. While methods exist for interpolating subdivision of curves [Dyn et al., 1987], quadrilateral grids [Kobbelt, 1996] and triangle meshes [Zorin et al., 1996] no interpolating subdivision rules exist for tetrahedral meshes and higher. Note that smooth approximating subdivision schemes exist for tetrahedral meshes, such as that of Schaefer et al. [2004], and an interpolating subdivision operator could be deduced using their approach using a mask based on [Zorin et al., 1996]. This is an area for future work.

4.6 Implementation

The implementation of this system required a factorised modular approach to Laplacian mesh editing:

- It supports manifolds consisting of 2– and 3–simplices in $\mathbb{R}^3$ and $\mathbb{R}^4$ respectively.
- It supports both primal and dual Laplacian editing techniques for comparison purposes.
- It implements several weighting schemes for primal Laplacian editing, namely equal weights [Taubin, 1995], cosine weights [Pinkall and Polthier, 1993] and the quadratic optimisation approach of Zhou et al. [2005].
- It handles manifolds which have a boundary, as described in Section 4.4.

- When rendering an animation manifold, the independent axis and its value can be altered (see Section 2.6).
- It implements the offline adaptive regularisation scheme defined in Section 4.5.1.
- It supports user editing in a form similar to that described by Botsch and Kobbelt [2004].

The system is implemented in C++ with the *LAPACK*, *BLAS* and *CHOLMOD* [Gould et al., 2005, Davis, 2006] libraries performing optimised linear algebra operations, and CGAL [2007] for quadratic programming of positive semi-definite systems. Rendering is in *OpenGL* and the user interface, including 2D interactions, is implemented using *Qt*.

4.6.1 Generic Laplacian editing

Both primal and dual Laplacian editing techniques have a great deal of similarity with respect to the data they store and the operations which they perform. We define the base class for Laplacian editing *Laplacian* as follows:

```
class Laplacian {
public:
    /// Constructor and Destructor
    ...

    /// Add/remove a constraint or anchor
    ...

    /// Build the Laplacian matrix and solve for the inverse
    virtual void initialise(Complex *);

    /// Deform the current Laplacian based on a displacement vector
    virtual void deform(···);

    /// Apply some post-process (optimisation) after the edit
    virtual void restore() {};
    ...

protected:
    /// The locations of the vertices which we are deforming
    cholmod_dense *V;

    /// The matrix of constraints, handles and differential coordinates
    cholmod_dense *Δ;
    /// The matrix which Δ must be multiplied by before solving for V
    cholmod_sparse *L^T;

    /// The factorisation which only has to be done once the deformation begins
    cholmod_factor *fac;

    /// Construct a matrix of differential coordinates from the given complex
    virtual void diffMatrix(Complex *complex);
    ...
};
```

The structure requires the storage of few matrices for the interactive computation of surface deformations. The matrix X stores the original vertex positions, *fac* contains the factorisation of the matrix $L^T L$, computed using Cholesky decomposition, and L^T is the sparse matrix used to quickly compute $L^T \Delta$ in the system $L^T L V = L^T \Delta$.

I have implemented four differential coordinate schemes which are derived from this base class:

- The even weighting method of the original scheme of Taubin [1995],
- the cotangent weights of Pinkall and Polthier [1993] and Desbrun et al. [1999],
- the weights used for the volumetric graph Laplacian of Zhou et al. [2005], and
- the dual graph Laplacian weights of Au et al. [2006].

The first three approaches only require the redefinition of the **diffMatrix()** function in their inherited class. The dual graph approach requires significant modifications to the method in which L is constructed, and additionally requires an optimisation step, accomplished by overloading **restore()**, which is called when the deformation is stopped.

Note that the cotangent weights of Pinkall and Polthier [1993] are only applicable to the surface editing in $\mathbb{R}^3$. For higher dimensional animation editing, it is necessary to use one of the other methods.

4.6.2 User interface

The user interface to this system is integral to the editing behaviour. I have implemented a sketch based interface similar to the method of Botsch and Kobbelt [2004] and others by which the user paints handles and anchors in screen space. The screen space edits are painted onto the mesh and stored in a Laplacian object. Once deformation begins, a trackball is built with a radius derived from the bounding box of the handle vertices. This facilitates deformations based on rotations and translations. Note that with Laplacian editing in general, vertices marked as handles and anchors move little — it is the remaining manifold which is deformed.

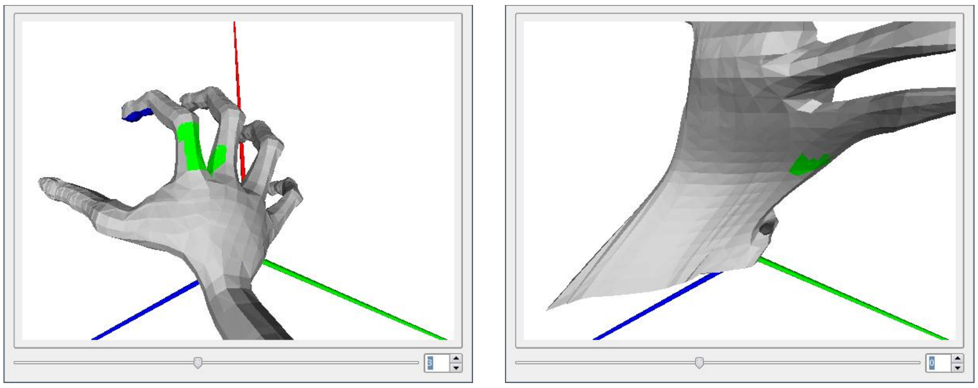


Figure 4.10: This figure demonstrates the 4D editing interface. The user has selected anchors (in blue) and handles (in green) in the frame of the animation sequence on the left. By changing the constant axis (the spinbox on the bottom right) the user views the animation sequence with x constant. Note that handle faces are still visible.

There are some special considerations to be made in the case of higher dimensional animation manifolds. As the user is only presented with a single frame of the animation sequence, it is unlikely for he/she to actually be viewing a keyframe. When the user selects a facet using the screen space selection tool, it selects the space–time simplex from which the rendered facet was extracted. This is demonstrated in Figure 4.10. An alternative option which has not been implemented is to split the simplex from which the selected facet was extracted such that the rendered facet explicitly exists in the current mesh.

Any independent axis of the 4D animation manifold can be rendered, and a simple slider bar is used to alter the value of the independent iso-value so that it may be visualised.

4.7 Comparison of Laplacian techniques

In Figure 4.11 I compare deformation techniques which are applicable to editing manifolds in any dimension. This includes the equal weighting of Taubin [1995], the dual graph Laplacian of Au et al. [2006] and a surface based derivative of the volumetric graph Laplacian of Zhou et al. [2005], which I refer to as the graph Laplacian. The graph Laplacian is a primal method which derives the weights w_{ij} from the quadratic program in Equation 4.2.

It is clear from the results that the dual graph Laplacian technique of Au et al. [2006] performs the best in this example, as deformations are "correctly" propagated across the entire surface to best preserve feature detail. The graph Laplacian does not adequately propagate the deformation across the surface, and twisting and mesh thinning occurs in the region nearest to the top cap, especially when the angle of rotation is extreme. The equal mesh weights behaves predictably poorly, and the mesh effectively "crumples" and self–intersects in the region near the top cap.

The average performance of the test case given in Figure 4.11 is given in Figure 4.12.

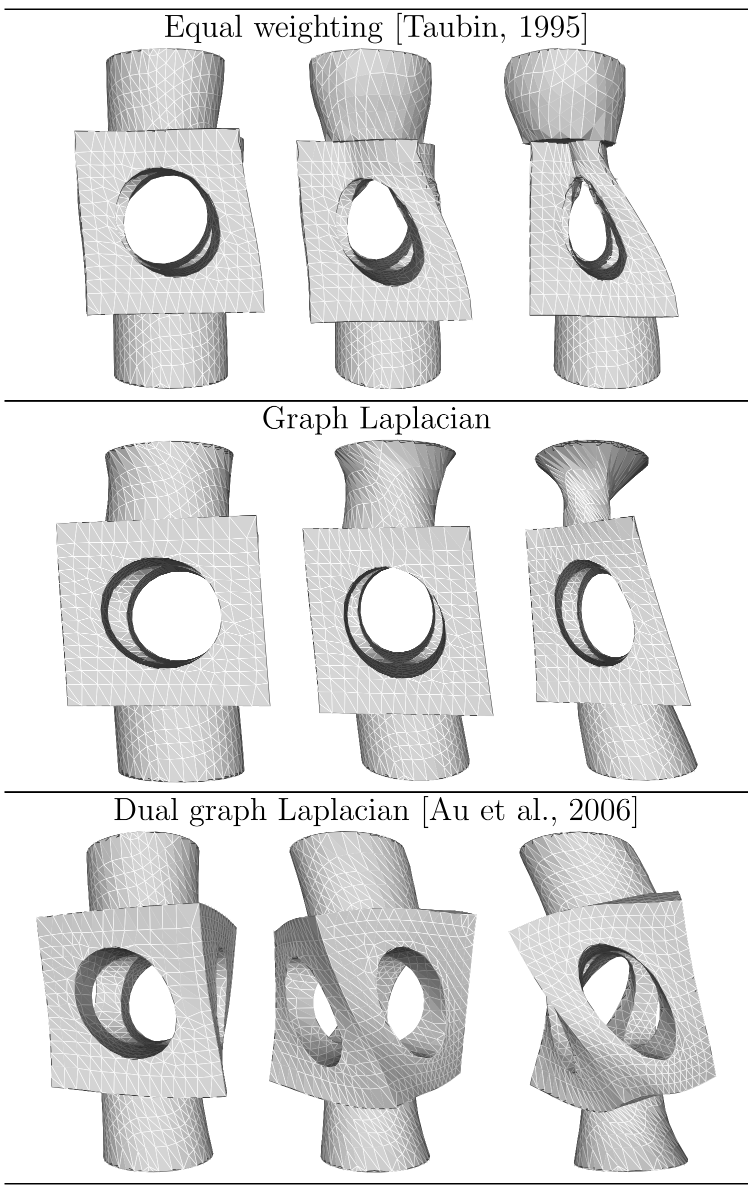

Equal weighting [Taubin, 1995]
Graph Laplacian
Dual graph Laplacian [Au et al., 2006]

Figure 4.11: Rotating an end-cap of the block model is a gruelling stress test for any deformation technique. All vertices of the bottom cap are marked as anchors, while all vertices in the top cap are marked as handles. The handle vertices are rotated around the dominant axis by $\frac{\pi}{4}$, $\frac{\pi}{2}$ and $\frac{3\pi}{4}$ radians, represented from left to right respectively.

Method	**Initialise**	**Deform**	**Restore**
Equal weights	$0.11s$	$0.01s$	$---$
Graph Laplacian	$55.41s$	$0.01s$	$---$
Dual graph Laplacian	$0.27s$	$0.01s$	$5.01s$

Figure 4.12: A comparison of average timing results for the stress test in Figure 4.11. The block model has about 2500 vertices.

The **Initialise** stage consists of building the matrix L and computing the factorisation of $L^T L$. This is very expensive in the case of the graph Laplacian, as it requires the solution to the quadratic programming problem given in Equation 4.2 for every vertex, the performance of which depends entirely on the library used — in this case CGAL. The complexity of setting up the matrix L will be $\mathcal{O}(n)$ in the number of vertices, and since $L^T L$ is very sparse, its factorisation is also $\mathcal{O}(n)$ except for very small systems.

The **Deform** time is the time taken to apply deformations to the positions of handles in Δ and to solve for the system $L^T LV = L^T\Delta$. In all cases involving sparse matrices, the high performance library *CHOLMOD* is used, while dense matrix multiplications are performed with *LAPACK*. With every method tested, deformation is *interactive*, and with most meshes up to 100000 vertices, matrix updating occurs > 30 times a second on a standard commodity PC.

The **Restoration** stage is only applicable to the dual Laplacian technique of Au et al. [2006]. This involves iteratively updating of the feature vectors of Δ and resolving $L^T LV = L^T\Delta$ until the difference in vertex locations from two iterations differ by a small amount. Unfortunately this approach can be unstable, as it can find a local minimum, and in extreme cases not converge at all. It is impossible to accurately evaluate the performance of the restoration stage, as it is dependent on the model, the locations of anchors and handles, as well as the nature and magnitude of the deformation. For most user defined deformations, however, I have found it to converge in only about 20 iterations.

The adaptive regularisation scheme presented in Section 4.5.1 requires repeating the initialisation, deformation and restoration stages for each iteration until no further edge splitting is necessary. It is clear that with the graph Laplacian method this can prove very expensive. I have found that only a single re-computation of the Laplacian is necessary, except in cases of extreme extrusion. In Figure 4.9 only one iteration was necessary to regularise the mesh.

An example of editing in $\mathbb{R}^4$ to produce a topological alteration is given in Figure 4.13 and Figure 4.14. One issue which becomes apparent when using this system is that specifying handles and anchors and applying deformations is difficult.

4.8 Summary

In this chapter I have extended Laplacian mesh editing for use with animation manifolds. These new extensions are

- boundary conditions for both primal and dual Laplacian mesh deformations, and
- an offline mesh regularisation technique for extruded manifolds.

Additionally I have applied several Laplacian editing techniques to editing animation manifolds:

- Equal weighting is the simplest approach to Laplacian editing, but produces the poorest results. It supports manifolds with boundary, and requires no additional

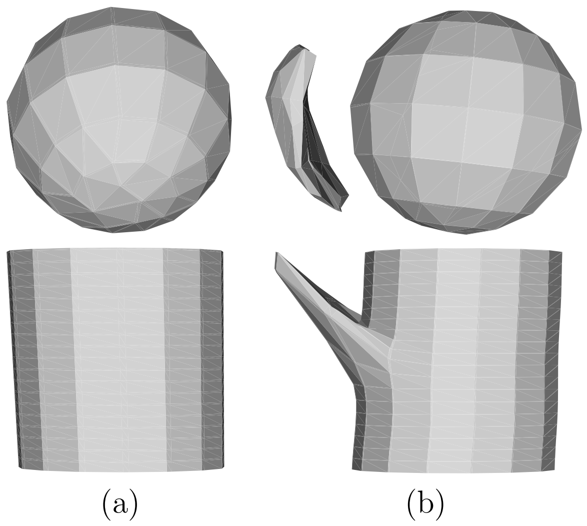


Figure 4.13: This figure shows the equivalent of Figure 4.1 in $\mathbb{R}^4$. In (a) I show a visualisation of the sphere sequence with t and z as the independent axis respectively. The user paints anchors at the top and bottom of the "tube", then paints a handle somewhere in the centre of the tube, and surrounds the handle with anchor points. The handle is then dragged *backwards in time*. The result is a piece of the sphere is separated during the animation. The full sequence is shown in Figure 4.14.

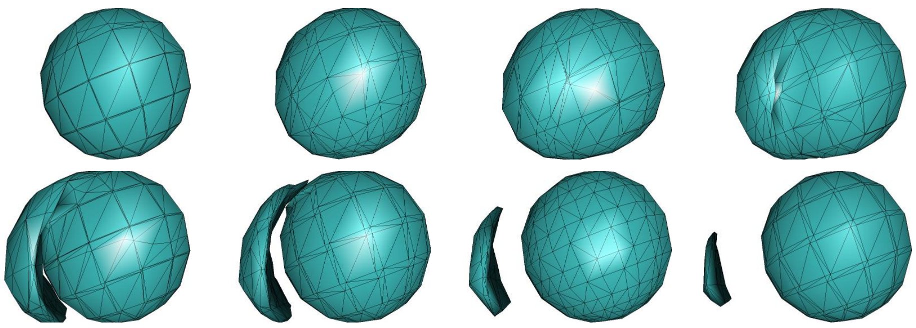


Figure 4.14: Several frames from the sphere splitting operation in Figure 4.13.

computation in deducing the weights or restoring features. It behaves best when a mesh has a uniform face area and edge length distribution.

- The graph Laplacian, adapted from the volumetric graph Laplacian of Zhou et al. [2005] is appropriate for meshes with boundaries and uneven edge lengths, properties which are more common in animation manifolds. However, the computation of the coefficients requires a the solution of a quadratic program for each vertex in the mesh, which is prohibitive in performance.

- The dual graph Laplacian produces the best results of the three methods compared as it restores features and propagates deformations through the entire mesh using an iterative approach. This restoration is expensive, but the results are significantly better. The boundary conditions for the dual graph, however, requires additional constraints, which may be undesirable for a particular deformation.

Laplacian editing is a powerful tool for producing organic looking, multi-scale and feature sensitive deformation. As I have shown, these properties offer interesting applications in editing animation manifolds such as topological alterations and detail propagation.

In this chapter I make use of the ubiquitous *bunny* mesh from the Stanford repository (`http://graphics.stanford.edu/data/3Dscanrep/`), and the *block* mesh acquired from the AIM@SHAPE model repository
(`http://www.aimatshape.net`). All models were available on the date of submission 31 January 2008.

Chapter 5

Connecting Planar Cross-sections

Constrained Delaunay triangulation is a method for triangulating a point set, given a set of constraints such that the constraints are preserved in the simplicial complex of the final triangulation. A conforming Delaunay triangulation is a Constrained Delaunay triangulation for which a Delaunay triangulation of the points conforms with the input constraints. While Delaunay based triangulation represent a large body of current work in Computational Geometry, no implementable method yet exists which supports general dimensional constrained Delaunay triangulation.

For the application presented in Chapter 8, I require a method which allows input manifolds in $\mathbb{R}^3$ to be connected using a triangulation in $\mathbb{R}^4$ such that the original manifolds are preserved in the triangulation.

In this chapter I propose a constrained Delaunay triangulation approach which connects two planar cross–sections in $\mathbb{R}^n$. My approach is initially to make the input cross–sections conforming Delaunay, and then apply a standard Delaunay triangulation to the result. In Theorem 5.2.1 I prove the resulting triangulation will preserve the input contours and is therefore a constrained Delaunay triangulation.

5.1 Background

5.1.1 Delaunay triangulation

The reader may well be familiar with the Delaunay triangulation [Delaunay, 1934]. Richard Shewchuk has suggested that at least one in twelve of all Computational Geometry publications is related to this pervasive topic[Shewchuk, 2002].

In $\mathbb{R}^2$, a triangulation $\mathcal{M} = \{P, F\}$ with points P and triangles F is Delaunay if no point $p \in P$ lies within the bounding circle S of any simplex $f \in F$. This local property is referred to as the Delaunay condition. The Delaunay triangulation of a point set P, denoted by $\mathcal{M} = \mathrm{DT}(P)$, is an operation that constructs the Delaunay triangulation of the point set (see Figure 5.1).

The principles presented here extend to $\mathbb{R}^n$ — triangles (2-simplices) become n-simplices and bounding circles are hyperspheres in $\mathbb{R}^n$.

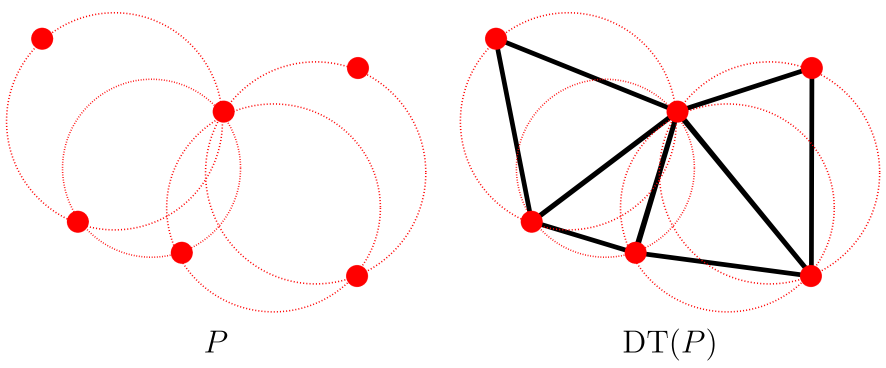


Figure 5.1: A Delaunay triangulation of point set P.

A triangulation is said to be *strongly Delaunay* if it is Delaunay, with the added condition that no point p lies on S. Note that if a triangulation is not strongly Delaunay, then on one or more occasions, 4 or more points lie on the same bounding circle, causing there to be multiple viable triangulations. A unique solution can be "manufactured" by applying small random perturbations to the input data [Barber et al., 1996], or the paraboloid lifting method of Edelsbrunner and Seidel [1986].

The topic of Delaunay triangulation is well studied [Edelsbrunner, 2001, Boissonnat and Teillaud, 2007], and two main algorithms exist for their construction:

- Most commonly, an **incremental** point insertion approach is used, also called *gift-wrapping* or *sweep-line*. A triangulation $\mathcal{M}$ is initialised with 3 points and the single 2-simplex which passes through these. Each remaining point of P is added to this mesh in turn. The simplices which are affected are removed and re-triangulated.

  A naïve algorithm for incremental Delaunay construction is $\mathcal{O}(n^2)$, which can be improved to $\mathcal{O}(n \log n)$ using a sweep-line algorithm. Optimisations of this form take advantage of the spatial coherence of input vertices which dramatically improves performance, as in the streaming method of Isenburg et al. [2006].

- A *divide and conquer* approach, first introduced by Lee and Schachter [1980] subdivides the problem domain into spatial cells which are solved separately. A clever merge operation merges neighbouring cells with $\mathcal{O}(n)$ complexity, bringing the complexity to $\mathcal{O}(n \log n)$. This method was later extended to arbitrary dimensions by Cignoni et al. [1998].

A Delaunay triangulation has several useful properties:

1. It fills the *convex hull* of the points P.

2. It is unique if it is strongly Delaunay.

3. It naturally minimises the maximum angle of the output triangles.

The last two properties make this method very attractive to methods which require a high quality triangulation for numerical stability, such as Finite Element Methods (FEM).

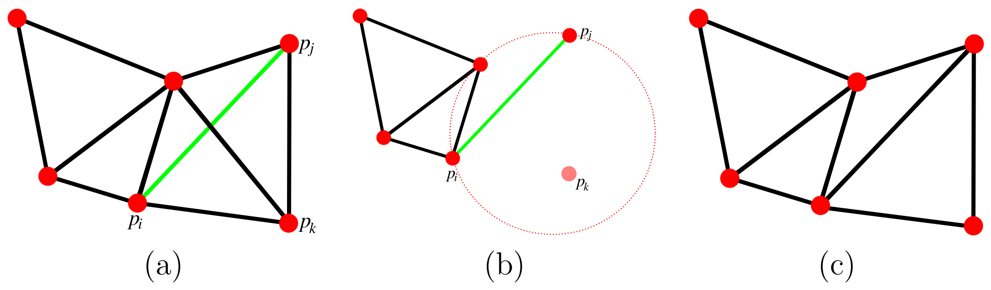


Figure 5.2: Constrained Delaunay Triangulation. In (a) the constraint $\overline{p_i p_j}$ violates the Delaunay condition. The Delaunay triangulation is recomputed in (b) by making p_k "invisible" to the circumcircle containing $\overline{p_i p_j}$. This allows the point p_k to lie within a circumcircle, and produces the Constrained Delaunay Triangulation in (c).

5.1.2 Constrained Delaunay Triangulation

A constrained Delaunay triangulation is a special class of Delaunay triangulation in which certain *features* are required to be present in the resulting triangulation, possibly violating the Delaunay condition. Features take the form of any k-simplex, $k = 1 \dots n$ (the 0-simplices will always be preserved by a standard Delaunay triangulation). This approach has several applications, such as embedding surface features or filling the space outside or inside a triangle mesh for use with finite element meshing.

The CDT operations $\mathcal{M} = \text{CDT}(P, \mathcal{X})$ returns a triangulation of the point set P such that for a set of input constraints $\mathcal{X}$, $\mathcal{X} \subset \mathcal{M}$.

A simple method of defining a CDT algorithm is to think of the local CDT property in terms of *point visibility*. Two points p_i and p_j are *visible* from eachother if there is no occluding simplex $x \in \mathcal{X}$. A simplex $f \in F$ is *constrained Delaunay* if there is a bounding circle S of f such that no vertex of $\mathcal{X}$ inside S is visible from any point in the relative interior of f. An example of point visibility is shown in Figure 5.2.

A typical CDT algorithm takes the form of a series of topological flips [Edelsbrunner and Shah, 1996] which allow edges to conform with the constraints complex $\mathcal{X}$. In $\mathbb{R}^3$ the situation can becomes considerably more difficult due to examples such as the Schönhardt prism [Schönhardt, 1928] in Figure 5.3 which simply cannot be meshed by this technique without vertex insertion. Unfortunately the problem of whether additional vertices are

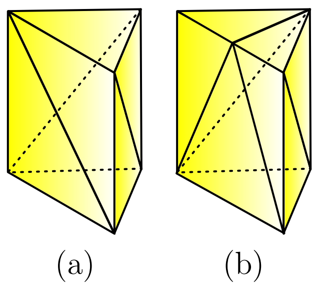
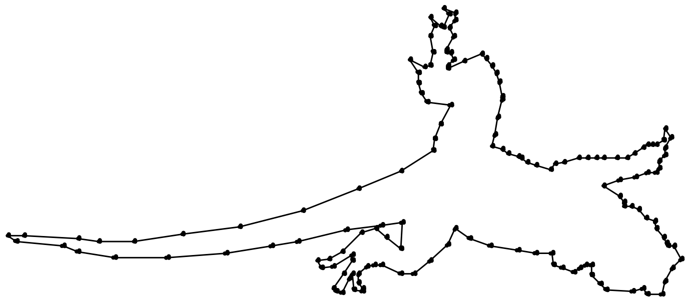
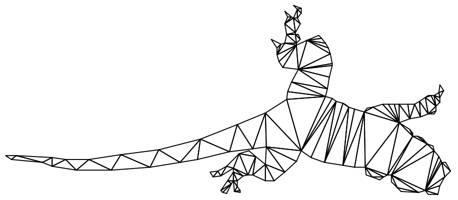


Figure 5.3: The Schönhardt prism in (a) cannot be meshed with a CDT using only edge flips. This is fixed in (b) by splitting a single edge.

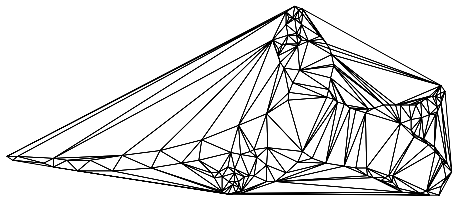
required to mesh a generic polyhedron has been shown in Ruppert and Seidel [1989] to be NP-complete. A condition for a higher dimensional CDT was examined by Shewchuk [1998], but the general case of the CDT in higher dimensions remains an open problem.



Point set P and constraint complex $\mathcal{X}$

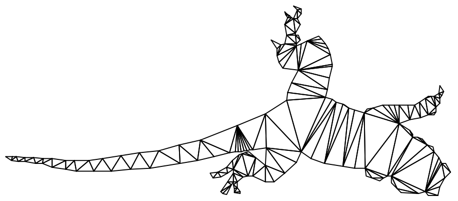


(a)

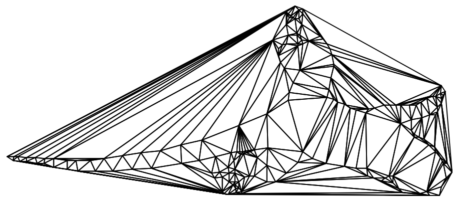


(b)

CDT$(P, \mathcal{X})$



(c)



(d)

C$_F$DT$(P, \mathcal{X})$

Figure 5.4: Constrained and Conforming Delaunay Triangulation. The lizard simplex is triangulated with a constrained Delaunay triangulator in (a). In (b) a regular Delaunay Triangulation is performed on the point set of (a). Note that the original constraint complex is not preserved. In (c) a conforming Delaunay triangulation is used on the input complex using Ruppert's method. An unconstrained Delaunay triangulation of the points still preserves the original constraints in (d).

For most applications, the resulting quality of the CDT is as important as whether the constraints themselves are met. The constraints complex $\mathcal{X}$ could contain sharp corners or regions which it is not possible to mesh without using very thin or degenerate triangles. This is resolved by inserting new vertices in the original point set and splitting $\mathcal{X}$ at appropriate locations. These new points are called *Steiner points*.

A Steiner point is any point which is added to the point set in order to comply with some geometric or quality constraint. A comparison of Steiner point insertion algorithms can be found in Shewchuk [1997]. A standard method of improving triangle quality in the resulting triangulation, or enforcing smooth triangle density grading, is to insert a point into P at the circumcentre of the bounding triangle of a degenerate triangle [Chew, 1993].

The resulting triangulation is guaranteed to have fewer degeneracies, but may violate one or more of the input constraints.

5.1.3 Conforming Delaunay Triangulation

A conforming Delaunay triangulation $\mathrm{C}_F\mathrm{DT}$ is a triangulation which conforms with the input constraints $\mathcal{X} \subset \mathrm{C}_F\mathrm{DT}(P, \mathcal{X})$ and inserts vertices into P to form P' such that $\mathcal{X} \subset \mathrm{DT}(P')$. The point set P' is therefore constructed so that the *Delaunay triangulation* preserves the input constraints $\mathcal{X}$.

This is demonstrated in Figure 5.4. It has some useful properties:

- Unlike a CDT, $\mathrm{C}_F\mathrm{DT}$ is *truly Delaunay*. This means that no topological flips are required to ensure the preservation of constraints, and triangle quality is implicitly good.
- Point insertion allows us to deal with challenging meshing problems such as those in Figure 5.3. Splitting one edge of the Schönhardt prism allows it to be meshed.

According to Si [2007] a true $\mathrm{C}_F\mathrm{DT}$ is unnecessary for most applications. A combination of techniques from CDT and $\mathrm{C}_F\mathrm{DT}$ can yield a quality, constrained triangulation which is sufficient for finite element meshing. I will make use of the $\mathrm{C}_F\mathrm{DT}$ methods of Shewchuk [1996] in $\mathbb{R}^2$ and Si and Gaertner [2005] in $\mathbb{R}^3$.

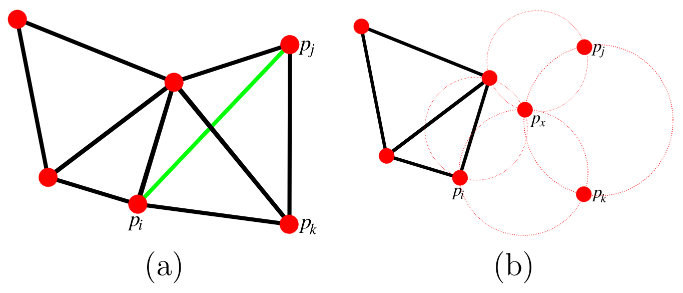
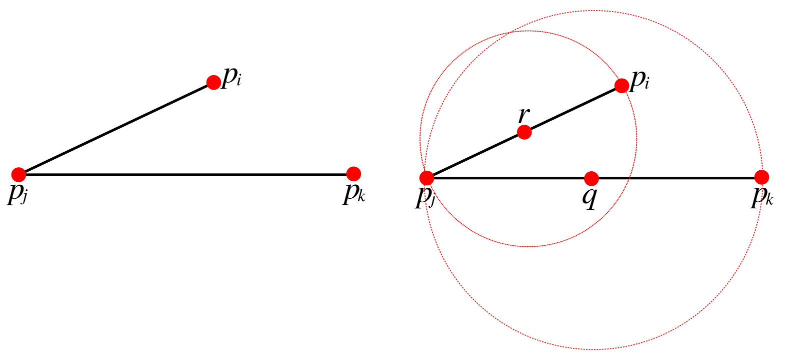


Figure 5.5: Conforming Delaunay Triangulation. In (a) the constraint $\overline{p_i p_j}$ violates the Delaunay condition (same as Figure 5.2). The Delaunay triangulation is recomputed in (b) by inserting a point p_x into the point set at a location which enforces the edge $\overline{p_i p_j}$ in the form $\overline{p_i p_x} + \overline{p_x p_j}$. In (c) the resulting triangulation observes the constraint, and conforms with the Delaunay condition.

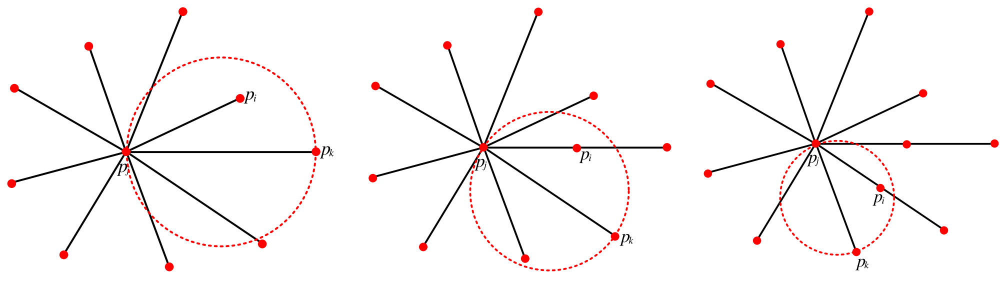
A vertex $p \in P$ is said to encroach upon a segment $x \in \mathcal{X}$ if it lies within the diametrical circle of x. The algorithm requires that a feature $x \in \mathcal{X}$ is split at its *midpoint* if another vertex encroaches on it. If no vertex encroaches on a segment, then $\mathrm{DT}(P)$ is guaranteed to preserve the features in $\mathcal{X}$, and is said to have the *Gabriel property*. The algorithm generates meshes which, assuming no input angle is less than $90°$, provably terminates.

The angle limitation can result in loops preventing termination, as in Figure 5.6(b). Ruppert [1995] introduces "shielding spheres" at input sites which are used to cut corners



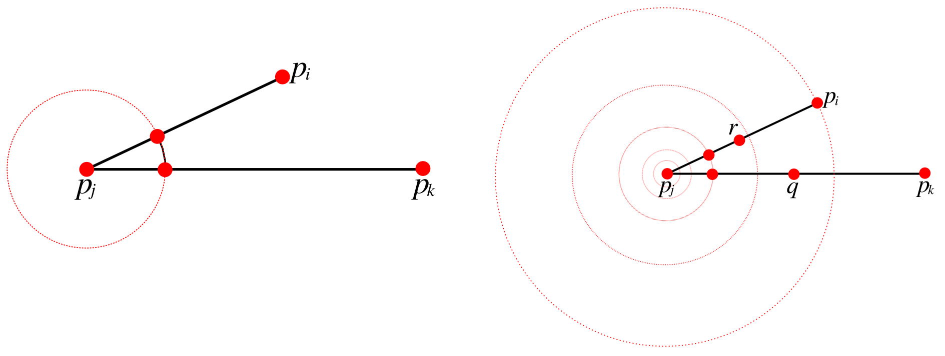
(a) Initial $\mathcal{X}$ (b) Midpoint point insertion

Figure 5.6: A problem arising in edge splitting methods for CDT is that for input angles small enough, a loop can occur. In (b), after each edge has been split in its midpoint, r lies within the diametrical circle of feature $\overline{p_j p_k}$, and q lies within the diametrical circle of feature $\overline{p_j p_i}$. A loop occurs and the edge splitting algorithm does not terminate.



(a) (b) (c)

Figure 5.7: Similar to the termination problem in Figure 5.6, the spoke configuration requires several edges to emanate from a single central vertex p_j. In each of (a), (b) and (c), the vertex p_i encroaches on the edge $\overline{p_j p_k}$. It is clear that the ordering of the encroaching tests may result in a non-terminating spiral. This configuration occurs commonly in triangle meshes about high valence vertices of the input surface.



(a) Shielding sphere (b) Concentric circle point insertion

Figure 5.8: In (a) the mesh is modified by inserting a *shielding sphere* into $\mathcal{X}$. The newly inserted vertices ensure that no angle in $\mathcal{X}$ is less than 90°and guarantees termination. Ruppert also proposed the method in (b), which will ensure that inserted points lie on the same concentric circle.

and reduce the input angles about these sites, a process referred to elsewhere as *edge protection* (see Figure 5.8(a)).

These are computationally expensive, and dramatically increase the algorithm complexity. Additionally, Ruppert suggests adaptively splitting edges by using the intersection of concentric circles surrounding each original vertex with the constraining edge (see Figure 5.8(b)). Initially a set of circles C_i with doubling radius are are generated about each point $p_i \in P$. For each split other than the first (when the midpoint is still used) the intersection point of C_i with $\mathcal{X}$ is used which is closest to the midpoint of the edge. As a result the points inserted into $\mathcal{X}$ surrounding p_i are co-circular. The algorithm was also modified by Miller et al. [2003], allowing for sharper corners by using an adaptive edge splitting approach.

Ruppert's method has been applied to triangulations in $\mathbb{R}^2$. Several problems arise when applying this algorithm to Constrained and Conforming Delaunay triangulations in $\mathbb{R}^3$ and higher:

- Edge protection is considerably more expensive to compute, and results in numerous extra Steiner points to be inserted into $\mathcal{X}$[Murphy et al., 2001]. This is alleviated to a degree by adaptive sphere radii [Cohen-Steiner et al., 2004], or using two concentric shielding spheres [Si and Gaertner, 2005]. A full version of Ruppert's concentric spheres algorithm (see Figure 5.8(b)) is not computationally feasible in $\mathbb{R}^3$.
- Ambiguities can arise which require special handling. These can be dealt with using a *parabolic lifting map* [Edelsbrunner and Seidel, 1986] but this cannot deal with the problem in Figure 5.3.

To my knowledge, all CDT and C$_F$DT algorithms in $\mathbb{R}^3$ make use of some combination (or modification) of

- Ruppert's [Ruppert, 1995] method to preserve constraints,
- Chew's [Chew, 1993] methods to improve triangle quality, and
- Edelsbrunner and Shah's [Edelsbrunner and Shah, 1996] topological flip operations to do either of these.

There are still many practical limitations to these, such as the 30° minimum dihedral angle limit in the package *Tetgen* [Si and Gaertner, 2005]. It is perhaps due to the complexity of CDT and C$_F$DT in $\mathbb{R}^3$ (see Grislain and Shewchuk [2003]) and the lack of applications for it that no practical implementation exists in dimensions higher than 3.

5.2 Conforming higher dimensional triangulations

In the context of defining a transition between planar cross-sections, we have as our inputs two constraint meshes $\mathcal{S}$ and $\mathcal{T}$ representing the source and target states of our transition

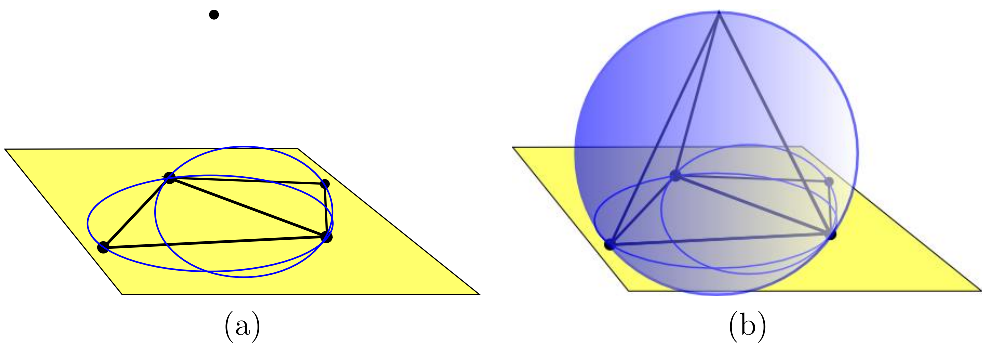


Figure 5.9: Preserving conformal Delaunay triangulations in higher dimensions. This diagram is referenced by Theorem 5.2.1.

respectively. Both $\mathcal{S}$ and $\mathcal{T}$ are closed, oriented contours / surfaces in $\mathbb{R}^n$. I define the combined space

$$\mathcal{F} = \text{embed}(\mathcal{S}, 0) \cup \text{embed}(\mathcal{T}, 1)$$

where embed$(\mathcal{M}, \tau)$ is simply an operator which embeds $\mathcal{M}$ within a $\mathbb{R}^{n+1}$ space at position $t = \tau$.

We define the function $\mathcal{M}' = \text{conform}(\mathcal{M})$ as a function which returns the modified constraint manifold resulting from $\text{C}_F\text{DT}(\mathcal{M})$.

The problem is to find a triangulation of $\mathcal{F}$ which conforms with $\mathcal{S}$ and $\mathcal{T}$. We resolve this by first finding $\mathcal{S}' = \text{conform}(\mathcal{S})$ and $\mathcal{T}' = \text{conform}(\mathcal{T})$. Now, with

$$\mathcal{F}' = \text{embed}(\mathcal{S}', 0) \cup \text{embed}(\mathcal{T}', 1),$$

$\text{DT}(\mathcal{F}')$ will conform to $\mathcal{F}$.

Theorem 5.2.1 $\text{DT}(\mathcal{F}')$ *conforms with* $\mathcal{S}$ *and* $\mathcal{T}$*.*

Proof Recall that the basic Delaunay condition is that a simplex s is Delaunay if no other point in the complex lies within the bounding hypersphere of s. Similarly a complex is considered Delaunay if every simplex is Delaunay.

Let us consider the problem of connecting $2D$ planar cross sections in $\mathbb{R}^3$. A conforming Delaunay triangulation of a complex uses point insertion in order to ensure that a basic Delaunay triangulation of the complex preserves the constraints.

Consider the planar cross sections embedded in $\mathbb{R}^3$. In Figure 5.9(a) we see a planar complex and a point in another planar complex. Should the point be connected to the triangle in Figure 5.9(b) then any circumsphere including the triangle in the plane must pass through the circumcircle of those three points.

We know that no other point within the plane lies within the bounding circle of those three points as the plane itself is Delaunay. Additionally no point in the parallel plane can be within the bounding sphere as the sphere only touches the plane at this single point. Therefore the triangle must be preserved within the final manifold. This argument holds in any dimension. ▮

Theorem 5.2.1 has several implications:

- It gives us the machinery to build conforming triangulations in higher dimensions between planar cross-sections, in particular between triangle meshes in $\mathbb{R}^3$.
- It allows us to exploit the actively studied field of CDTs and C$_F$DTs in $\mathbb{R}^3$.
- It reduces the complexity of the problem of higher dimensional CDTs to a simple Delaunay triangulation in $\mathbb{R}^4$.

5.3 An algorithm for connecting planar cross-sections

A simple algorithm can be derived from Theorem 5.2.1, which is shown in Figure 5.10.

```
Given a source mesh $\mathcal{S}$ and target mesh $\mathcal{T}$:
    Find $\mathcal{S}' = \text{conform}(\mathcal{S})$ and $\mathcal{T}' = \text{conform}(\mathcal{T})$
    Create $\mathcal{F}' = \mathcal{S}' \cup \mathcal{T}'$
    Find $\text{DT}(\mathcal{F}')$
```

Figure 5.10: An algorithm for finding a full space triangulation of $\mathcal{F}$ which conforms with the input cross-sections $\mathcal{S}$ and $\mathcal{T}$.

DT($\mathcal{F}'$) fills the convex hull of the points of $\mathcal{F}'$. Trimming concave regions in $\mathbb{R}^2$ is trivial, but in $\mathbb{R}^3$ and above ambiguities arise when attempting to remove simplices from the full space triangulation[1] of $\mathcal{F}'$. This will be discussed in more depth in Chapter 8.

The performance of these algorithms depends entirely on the performance of CDT and DT algorithms — all remaining operations entail data shuffling and file handling operations which are trivial and platform dependent. I used the packages *Triangle* [Shewchuk, 1996] and *Tetgen* [Si and Gaertner, 2005] for 2D and 3D C$_F$DT respectively, and the *QHull* package [Barber et al., 1996] for finding general dimensional Delaunay triangulations.

5.4 Results and discussion

An example of two connected planar contours using the algorithm in Section 5.3 is shown in Figure 5.11. Neither the *lizard* nor the *blobs* contours are initially conforming — both are made conforming using the *Triangle* package. The final result is used to show that in the final meshed result (in this case, a tetrahedral mesh) the result conforms with $\mathcal{S}$ and $\mathcal{T}$.

Note that CDT of 2D contours in $\mathbb{R}^3$ is possible with existing packages (such as *Tetgen*) by specifying them in the form of edges of a planar straight line complex. However, finding a CDT of 3D contours in $\mathbb{R}^4$ is not possible with other methods.

[1]A full space triangulation in $\mathbb{R}^n$ consists of n-simplices.

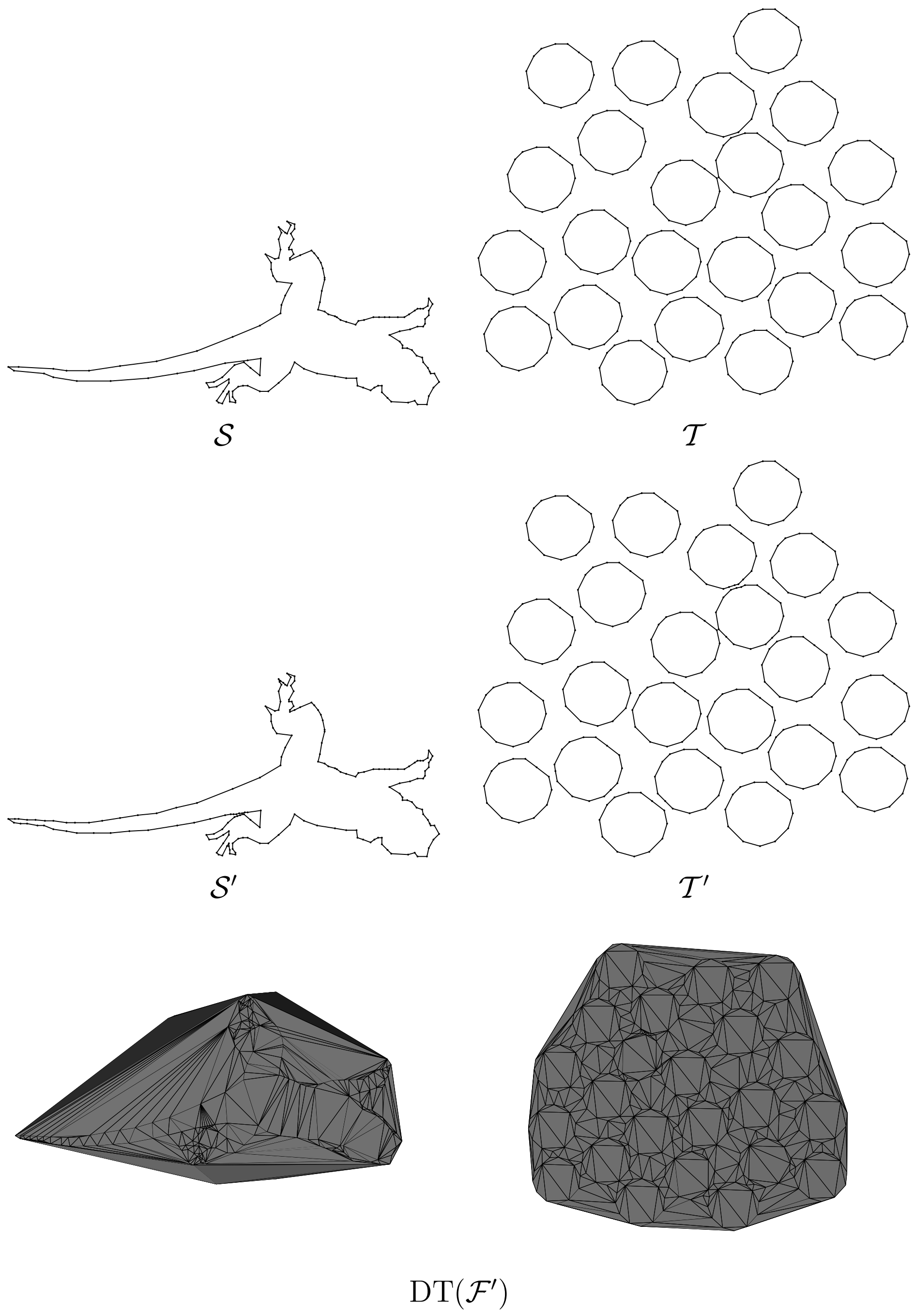


Figure 5.11: An example of connecting 2D contours in $\mathbb{R}^3$ using the algorithm described in Section 5.3.

Unfortunately, while it is possible to show the results of a triangulation in $\mathbb{R}^3$, it is impossible to visualise the full space triangulations in $\mathbb{R}^4$. These results will be shown in Chapter 8.

5.4.1 On meshing between contours

Initially I attempted to following an active front meshing approach which was not full space. We only require the boundary of $\mathrm{DT}(\mathcal{F}')$ in order to find frames of a sequence such as the one in Figure 8.7. However, while methods to determine the orientation of an active front algorithm work but for connecting 2D contours embedded in $\mathbb{R}^3$ with a triangulation (such as in Bajaj et al. [1996]), for 3D contours in $\mathbb{R}^4$ this approach does not work.

This is because in $\mathbb{R}^3$ the direction along the boundary of the 2D contour can be deduced from the ordering of the points on an oriented edge. In contrast, identifying where a active front can advance from 3D contours in $\mathbb{R}^4$ requires the orientable triangles (faces of tets) in $\mathbb{R}^4$.

In this chapter I instead present a simple method to fill the entire space between two constraints, from which the boundary can be quickly extracted.

Chapter 6

Barycentric Refinement

In this Chapter I present a new method that can be used to split a given connected simplicial complex given a list of tuples consisting of simplices to be split, and the barycentric coordinates of the desired vertex.

Barycentric coordinate splitting provides a simple method to insert vertices into a manifold in any dimension. Unlike the ordered approach of atomic operations used by Hoppe [1996], the ordering of operations is arbitrary. Affected operations are updated using the split operator described in Section 6.1. Additionally, since the new vertex location is encoded relative to the vertices of each facet, re-computation of the vertex location is unnecessary.

The barycentric splitting approach is applicable to any method which requires the insertion of vertices into a manifold while preserving consistent connectivity. Additionally it applies to simplices embedded in any dimension. There are a number of applications of this approach:

- *Point set embedding:* In some surface editing applications it is useful to insert vertices into a manifold so that a feature, can be placed on the surface. A simple application of this is shown in Figure 6.3. Note that a contour connectivity cannot be enforced with the given algorithm.
- *Mesh regularisation:* An adaptive refinement approach based on the above algorithm could adjust the density of vertices in selected regions of the surface. This is used in Section 4.5.1 to regularise the mesh after deformation, and may also be applicable to Finite Element methods in which the shape and regularity of the triangulation is important.
- *Voronoi skeleton extraction:* In Chapter 7 I use this refinement method to enforce Voronoi separability in an input manifold. My method for enforcing Voronoi separability conveniently lends itself to a barycentric refinement method.

There are a number of useful properties of this approach:

- *Simplicity*: Each split is applied locally and affects only a small region of the surface. A Delaunay based triangulation approach would require the determination of coplanar facets.

- *Ordering*: There is no particular ordering in which splits need to be applied, unlike hierarchical approaches such as that of Hoppe [1996]. This gives us the freedom to specify an ordering which reduces resulting triangle degeneracy in Section 6.3.
- *Topological consistency*: At each step of the refinement, the mesh has valid topology and connectivity. This allows operations to be applied incrementally rather than in a batch.

6.1 The split tuple

Barycentric coordinates are used to define a relative point location within a simplex. Any point within a given simplex is defined by an affine combination of the vertices of the simplex. For example, any point in an $(n-1)$-simplex is defined by $q = \sum_{i=1}^{n} \gamma_i p_i$, with $\sum_{i=1}^{n} \gamma_i = 1$ and for all i, $0 \leq \gamma_i \leq 1$. The coefficients γ_i are called the barycentric coordinates.

A split tuple $[\mathbf{b}, f]$ consists of an $(n-1)$-simplex f and the associated barycentric coordinate $\mathbf{b} = \{\gamma_i\}, i = 1 \ldots n$. The *split* operator creates a new vertex in f which is associated with the tuple and updates the surrounding connectivity.

Once a split tuple has been applied, other dependent splits tuples need to be updated. We consider the general dimensional case that an $(n-1)$-simplex is split to form vertex $q_1 = \sum_{i=1}^{n} \gamma_i p_i$, invalidating another split operation applied to that same simplex which would create $q_2 = \sum_{i=1}^{n} \beta_i p_i$.

We can write:

$$\begin{aligned}
-\frac{1}{\gamma_j}\sum_{\substack{i=1\\i\neq j}}^{n} \gamma_i p_i + \frac{1}{\gamma_j} q_1 &= p_j \\
\sum_{\substack{i=1\\i\neq j}}^{n} \beta_i p_i + \beta_j \left(-\frac{1}{\gamma_j}\sum_{\substack{i=1\\i\neq j}}^{n} \gamma_i p_i + \frac{1}{\gamma_j} q_1 \right) &= q_2 \\
\sum_{\substack{i=1\\i\neq j}}^{n} (\beta_i - \gamma_i \frac{\beta_j}{\gamma_j}) p_i + \frac{\beta_j}{\gamma_j} q_1 &= q_2 \qquad (6.1)
\end{aligned}$$

Equation 6.1 gives us new barycentric coordinates for the split cell based on the original corner p_j being replaced by the new vertex q_1. In order to update the affected tuple which inserts vertex q_2 into the simplex, both the barycentric coordinates and the associated simplex need to be identified.

$$\bar{\beta}_i = \begin{cases} \beta_i - \gamma_i \frac{\beta_j}{\gamma_j} & i \neq j \\ \frac{\beta_j}{\gamma_j} & i = j \end{cases}$$

The replaced corner p_j is one for which all the new coefficients comply with $0 \leq \bar{\beta}_i \leq 1$. This only true if $\frac{\beta_i}{\gamma_i} \geq \frac{\beta_j}{\gamma_j}$ for all j, $j \neq i$. This simple ratio test gives us a quick way to determine which new triangle our original barycentric coordinate will lie in. Note that degenerate cases, such as the vertex lying on an old or a new edge, will result in division by zero. These are dealt with as special cases.

In the case where the dimension of one of the simplices being updated if different to the dimension of the split simplex, we simply use $\gamma_i = 0$ and $\frac{\beta_i}{\gamma_i}$ very large in the above formulation, leaving the updated barycentric coordinate unchanged, i.e. $\bar{\beta}_i = \beta_i$. This allows us to define rules for updating barycentric coordinates for splitting arbitrary simplices.

6.2 A splitting algorithm

```
while ($split \neq \emptyset$)
    Apply the first operation $s = \{\mathbf{b}, f\}$ in the list
    if ($f$ is a $(n-1)$-simplex)
        Update operations in split neighbouring $f$
    else
        Update operations in split neighbouring the simplicial complex of $f$
    Remove $s$ from split
end while
```

Figure 6.1: An algorithm for applying split tuples to a mesh.

An algorithm for refining a mesh based on a list of split tuples is given in Figure 6.1. My implementation uses a list of split operations, hashed by a key generated from the simplex. The order of complexity of this algorithm is in general $\mathcal{O}(n)$ where n is the number of operations, as the number of affected operations is typically a small number dependent on the application. An efficient hash function will reduce the search complexity of finding affected operations to $\mathcal{O}(1)$.

6.3 Reducing face degeneracy

This simplex splitting process will allow all barycentric tuples to be applied to the mesh while maintaining consistent topology. However the manifold may contain thin triangles or *slivers* as a result of such split operations. For example, adding a vertex to the midpoint of a perfectly equilateral triangle creates three poorly shaped triangles.

Slivers will form when a new point lies within a face, but particularly close to an edge. These can be quickly identified — if the vertex lies on an edge, then the contribution of the vertex opposite that edge is 0, which is represented by the corresponding barycentric coordinate. Our approach is to identify any $\gamma_i < \epsilon$ and downgrade the tuple by removing the component of the barycentric coordinates $< \epsilon$ and normalising the remaining coordinates.

If all but one of the barycentric coordinates have $\gamma_i < \epsilon$, then the new vertex defined by this operation is very close to an existing vertex. In this case, we remove the tuple. The variable ϵ is a user defined *sliver tolerance value*.

Removing a tuple will prevent the insertion of a point into the manifold as another point that is sufficiently close already exists. This is a useful feature with reference to the algorithm of Section 7.5 where duplicate tuples may arise in the list of split operations. Applying both split operations would result in very thin, degenerate faces.

Another approach to reduce slivers in the resulting mesh is to measure the face degeneracy which would result after the application of an operation. Each tuple can be ranked according to the degeneracy that would result from performing the split. I use the *variance* of the triangle edge lengths as a measure of degeneracy for a facet [Smith, 2007]. The *maximum* measured degeneracy from all the facets that would be created by the operation is used as a cost associated with the application of the tuple.

I process the tuples in a priority queue in a greedy fashion, sorted on error. Tuples which are affected by an applied split need to have error updated when they are reinserted into the queue. One convenient property of a greedy approach is that operations which would normally result in degeneracies are applied towards the end of the split sequence, allowing the triangle slivers to be restricted to the final stages of the split algorithm. The results of using this approach are shown in Figure 6.3.

6.4 Example

I apply a set of barycentric splitting operations to an input mesh representing the surface of an icosahedron in Figure 6.2. In this example, we initially begin with the following list of barycentric split operations:

1. $[A, B], [0.5, 0.5]$
2. $[A, B], [0.4999999, 0.5000001]$
3. $[B, C], [0.5, 0.5]$
4. $[A, B, C], [0.3, 0.3, 0.4]$

After applying the first operation, the affected operations are checked. The second operation is updated, and is determined to have a barycentric coordinate $< \epsilon$, so it is discarded. One updated barycentric component of operation 4 is also determined to be $< \epsilon$, so that component is discarded and the operation renormalised. After this operation, the operation list becomes this:

1. $[B, C], [0.5, 0.5]$
2. $[D, C], [0.6, 0.4]$

Both these operations can be applied without further updates of the operation list.

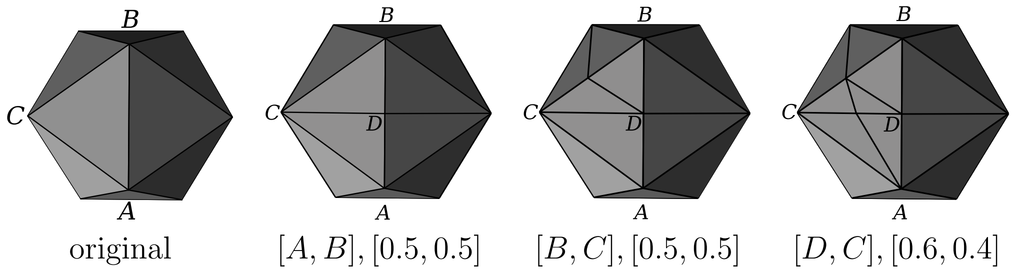


Figure 6.2: Splitting an icosahedron with barycentric splitting operations.

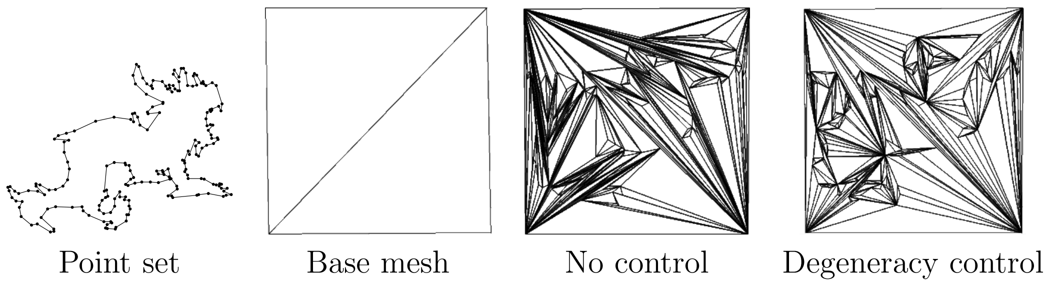


Figure 6.3: The point set from the input contour is embedded in the base mesh — the side of a cube. Without any form of error measure degenerate triangles occur. With the degeneracy measure, fewer degenerate triangles are produced. Note that the connectivity of the contour itself cannot be reproduced with this method.

6.5 Discussion

The barycentric refinement algorithm presented does not typically produce well formed, fair and regular meshes. It is applicable to the applications I present here because of its speed and simplicity. Additionally it is progressive and atomic, allowing operations to be performed independently.

For the applications which I present in this document, triangle degeneracy in the resulting mesh are not worsened since only a small number of operations are typically performed in the same region of the mesh.

An alternative approach to embedding vertices in facets is by using a Delaunay triangulation approach. For each face f_i:

- Append all split operations affecting that face to *splitList*.
- Append the locations of all vertices of *splitList* to *vertList*.
- Rotate f_i and *vertList* onto a lower dimensional plane.
- Use Delaunay triangulation on the vertices of f_i and *vertList* to deduce connectivity.

The mesh must then be reassembled, and degenerate facets (which can arise from operations which split edges) must be removed.

Chapter 7

The Voronoi skeleton

Many applications in computational geometry, computer graphics and computer vision require a *topological skeleton.* The most well known of these is the *medial axis*, which is also unfortunately the most difficult to accurately compute. The medial axis of a contour in $\mathbb{R}^2$ can be defined as the centres of an infinite set of circles which are tangential to the surface at two or more points and do not intersect the surface.

Many authors have noted that a subset of Voronoi diagram of a point set sampled from a surface closely resembles the medial axis[Attali et al., 2007]. In this chapter I prove that as the sampling density of the surface increases, the Voronoi diagram tends to the medial axis.

In order to find the skeleton, I define *Voronoi separability* — a property of a surface mesh $\mathcal{M}$ which requires that the Voronoi diagram of the vertices of $\mathcal{M}$ does not intersect $\mathcal{M}$. The Voronoi skeleton is then a subset of the Voronoi diagram of the vertices of $\mathcal{M}$, and is an approximation to the medial axis of $\mathcal{M}$. I will present an algorithm for ensuring this property, and conditions for existence. The Voronoi skeleton can be extracted from a surface which has this property.

While this method applies to closed, oriented manifolds in $\mathbb{R}^2$, a dihedral angle limitation applies to the types of input manifolds which can be processed in $\mathbb{R}^3$. I will compare this method to existing Voronoi based skeletonisation techniques in terms of their efficacy in identifying the skeleton, and show that despite its limitations, the Voronoi skeleton is a more complete skeleton, which by definition will not intersect the input manifold.

7.1 Preliminaries

A Voronoi cell C_i of a point $p_i \in P$ in $\mathbb{R}^n$, $i = 1 \ldots |P|$ is a spatial partition such that for some point $x \in \mathbb{R}^n$, some distance function $\delta(p, q)$,

$$C_i = \{x \mid \delta(p_i, x) \leq \delta(p_j, x)\},$$

for all $i, j = 1 \ldots |P|$, $i \neq j$. The union of these Voronoi cells fill space in $\mathbb{R}^n$. The Voronoi diagram is a structure which encodes these cells.

In some circumstances it helps to consider the Voronoi diagram in terms of the partitions between these cells. A bisector b_{ij} is the set of points equidistant from sites p_i, p_j or more formally,

$$b_{ij} = \{x \mid \delta(p_i, x) = \delta(p_j, x)\}\,, \tag{7.1}$$

for all $i \neq j$. A bisector forms an equidistant partition between two Voronoi regions. In a diagram consisting of three or more objects, the diagram bisectors are bounded by intersections with other bisectors. These bounds are identified using the incidence condition: for all distinct i, j, k,

$$b_{ij} \cap b_{jk} = b_{jk} \cap b_{ki}.$$

We can define the bounded bisector as a bisector between two cells i and j, bounded by the bisectors of all other cells k. For all $i, j, k = 1 \ldots |P|$, $i, j \neq k$:

$$\begin{aligned} \bar{b}_{ij} &= \{x \mid \delta(p_i, x) = \delta(p_j, x) \leq \delta(p_k, x)\} \\ &= C_i \cap C_j. \end{aligned}$$

Note that for all $i \neq j$, $\bar{b}_{ij} = \bar{b}_{ji}$, and it is possible for $\bar{b}_{ij} = \emptyset$. Therefore the boundary of a Voronoi cell C_i can be defined by

$$\partial C_i = \bigcup_{i \neq j} \bar{b}_{ij} = C_i \cap \left(\bigcup_{i \neq j} C_j \right).$$

In this application, the Voronoi diagram in $\mathbb{R}^3$ is stored in the form of a set of planar convex polyhedral bisectors. It is easy to show that Voronoi regions which do not have points at infinity are *convex* [Boissonnat and Teillaud, 2007].

Intersections between bisectors form bisecting sub–simplices, for example intersecting polygonal facets meet at an edge or a point. If a number of bisectors intersect at a single point, that point is called a Voronoi vertex.

The Voronoi diagram as a spatial partition can be extended to incorporate the notion of *objects* by defining p_i to be *lines* or *areas*, as is shown in Figure 7.1. While the definition of line and area Voronoi diagrams are simple, their construction is not.

It is important to recognise the special relationship between the Voronoi diagram and the *medial axis*. The medial axis $\mathrm{MA}(\mathcal{S})$ of the boundary $\mathcal{S}$ is the set of points which are *equidistant* to two or more points on $\mathcal{S}$. The medial axis can be thought of as an extension of the Voronoi diagram to infinite sets [Attali et al., 2007].

Voronoi diagram construction will not be discussed here, as numerous existing documents exist which more than adequately cover the topic (see Edelsbrunner [2001], Boissonnat and Teillaud [2007] for good references), in addition to numerous public implementations.

I will use a standard definition of a polygonal mesh $\mathcal{M}$ in $\mathbb{R}^n$ which is $\mathcal{M} = \{P, F\}$ with vertices P and faces F, which is some approximation of a surface $\mathcal{S}$.

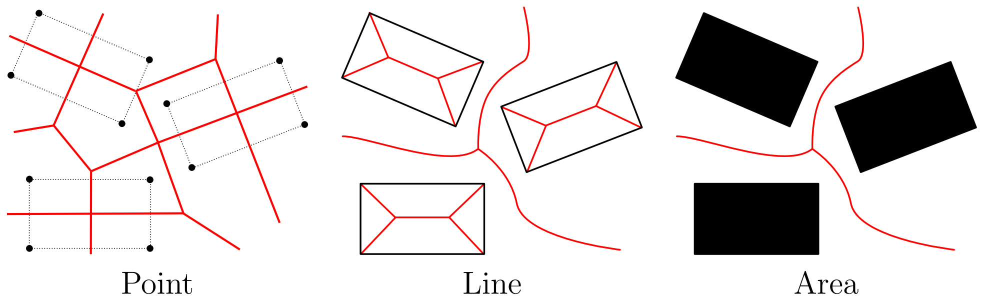


Figure 7.1: A Voronoi diagram can be constructed from different types of input sites. In each case, the bisectors (in red) are defined as per Equation 7.1, using a modified distance function δ which applies to point, line and area primitives respectively. Note that the line and area Voronoi decomposition are very similar.

7.2 Related work

Blum [1967] initially proposed the idea of the medial axis as a skeletal shape descriptor which is applicable to numerous fields of computer graphics, from manufacturing and medical imaging to robot path planning and shape deformation. Numerous techniques have been proposed to approximate this important structure.

The straight skeleton approach of Aichholzer et al. [1995] is an example of a wavefront-based algorithm. Each facet of some manifold $\mathcal{M}$ is grown or shrunk to find the external and internal skeletons respectively. Self-intersections are handled with specialised atomic operations which ensure the topology of the skeleton remains consistent. The resulting skeletons are considerably less complicated that the true medial axis, consisting of no curved segments and relatively few poles and edges. However, the algorithm has yet to be extended to $\mathbb{R}^3$, where the number of special cases for handling self-intersections makes the method unattractive.

Ogniewicz and Ilg [1992] find a medial axis approximation by using a discrete version of the surface derived from a discrete distance function with regularisation coefficients. A similar approach is used by Vleugels and Overmars [1998] to discretise space in order to approximate a medial axis in a voxel grid. While these methods have the advantage of being fast to compute and generalising to any dimension, the connection with the original mesh is all but lost, and the regular sampling of the grid means that the surface may have to be very densely sampled.

Culver et al. [1999] use fourth order algebraic curves to identify the precise medial axis of simple polyhedral objects. The authors use exact arithmetic and as a result the extracted axis is theoretically accurate. However the technique is slow, taking nearly 6 hours to process a polyhedron consisting of 250 faces on a commodity PC.

The reader is referred to Attali et al. [2007] for a survey of methods of medial axis estimation and their stability. I will limit this section to a discussion of methods most relevant to my work, i.e. methods that produce a *simple* skeleton which exploit the relationship between the Voronoi diagram and the medial axis.

The Voronoi diagram serves as a basis for several methods of medial axis extraction. These typically exploit the geometric property that the Voronoi cell of a vertex is ideally long and thin, the longest part parallel to some predicted surface normal at that vertex. The medial axis can be thought of as a "cap" of this long thin structure.

Geiger [1993] define the external Voronoi skeleton for polyhedra in $\mathbb{R}^2$ in a similar manner to the one presented here. The input polygon is first processed to ensure that no vertex of the input encroaches on any segment. Then the skeleton is the subset of bisectors which do not intersect the polyhedron. As I will show in Section 7.5.1 the methods are equivalent in $\mathbb{R}^2$. However, extending this method to surfaces in $\mathbb{R}^3$ significantly increases complexity.

Sheehy et al. [1996] define an algorithm for computing the topologically equivalent medial surface from a large class of boundary representations of solids. Using a specialised Delaunay triangulation to fill a sampling of points on the boundary shape, each tetrahedron is classified according to how many different surface elements touch it. These are then verified according to a non–linear formulation which accurately classifies each tetrahedron according to tangency information from touched surface sites. A medial reconstruction algorithm then walks over these tetrahedra, stitching together a medial surface from the centres of the circumspheres of each tetrahedron.

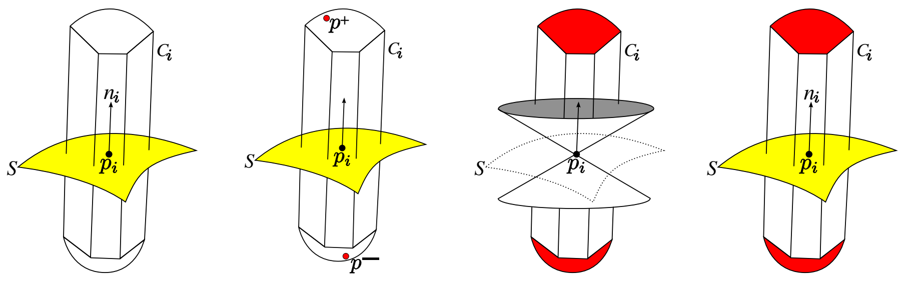
While the computation cost of such an algorithm is particularly expensive, it may also fail. Numerical problems in some tangential cases may cause there to be no solution to the six optimisation cases. In addition, tetrahedra which cannot be classified ("rogue" tetrahedra) can only be effectively classified according to neighbouring tetrahedra. A cluster of rogue tetrahedra cannot be classified. In comparison, I use the Voronoi diagram directly, rather than the results of the Delaunay triangulation. Rogue tetrahedra are analogous to a mesh configuration in which a vertex encroaches on a simplex (see Section 7.6) but is also attached to that simplex. This is geometric property is comparatively simple to identify.

The Power Crust algorithm of Amenta et al. [2001] builds a power Voronoi diagram [Edelsbrunner, 1993], and identifies *poles* for Voronoi cell C_i about point p_i as the furthest points on ∂C_i from p_i, and distinguishes between those that are inside or outside the unknown surface $\mathcal{S}$, marking these as p^- and p^+ respectively.

Faces of the power diagram which separate these poles are output as the *power crust* — a watertight surface[1] fitting the input points. The algorithm is robust enough to deal with issues which are typical in applications utilising data from laser range scanners, such as noise and varying sampling density. Additionally, the surface is guaranteed to be watertight.

A triangulation of the inside poles yields the *power shape* which is serves as an approximation to the medial axis. Unfortunately the power crust algorithm often yields a "spiky", non-manifold structure, which offers no guarantees that the original surface is not intersected by its medial axis.

[1]A watertight surface is closed manifold with no boundary.



(a) (b) Power crust (c) Cocone (d) My method

Figure 7.2: A comparison of medial axis extraction approaches. In (a) the initial configuration is shown, consisting of some surface S in the neighbourhood of a point p_i with n_i a normal to the surface at p_i. The Voronoi cell C_i, associated with p_i is typically elongated in the direction of n_i, a property exploited by most Voronoi based medial axis extraction approaches. In (b), the poles used by the power crust are simply the furthest points on C_i above and below p_i. In (c) a subset of C_i which lies within a cone originating at p_i. In (d), my method uses the subset of C_i which does not intersect surface S.

Yoshizawa et al. [2007] applies this method to connected triangle meshes. The inner poles of Amenta et al. [2001] are used as vertices of a skeleton, while connectivity is inherited from the original mesh. This may lead to degenerate faces, so unwanted poles and degenerate facets are smoothed away yielding a clean skeleton. Relationships with the original manifold are also maintained.

Due to smoothing and cleaning steps which are used to ensure that the generated skeletons are attractive, there are no guarantees that the skeleton is close to the true medial axis. Additionally the method of Yoshizawa et al. [2007] produces a manifold surface, rather than a one-sided non–manifold medial axis. This has the disadvantage that the axis has self–intersections as it encloses almost no volume.

Dey and Zhao [2002] use the methodology of Tight Cocone [Dey and Goswami, 2003] to extract a medial axis approximation as a subset of the Voronoi diagram. The cocone is a two sided cone, centred at a vertex p_i with some associated tolerance angle. The medial axis is defined as the facets of the Voronoi cell C_i associated with p_i whose surface normal lies within the cone. The cocone approach often yields gaps in the medial axis, caused when the surface has sharp angles or saddle points. Additionally there are no guarantees that the extracted axis does not intersect the input surface.

7.3 Voronoi skeleton

In order to define the Voronoi Skeleton, we must first define *Voronoi separability.* Voronoi separability is a property of a closed oriented manifold which effectively gives a lower bound on the density points sampled on the manifold.

Definition A mesh $\mathcal{M}$ is *strongly* Voronoi separable if and only if for all $p_i \in P$, $f \in F$,

$$p_i \notin f \implies \partial C_i \cap f = \emptyset.$$

Strong Voronoi separability implies that the only facets of the mesh $\mathcal{M}$ which intersect the Voronoi cell boundary ∂C_i are those containing the vertex p_i. This means that each edge of the cell boundary ∂C_i may cut the surface $\mathcal{M}$ *at most* once. This allows us to guarantee the successful isolation of the portions of the cell which are above or below the surface, allowing us to extract internal and external skeletons. For extracting the Voronoi skeleton from triangle surfaces in $\mathbb{R}^3$ we need to relax this surface property:

Definition A mesh $\mathcal{M}$ is *weakly* Voronoi separable if and only if for all flat neighbourhoods $\mathcal{N}_i$ about $p_i \in P$,

$$C_i \cap (\mathcal{M} \setminus \mathcal{N}_i) = \emptyset.$$

With this definition, each Voronoi cell C_i is only permitted to intersect the surface $\mathcal{M}$ within some neighbourhood of p_i. The neighbourhood condition is discussed in more detail in Section 7.6.

Once a surface $\mathcal{M}$ is Voronoi separable, a skeleton can be identified as the parts of each Voronoi cell which do not intersect $\mathcal{M}$. Additionally this is restricted to the convex hull of $\mathcal{M}$ in order to avoid the inclusion of points at infinity.

Definition The *Voronoi Skeleton* of a *weak* Voronoi separable, closed, oriented, manifold mesh $\mathcal{M}$ is defined as

$$\mathrm{VS}(\mathcal{M}) = \left(\bigcup_{i,j \notin I} \overline{b}_{ij} \right) \cap \hat{\mathrm{CH}}(\mathcal{M})$$

where

$$I = \left\{ i, j \mid f \in F \implies \overline{b}_{ij} \cap f = \emptyset \right\}$$

is the set of bisector index pairs such the bisector $\overline{b}_{ij}$ does not intersect $\mathcal{M}$, and $\hat{\mathrm{CH}}(\mathcal{M})$ is the region inside the convex hull of $\mathcal{M}$.

The Voronoi skeleton of a closed, oriented mesh $\mathcal{M}$ is the set of bisectors of the Voronoi diagram which lie within the *convex hull* of $\mathcal{M}$ and do not intersect the faces of mesh $\mathcal{M}$. I also classify the *internal* Voronoi Skeleton as the portion of $\mathrm{VS}(\mathcal{M})$ which is inside $\mathcal{M}$, or

$$\mathrm{VS}_{in}(\mathcal{M}) = \mathrm{VS}(\mathcal{M}) \cap \hat{\mathcal{M}}$$

and the *external* Voronoi skeleton is simply

$$\mathrm{VS}_{out}(\mathcal{M}) = \mathrm{VS}(\mathcal{M}) \setminus \hat{\mathcal{M}}$$

where $\hat{\mathcal{M}}$ is the region enclosed by mesh $\mathcal{M}$. This approach is shown graphically in Section 7.4.

By sampling vertices on $\mathcal{M}$, the Voronoi skeleton will eventually converge to the true *medial axis*. Several documents already state this as fact, although I have not yet found a proof, which I include here.

Theorem 7.3.1 *For a mesh $\mathcal{M}$ with distance between adjacent vertices ϵ, as $\max \epsilon \to 0$, $\mathrm{VS}(\mathcal{M}) \to \mathrm{MA}(\mathcal{M})$. Additionally the convergence rate is $\mathcal{O}(\epsilon^2)$.*

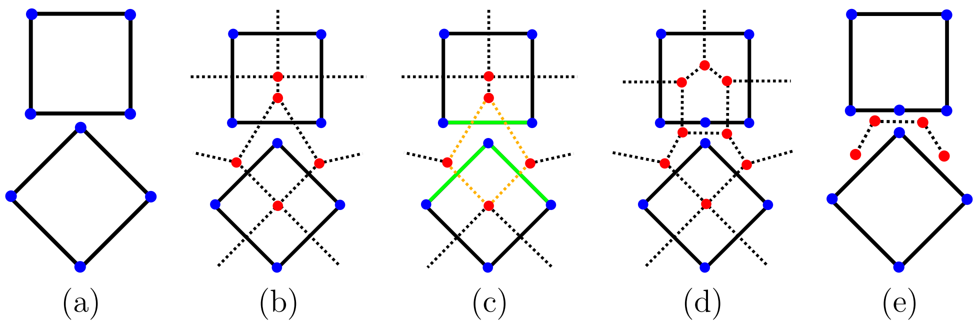


Figure 7.3: In (b), a Voronoi diagram is built of the original manifold (a). In (c), the intersections of a complete cell (marked in orange) with the manifold are marked in green. The green edge not containing the Voronoi vertex of the orange cell is split, and the Voronoi diagram is recomputed in (d). The manifold is now Voronoi separable, and in (e) the external Voronoi Skeleton can be identified.

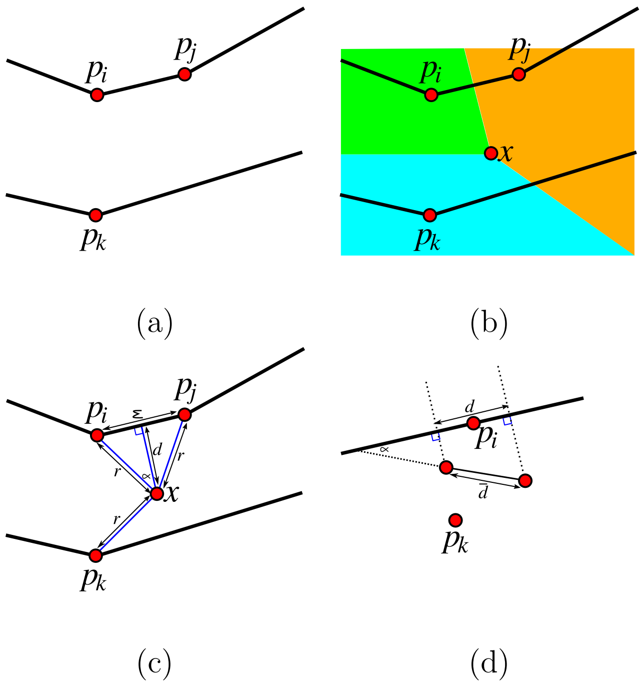


Figure 7.4: A figure for the proof of Theorem 7.3.1.

Proof The proof makes reference to Figure 7.4. Some portion of $\mathcal{M}$ is given in (a). The point of the intersection of the Voronoi cells is shown in x, such that $x = \partial C_i \cap \partial C_j \cap \partial C_k$. Therefore the lengths $|\overline{p_i x}| = |\overline{p_j x}| = |\overline{p_j x}| = r$. I define the length d as the perpendicular distance from x to $\overline{p_i p_j}$. Additionally I define $\alpha = \angle p_i x p_j$.

As $\epsilon \to 0$, then $\alpha \to 0$, implying that $d \to r$. Therefore x is equidistant from p_i and p_k, implying that x is on the medial axis of $\mathcal{M}$.

Since

$$\begin{aligned} d &= r\cos\alpha \\ &\approx r(1-\alpha^2/2), \end{aligned}$$

and as α is $\mathcal{O}(\epsilon)$ as $\epsilon \to 0$, the convergence of d to r (and hence the convergence towards the medial axis) is $\mathcal{O}(\epsilon^2)$.

It is also necessary to show that a regular sampling of $\mathcal{M}$ implies a regular sampling on VS($\mathcal{M}$). Consider the case where $\mathcal{M}$ consists of only two points. Then VS($\mathcal{M}$) is the bisector between these points. Even though there is no way to sample $\mathcal{M}$ more densely, in this case VS($\mathcal{M}$) is *precisely* MA($\mathcal{M}$).

Next we consider the case between a point p_k and a line segment containing vertex p_i. In Figure 7.4(d) the perpendicular bisectors through $\mathcal{M}$ about p_i create a region of $\mathcal{M}$ of width d. Additionally, these perpendiculars intersect the bisector b_{ik} creating a segment of VS($\mathcal{M}$) of length $\overline{d}$. Define the angle α as the angle between b_{ik} and the surface at $\mathcal{M}$. Now $\overline{d} = d\mathrm{cosec}\alpha$. Since $|\alpha| < \pi/2$, we know that $\mathrm{cosec}\alpha \neq \infty$. Therefore as $d \to 0$ (at a rate of $\mathcal{O}(n)$), so does $\overline{d}$. In the case between two line segments, the same rule applies. ▮

7.4 External skeleton

The external skeleton is frequently used in Chapter 8 in order to identify a means to collapse an input shape to a shape that is genus 0. In Figure 7.5 I present a method to do this which is compatible with all methods for computing the medial axis.

A *good* external skeleton for this application is one which ensures that after a full space Delaunay triangulation no simplices on the exterior of the input surface $\mathcal{M}$ connect only vertices on the input surface, and it must not intersect $\mathcal{M}$. In this application it is not necessary to trim the Voronoi Skeleton against the manifold $\mathcal{M}$ — intersected bisectors are simply discarded to reduce computational complexity.

The methods of Amenta et al. [2001], Dey and Zhao [2002] and Yoshizawa et al. [2007] cannot guarantee that the skeleton does not intersect $\mathcal{M}$. Additionally the method of Dey and Zhao [2002] is not watertight. The Voronoi skeleton will not, *by definition* intersect $\mathcal{M}$. Additionally the Voronoi skeleton is a watertight structure.

7.5 Ensuring Voronoi separability

Theoretically any polyhedron $\mathcal{M}$ can be made Voronoi separable simply by sampling infinitely many points on the faces of $\mathcal{M}$. As the density of points in $\mathcal{M} \to \infty$, then at point p_i, with neighbouring points p_j and p_k, the bisectors $\overline{b}_{ij} \to \overline{b}_{ik}$. The remaining bisectors of C_i tend to a single point, which is a point on MA($\mathcal{M}$). By definition, the medial axis cannot intersect the mesh $\mathcal{M}$, so the mesh is Voronoi separable.

This result gives reasonable cause to use an iterative approach to finding the Voronoi skeleton. We need to find a minimum error bound to guarantee Voronoi separability. In Figure 7.6 p_i, p_j and p_k are points on some manifold $\mathcal{M}$, a is the the length of $\overline{p_j p_k}$, and b is the minimum distance from p_i to $\overline{p_j p_k}$. As the manifold is subdivided, b never changes, while a will get progressively smaller.

This yields a solution which is unsatisfactory, as this may require the mesh to be densely sampled. Another solution to the example given in Figure 7.6 would be to split

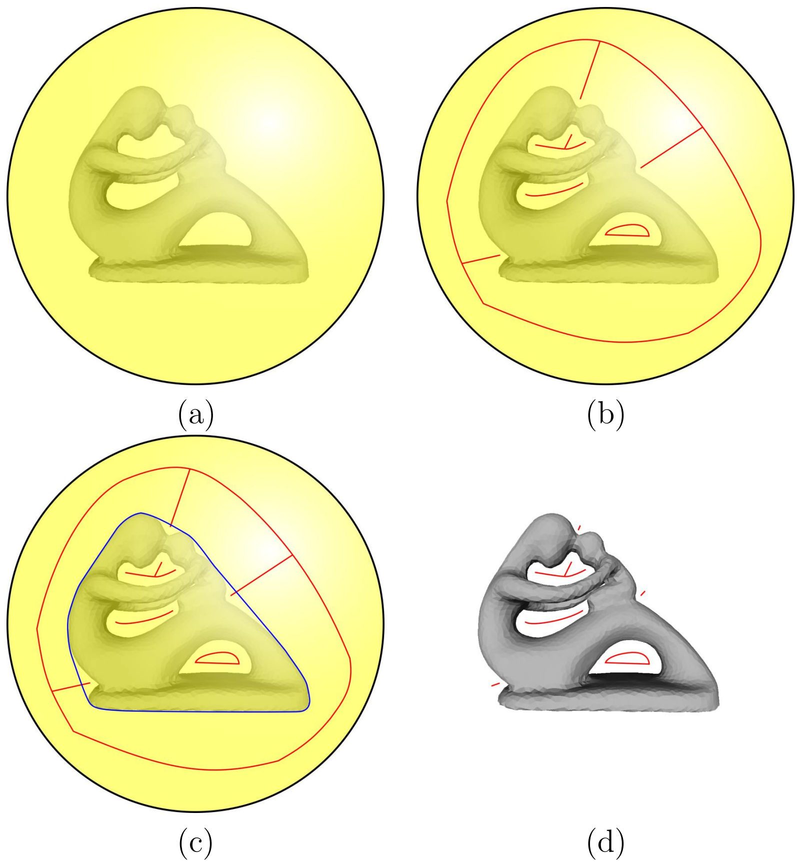


Figure 7.5: An algorithm for finding the external skeleton. In (a), the input polyhedron is wrapped in a sphere. In (b), the medial axis (marked in red) is found of the entire shape. In (c), the convex hull of the input shape is found, which is then used as a clipping shape against the external axis to yield the external skeleton in (d).

$\overline{p_j p_k}$ at the point at a point such that b is minimised. This will ensure that $\overline{p_j p_k}$ is not cut by the Voronoi cell C_i, because the newly created bisector will be *parallel* to $\overline{p_j p_k}$. I devise an iterative algorithm based on this method to enforce Voronoi separability in input manifolds. A full example of this is shown in Figure 7.3.

7.5.1 Constrained Delaunay triangulation

The Voronoi diagram of a set of points is the dual of the Delaunay triangulation. It is also possible to show in $\mathbb{R}^2$ that Voronoi separability implies that no vertex $p_i \in P$ *encroaches* on a segment $f \in F$. This approach was first used by Geiger [1993].

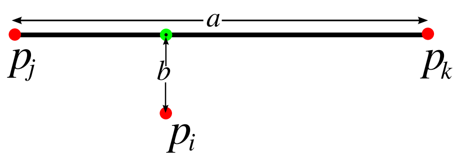


Figure 7.6: Convergence towards Voronoi separability.

As a reminder, a vertex p_i encroaches on a segment x if it lies within the diametrical circle of x. If no vertex p_i encroaches on any segment in F then the mesh has a Delaunay triangulation that *conforms* with F (see Section 5.1.1).

Lemma 7.5.1 *A manifold $\mathcal{M} = \{P, F\}$ in $\mathbb{R}^2$ is strongly Voronoi separable if no vertex $p_i \in P$* encroaches *on any segment in F.*

Proof This simple proof references Figure 7.7. In (a), vertex p_k encroaches on segment $\overline{p_i p_j}$. In (b), we construct triangle $\triangle p_i p_j p_k$. As $\angle p_i p_k p_j$ is 90°if p_k lies on the circle (from *Thale's theorem*[Page, 2008]), $\triangle p_i p_j p_k$ must be obtuse. The circumcentre of an obtuse triangle lies outside of the triangle. As the bisectors b_{ik} and b_{jk} intersect the segment $\overline{p_i p_j}$ we can state that the surface is not Voronoi separable. ▮

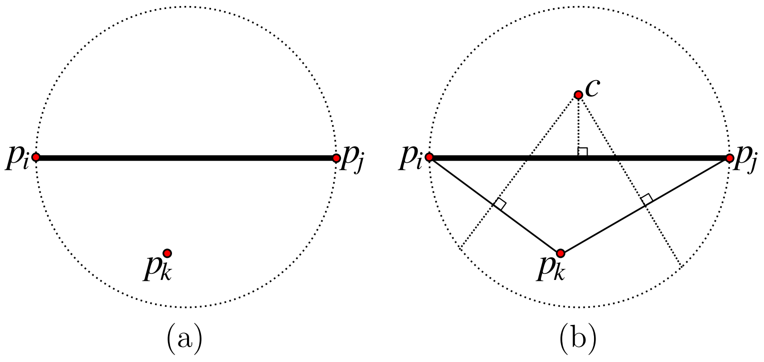


Figure 7.7: The relationship between vertex to segment encroachment and Voronoi separability.

Recall from Section 7.3 that by splitting an input segment at a point a minimum distance from the Voronoi cell vertex, the newly generated bisector will not intersect the input segment. *This gives us a new, optimal location at which to split constraints.* In Figure 7.8 the simplest case of encroaching vertices is shown.

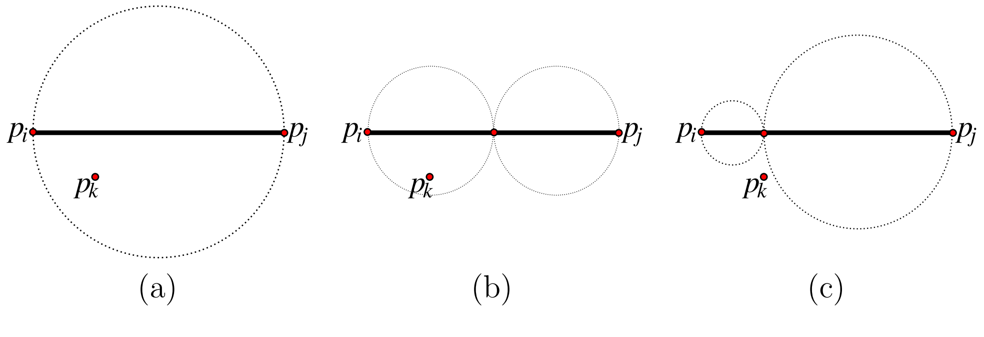


Figure 7.8: Different splitting strategies for encroaching vertices. In (b) the edge $\overline{p_i p_j}$ is split at the midpoint. The vertex p_k still encroaches on one of the new line segments. In (c) the edge $\overline{p_i p_j}$ is split at a point a minimum distance to p_k instead. It is clear that with the splitting strategy in (c), p_k will never encroach on the input segment.

Lemma 7.5.1 provides a necessary condition for Voronoi separability. However it does not extend naturally to $\mathbb{R}^3$. This will be discussed in Section 7.6.

Given a mesh $\mathcal{M} = \{P, F\}$:

```
while (true)
    foreach ($f \in F$)
        foreach ($p_i \in P$)
            if ($p_i$ encroaches on $f$)
                Find the barycentric coordinates $\beta$
                    of the orthogonal projection from $p_i$ to $f$
                Append the tuple $[\beta, f]$ to splits
            end if
        end foreach
    end foreach
    if ($|split| \neq 0$)
        Apply splits to $\mathcal{M}$
    else
        break
    end if
end while
```

Figure 7.9: An algorithm for converting a polygon $\mathcal{M}$ into one which is Voronoi separable by vertex insertion.

7.5.2 An algorithm for Voronoi separability in 2D

An algorithm for enforcing Voronoi Separability is given in Figure 7.9. Note that *splits* is a list of tuples representing a vertex we would like inserted into the manifold. It consists of a simplex which should be split, and the barycentric coordinates associated with each vertex of that simplex. This approach is described in Chapter 6.

In Figure 7.10 we see a common problem which arises from the algorithm in Figure 7.9. If a new point is inserted at an orthogonal projection from the vertex p_i it forms a right angled triangle with the point at the tip. As the edge bisectors of a right angled triangle intersect on the hypotenuse we can be sure that a newly created Voronoi vertex will lie exactly on the manifold $\mathcal{M}$.

To deal with this situation, we flag Voronoi vertices which we know to be on $\mathcal{M}$ identified in this manner, and deal with them as being both inside and outside the manifold, avoiding the computation of intersections with any bisectors which include this Voronoi vertex. This avoids any numerical instabilities when computing the skeleton.

In this section I have shown that for polyhedra in $\mathbb{R}^2$, inserting points in an encroached segment at the closest point on the encroached segment to the encroaching vertex is more likely to converge than placing the new vertex at the midpoint. This simple modification to the algorithm of Ruppert [1995] will improve the convergence rate towards a weak C_FDT.

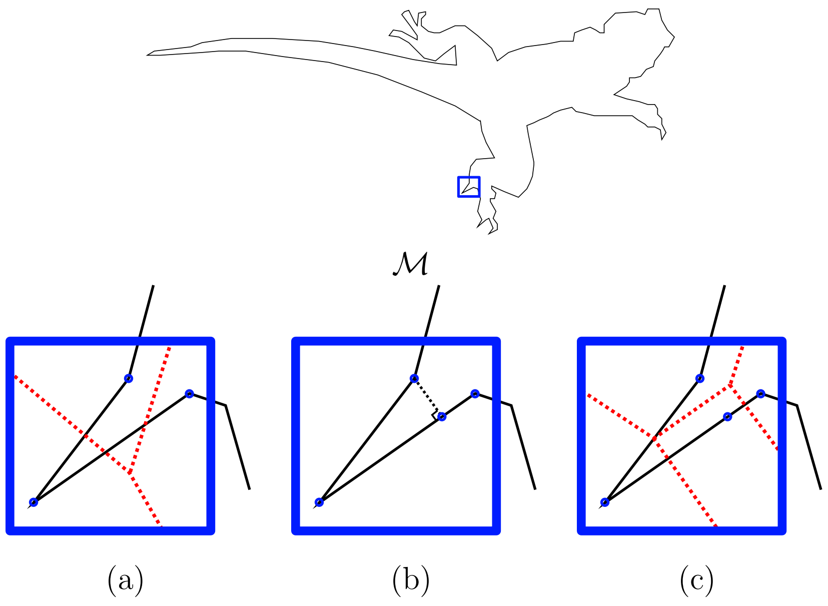


Figure 7.10: An example of numerical instabilities resulting from using the closest point insertion algorithm to enforce Voronoi separability. The Voronoi cell bisectors of a portion of $\mathcal{M}$ are shown in (a). As a non-local edge is intersected by the bisector, $\mathcal{M}$ is not Voronoi separable. A new vertex is introduced on $\mathcal{M}$ at a minimum distance to the affected vertex in (c), but the resulting Voronoi diagram results in a Voronoi vertex lying **on** the manifold $\mathcal{M}$.

7.6 Voronoi separability in 3D

The Voronoi diagram of a triangle mesh $\mathcal{M}$ in $\mathbb{R}^3$ can be loosely described as having two degrees of freedom — it may vary between local sub–surfaces of $\mathcal{M}$, or it may vary within a local sub–surface of $\mathcal{M}$ projected into $\mathbb{R}^2$. Decoupling these two degrees of freedom is central to finding the Voronoi skeleton.

7.6.1 Limitations of strong Voronoi separability

Unfortunately the conditions for strong Voronoi separability in Section 7.5 are particularly difficult to meet on triangle meshes in $\mathbb{R}^3$. Clearly a cell C_i about a point p_i would not be strongly Voronoi separable if a facet containing p_i was *obtuse angled.* This situation can be simply visualised in Figure 7.11, where two neighbouring obtuse angled triangles are split according to the nearest point splitting method. It is clear that this example will not terminate, and that the only way to guarantee strong Voronoi separability on triangle meshes is to restrict them to having *no* obtuse angled triangles!

Clearly this is unsatisfactory, as these mesh configurations are often acceptable for extracting the Voronoi skeleton. The distinction between acceptable and undesirable mesh configurations is shown in Figure 7.12. The cell C_i in Figure 7.12(a) intersects a face which does not contain p_i,

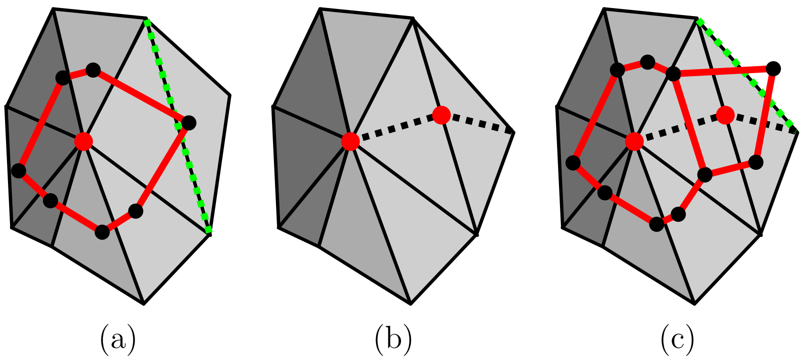


Figure 7.11: In (a) the cross section of a Voronoi cell (in red) cuts an edge (in green). That edge is split in (b), but as the split triangle was isosceles, splitting it results in further obtuse angled triangles, which will cut further edges in (c).

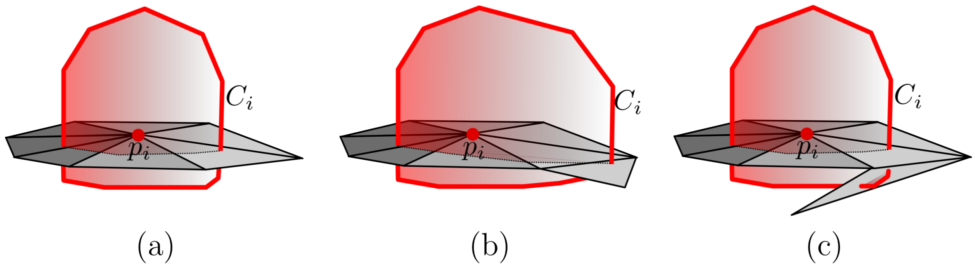


Figure 7.12: This figure shows a side view of a cell (in red) intersecting a surface. In (a) the cell only intersects facets including p_i so it is *strongly* Voronoi separable. In (b) the cell intersects a facet outside of the one–ring neighbourhood of p_i, making it *weakly* Voronoi separable. In (c) the cell intersects a neighbouring facet in an illegal way.

7.6.2 Encroaching segments in 3D

On triangle meshes, a vertex p_i may encroach on a facet f as it does in the 2D case. The modified rules for edge and face encroachment in $\mathbb{R}^3$ are given in Figure 7.13.

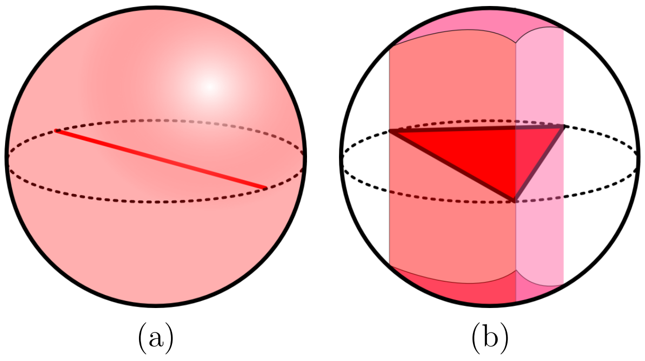


Figure 7.13: Edge and face encroachment rules in $\mathbb{R}^3$. In (a) a vertex encroaches on the edge if it lies within its diametrical sphere. In (b), a vertex encroaches on the facet if it lies within its minimum equatorial sphere — the minimum sphere which contains the circumcircle through the points of the face — and it lies above or below the face. I refer to these two rules as edge and face encroachment respectively. These rules are adapted from Shewchuk [1997] in the context of Constrained Delaunay mesh refinement.

In this figure, I distinguish between edge encroachment, where p_i lies within the minimum circumsphere through an edge of f, and face encroachment, where p_i lies within the minimum circumsphere of f and above or below the facet. The results of an encroachment test in $\mathbb{R}^3$ have important implications when determining the encroachment type. These cases are labelled as follows:

1. If p_i fails both encroachment tests, then its Voronoi cell C_i does not intersect f in any way.

2. If p_i encroaches on an edge of f but does not face encroach f, the vertex "pushes" its Voronoi cell C_i across the given edge.

3. If p_i encroaches on the face f but does not edge encroach on any edges of f, the Voronoi cell C_i intersects f *without intersecting any of its edges.*

4. If p_i both encroaches on both the edge and face f, then C_i is pushed across an edge into f.

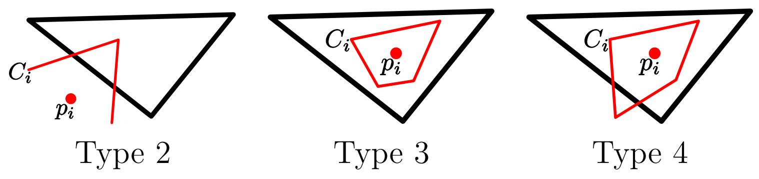


Figure 7.14: Distinguishing between different types of encroachment.

Using this encroachment machinery, we can distinguish between Voronoi separability *in the plane* and Voronoi separability *between planes.* If no vertex of $\mathcal{M}$ encroaches on any edge or face of $\mathcal{M}$ it is *strongly* Voronoi separable. If p_i encroaches on an edge *which is in some neighbourhood of* p_i but not the face, it is locally *weakly* Voronoi separable. If p_i encroaches on any facet of $\mathcal{M}$, i.e. is of Type 3 or Type 4 it is not Voronoi separable.

With this distinction we can state a simple rule for defining Voronoi separability on triangle meshes. Weak Voronoi separability can be ensured locally by *ignoring the edge encroach condition* in some neighbourhood of p_i. This represents the main distinction between my method and other methods for constructing Conforming Delaunay triangulations.

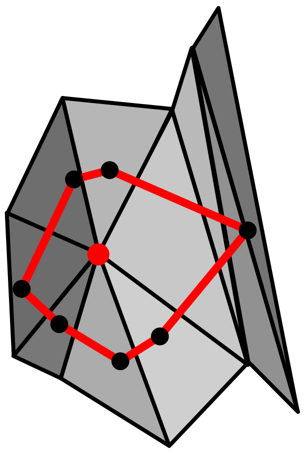


Figure 7.15: In some degenerate cases, such as the example depicted here, a neighbourhood may be more difficult to define.

Defining the correct neighbourhood to ignore the edge encroach condition can be problematic, as shown in Figure 7.15. In this case, it would be prudent to skip edge encroachment tests for all edges which are intersected by the contour of C_i. However if these edges represent undulating or sharp angled facets, these edges may need to be split as a portion of the skeleton may be missing. For this application, I skip encroachment tests for edges belonging to triangles in the one-ring of v_i, which has worked well in practise.

The conditions in Figure 7.13 are stricter than the “diametrical lemon” encroachment rule of Shewchuk [1997] in the context of constrained Delaunay triangulation, which allows input surfaces to have surface angles of 45° and sharper.

Shewchuk's rule allows conforming Delaunay triangulations to be constructed, without them necessarily being Voronoi separable. This is because even though some Voronoi cell C_k may intersect a constrained edge $\overline{p_ip_j}$, the Voronoi diagram may still contain portion of bisector b_{ij}, which guarantees that in the Delaunay triangulation edge $\overline{p_ip_j}$ will be preserved.

7.6.3 Operation ordering

In deciding which split operations to perform, I use the same approach as Si and Gaertner [2005]. Split each encroached facet f according to the encroaching vertex p_i which is closest to f. This is the equivalent of maximising the circumsphere through f and p_i.

7.6.4 Cells to infinity

In the algorithm in Figure 7.9 Voronoi cells to infinity are typically discarded as intersections with the cell are difficult to compute. If a mesh is sufficiently densely sampled these will not arise, but they *do* arise in sparsely sampled meshes with sharp corners (see Figure 7.16(a)). I have found equivalent configurations to be quite common in triangular meshes.

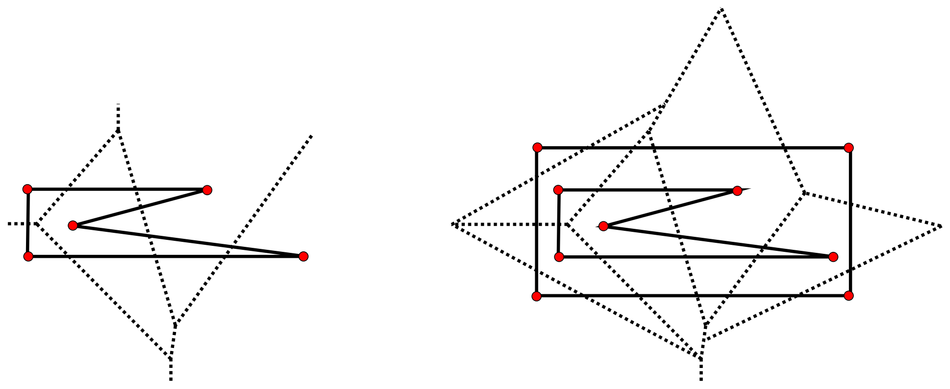


Figure 7.16: Dealing with cells to infinity. In the example in (a), several vertices have Voronoi cells which have vertices at infinity. This will leave a portion of the skeleton unresolved in the concave region. In (b) the input manifold is *wrapped* in a simple bounding box, which guarantees that none of the Voronoi cells associated with any of the original manifold vertices have corners at infinity. This allows us to identify concave regions with the skeleton in this example.

In Figure 7.16(b) I show a simple method by which the input manifold is first *wrapped* in some bounding shape first. This guarantees that each of the original vertices will have a bounded Voronoi cell.

7.6.5 Performance analysis

A full complexity analysis of this algorithm is difficult given that the number of splits that need to be applied and the number of steps required before termination depends on the models used. Experience has shown that models with particularly thin shells with a low polygon density may take several iterations before termination is reached.

All $\mathbb{R}^2$ contours tested take only a single iteration to converge. Most models in $\mathbb{R}^3$ (the *bunny* models for example) take three steps to converge. The worst case complexity is $\mathcal{O}(n)$ in the number of vertices. I have found the number of split operations that need to be performed decreases *logarithmically* at each step, implying that the average case complexity is $\Theta(n \log n)$.

The most time consuming operation in the above algorithm is the computation of the diametrical circle. In $\mathbb{R}^3$ the sphere centre p_c is given by $p_c = \alpha p_1 + \beta p_2 + \gamma p_3$ where

$$\begin{aligned}
\alpha &= \frac{|p_2 - p_3|^2 (p_1 - p_2) \cdot (p_1 - p_3)}{2|(p_1 - p_2) \times (p_2 - p_3)|^2} \\
\beta &= \frac{|p_1 - p_3|^2 (p_2 - p_1) \cdot (p_2 - p_3)}{2|(p_1 - p_2) \times (p_2 - p_3)|^2} \\
\gamma &= \frac{|p_1 - p_2|^2 (p_3 - p_1) \cdot (p_3 - p_2)}{2|(p_1 - p_2) \times (p_2 - p_3)|^2} \qquad (7.2)
\end{aligned}$$

In the general dimension the centre of the smallest circum-hypersphere through the vertices of the $(n-1)$-simplex s with n corners and normal $\mathbf{n}$ is the solution to the system $N\mathbf{x} = \mathbf{d}$ where

$$N = \begin{bmatrix} 2(p_1 - p_2) \\ \vdots \\ 2(p_1 - p_n) \\ \mathbf{n} \end{bmatrix} \quad \text{and} \quad \mathbf{d} = \begin{bmatrix} (p_1 - p_2) \cdot (p_1 + p_2) \\ \vdots \\ (p_1 - p_n) \cdot (p_1 + p_n) \\ \mathbf{n} \cdot p_1 \end{bmatrix}$$

Although a point p_i has no directional component, it is be treated as a vector for the purposes of the above calculations. This formulation gives the intersection of $(n-1)$ perpendicular bisectors of the simplex and the hyperplane through the simplex.

Note that with this method the Voronoi diagram of the input vertices only needs to be computed *once* like the approach of Dey and Zhao [2002], and unlike the approach of Amenta et al. [2001], which requires the computation of the Voronoi diagram twice.

7.6.6 Implementation

The skeleton extraction algorithm has been implemented in $\mathbb{R}^2$ and $\mathbb{R}^3$, although it theoretically scales to oriented manifolds in any dimension. This implementation makes use of *QHull* of Barber et al. [1996] for finding the Voronoi decomposition of space.

Initially a dihedral angle test is performed on $\mathcal{M}$ to ascertain if the input geometry has no surface angles sharper than 45°. If this test fails, the Voronoi skeleton which is extracted with this algorithm may have missing facets near these sharp corners. An alternative skeleton method such as *powercrust* or *tcocone* may be used, although there are no guarantees that these methods will also succeed. It should be noted that a corner cutting approach (such as subdivision) can be used to increase surface angles of meshes which fail this test.

Should it pass this test, $\mathcal{M}$ is made Voronoi separable as follows:

1. The model is wrapped in a bounding cube.

2. Encroachment spheres are computed for each facet and edge.

3. Each vertex is tested for encroachment.

4. Splitting operations are created and applied to $\mathcal{M}$ if necessary, using the barycentric refinement method described in Chapter 6.

5. If splitting operations were applied, goto step 2.

It took 80.5 seconds over 2 iterations to make the 5000 vertex *fertility* model shown in Figure 7.17 Voronoi separable. Due to the dihedral angle limitation, the Voronoi separability algorithm may not terminate. In these circumstances the number of iterations is limited to ensure termination.

Once the surface is Voronoi separable, the Voronoi skeleton must be extracted. This process is performed geometrically. First, the Voronoi diagram is built from the vertices of $\mathcal{M}$. Then for each p_i and its associated C_i, the facets of C_i are tested for intersection against facets of $\mathcal{M}$ in the vicinity of p_i. Intersecting facets of C_i are discarded. This process takes several seconds.

If the mesh is not Voronoi separable, the algorithm will not terminate. For this reason, I limit the number of splitting iterations to 4 steps. The resulting skeleton may have missing faces in areas where the surface represents a very thin shape, or where the corners are sufficiently sharp. In most circumstances this skeleton is still usable.

The implementation in $\mathbb{R}^3$ takes the form of a number of small self contained executables:

- *wrap* encloses an input mesh in either a cube or a sphere of user specified density.

- *test* tests the dihedral angle limitation of the input mesh.

- *skel* builds the Voronoi skeleton from an input mesh using the method described in Section 7.6.

- *trim* accepts an input mesh and the Voronoi skeleton which is output from *skel*, and extracts the external skeleton as described in Section 7.4.

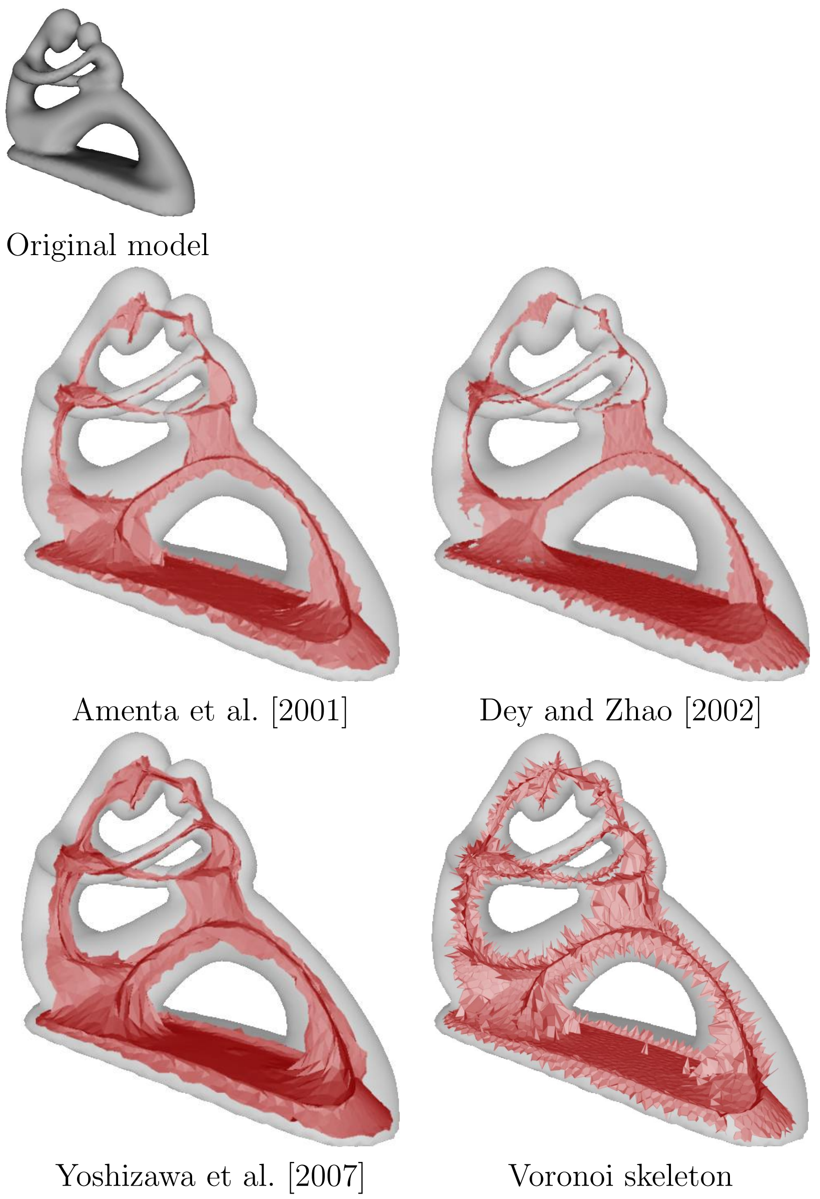
Given an input manifold $\mathcal{M}$, an execution script for extracting the external skeleton may proceed as follows:

1. If the external skeleton is required, $\mathcal{M}$ is initially wrapped in a sphere, the result is called $\mathcal{W}$.

2. *test*($\mathcal{W}$) then determines if the mesh is suitable for Voronoi skeleton extraction. If it is successful, $\mathcal{S}$=*skel*($\mathcal{W}$) returns the skeleton, else *powercrust* or *tcocone* is used.

3. The external skeleton is trimmed against its convex hull using the *trim* function.

Both *powercrust* and *tcocone* have been used as replacement approaches to identify the external skeleton. *tcocone* is used as a default approach as it has performed well in my experiments. Its limitations are explored in Figure 7.20.



Amenta et al. [2001]	Dey and Zhao [2002]	Yoshizawa et al. [2007]	Voronoi skeleton
Non-manifold	Non-manifold	Manifold	Non-manifold
Voronoi poles	Voronoi subset	Voronoi poles	Voronoi subset
Watertight	Not watertight	Self-intersecting	Watertight

Figure 7.17: A comparison of the medial axis computation techniques based on the Voronoi diagram. The method of Yoshizawa et al. [2007] differs from the others in that the resultant skeleton is a surface, which will often self-intersect. Notice also that there are several holes in the skeleton resulting from the method of Dey and Zhao [2002] due to the specified angle of the cone. Increasing this angle may result in unwanted facets.

7.7 Results

Three 2D examples of extracting the internal and external Voronoi Skeletons are shown in Figure 7.18. The ring model is initially Voronoi separable, and requires no simplex splits to extract the Voronoi Skeleton. The blob model requires 8 split operations before convergence. Convergence also occurs in only one iteration. The lizard model requires 24 split operations to converge. Convergence takes a single iteration in each case.

In Figure 7.17 a 3D skeleton is extracted from the *fertility* model using four different approaches. Each of the methods of Amenta et al. [2001], Dey and Zhao [2002] and Yoshizawa et al. [2007] take no more than 10 seconds to find the skeleton, while the Voronoi skeleton takes 4 minutes for this 5000 vertex model. The performance discrepancy can be attributed to an unoptimised implementation, which is heavily dependent on slow geometric operations to extract the Voronoi skeleton from the Voronoi diagram. I believe that an order of magnitude performance improvement is possible with a more efficient implementation.

In Figure 7.19 we show an application of the external Voronoi Skeleton. In this case, correspondences between Voronoi vertices on the skeleton and the vertices of $\mathcal{M}$ are easily deduced from overlapping cells. This is possible if the Voronoi skeleton is complete, as a subset of the Voronoi cell C_i is present for each vertex p_i.

In Figure 7.20 a comparison between the Voronoi skeleton approach and the method of Dey and Zhao [2002] is shown. Both methods identify a subset of the Voronoi diagram as the skeleton, but the Voronoi skeleton approach will also include facets which do not have a normal component within the cocone. This explains why Tight Cocone fails to identify facets between these correctly oriented facets, leading to holes. For this reason, the Voronoi skeleton provides a *more complete* skeleton, should the surface meet the angle requirements. Note that while the tolerance angle of the cocone can be arbitrarily increased, in this case it would result in unwanted facets being included which intersect the mesh.

7.8 Summary

The Voronoi skeleton is a new medial structure for closed surfaces. The skeleton of a manifold $\mathcal{M}$ consists of facets extracted from the Voronoi diagram of a Voronoi separable manifold which do not intersect $\mathcal{M}$. With Theorem 7.3.1 I show that should the surface of $\mathcal{M}$ be subsampled indefinitely, the Voronoi skeleton itself will converge to the true medial axis.

With Lemma 7.5.1 I prove that the Voronoi separability property defined in Section 7.3 is the equivalent to Ruppert's encroachment property in the context of constrained Delaunay triangulation. This significantly reduces the complexity of converting input manifolds to ones which are Voronoi separable. The Voronoi skeleton is a fast and effective method to find the skeleton of manifold contours in $\mathbb{R}^2$.

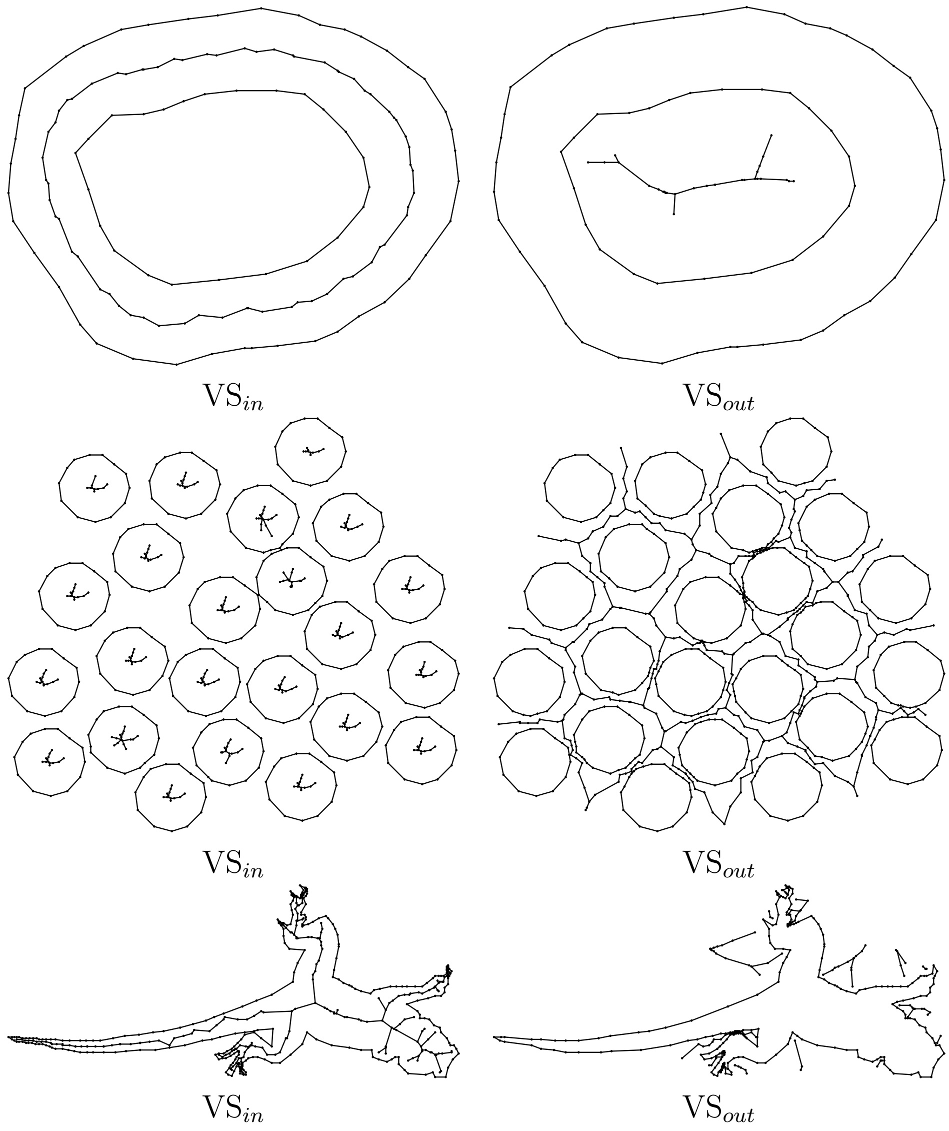


Figure 7.18: The Voronoi Skeleton extraction (internal and external) for three contours of increasing complexity.

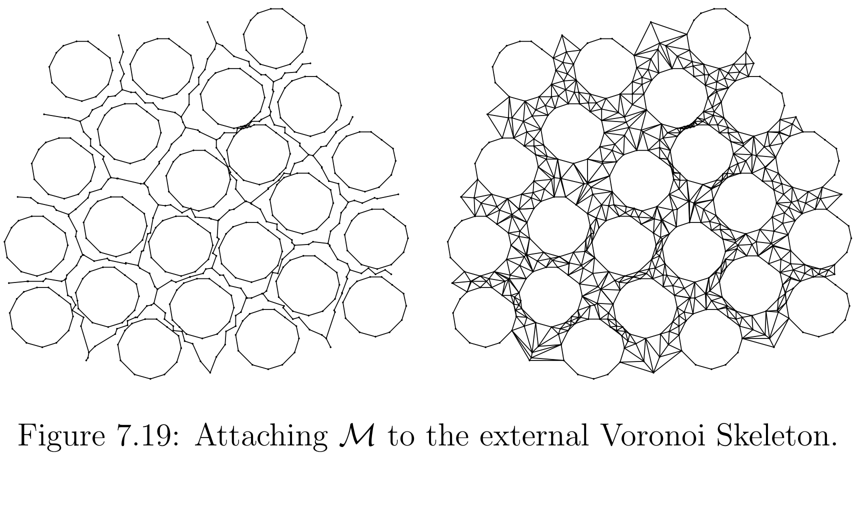


Figure 7.19: Attaching $\mathcal{M}$ to the external Voronoi Skeleton.

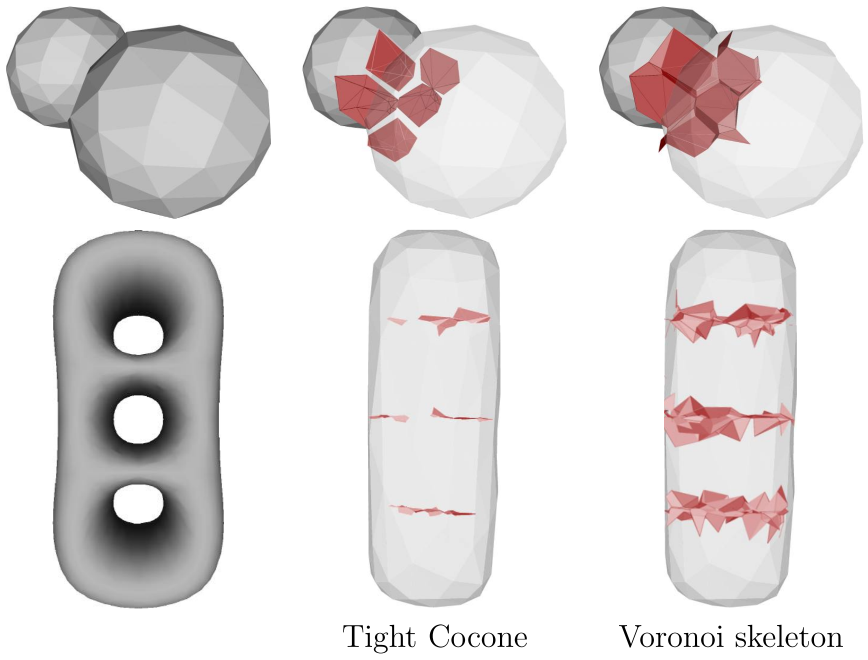


Figure 7.20: In this figure, my method of external skeleton extraction is compared with the method of Dey and Zhao [2002] on a model consisting of two spheres and the three holed torus.

For surfaces in $\mathbb{R}^3$ I have shown that the edge encroachment tests can be ignored in some cases for determining if a mesh is weak Voronoi separable, which allows meshes to be constructed with obtuse angled triangles. Unfortunately there is sharpness limitation between facets of $\pi/2$, which limits the completeness of the skeleton due to the fact that the algorithm may not terminate.

For this work I make use of the *fertility* model and the three holed torus models, which are taken from the AIM@SHAPE model repository (`http://www.aimatshape.net`).

Chapter 8

Morphing between contours

Geometric morphing between arbitrary contours is a complex but well studied problem. Given two input surface meshes $\mathcal{S}$ and $\mathcal{T}$, find a sequence of intermediate meshes such that the transition from $\mathcal{S}$ to $\mathcal{T}$ appears plausible.

The relationship between $\mathcal{S}$ and $\mathcal{T}$ is used to make this problem considerably simpler. In an increasing order of difficulty, here are some *types of correspondences*:

1. $\mathcal{S}$ and $\mathcal{T}$ are closed surfaces of topological genus 0 with the same polygon count and an exact and known correspondence between the vertices of the two meshes.

2. $\mathcal{S}$ and $\mathcal{T}$ of Type 1 but with no polygon / vertex correspondences.

3. $\mathcal{S}$ and $\mathcal{T}$ of Type 2 but are of the *same* topological genus that is greater than 0.

4. $\mathcal{S}$ and $\mathcal{T}$ are closed surfaces of differing topological genus.

5. $\mathcal{S}$ and $\mathcal{T}$ are open manifolds of differing topological genus.

The problem of generating an animated sequence between the two input meshes of Type 2 has been exhaustively studied. Most work in this field has been targeted at converting meshes from Type 2 to Type 1 by first allowing the user to define a small number of correspondences, and then to remesh one (or both) of the input meshes $\mathcal{S}$ and $\mathcal{T}$ to have exact vertex correspondences according to a mapping. Similarly Type 3 models are mapped to the same parametric domain to convert them to Type 1.

Once $\mathcal{S}$ and $\mathcal{T}$ are of Type 1, morphing between the two sequences is performed by using some time dependent function to transfer from the vertex locations of $\mathcal{S}$ to the vertex locations of $\mathcal{T}$.

This mapping approach has the advantage that once in the form of a Type 1 correspondence, the animator has a great deal of control over the path, speed and shape characteristics of the morph sequence. However, it is inefficient to convert from a Type 2 or 3 correspondence to a Type 1 correspondence of surface models for two reasons:

- It is slow to remesh one or both models, and

- Some error will be incurred when the meshes are reconstructed.

One goal is to design a morphing technique which would be able to morph between models of Types 2 and 3 without having to convert first to Type 1. The challenge is in the smooth introduction of polygons during the animation sequence, and in the relative smoothness of the vertex transitions.

I will present a method to construct a morph sequence from input of Type 4, of which all of the previous types are a subset. While it is possible to convert from models of Type 5 to Type 4 by making open mesh portions into a two sided manifold, my method will typically fail on models of Type 5.

A fairly recent survey of mesh morphing techniques is available by Alexa [2002].

8.1 Related work

My method draws on work in the field of surface reconstruction from planar cross-sections. It is important in medical imaging as it allows three dimensional models to be reconstructed from images generated by X-Ray or MRI data [Bajaj et al., 1996]. It is also applicable to finite element methods.

8.1.1 Morphing between surfaces in 2D

There are several methods which, like that of Bajaj et al. [1996] and Barequet et al. [2003], construct the correspondences between contours $\mathbb{R}^2$ by constructing some ad-hoc (non-Delaunay) higher dimensional triangulation. As discussed in Section 5.4.1, these methods fail in the higher dimensional context due to ambiguities relating to orientation.

My method bears a significant resemblance to the methods of Boissonnat [1988] and Geiger [1993] in that the solution to their planar cross section problem is constrained Delaunay and therefore unique. This method encounters some similar problems to the method presented here. These problems [Bajaj et al., 1996] are:

- Many methods for connecting planar cross-sections require effective *tiling* methods to create quality triangulations between contours. Delaunay based methods of contour connection tiling typically yield good quality meshes, although methods to enforce better triangulation quality can be employed.

- The correspondences between segments or regions of input contours are difficult to automatically deduce, as multiple valid configurations can occur. I will rely on the user specified rigid body alignment to achieve the correct correspondence.

- If $\mathcal{S}$ consists of two or more components and $\mathcal{T}$ is a single component enclosing $\mathcal{S}$, then the animation manifold must encode some *branching* behaviour to simulate the splitting of the object(s). Bajaj et al. [1996] and Geiger [1993] use a skeleton as an approximation to the medial axis as a means to define a correct edge from which to branch. I use a similar approach.

A fundamental difference between using planar cross-sections for animation as opposed to surface reconstruction is that correspondences in surface reconstruction are often automatically deduced because surface slices are aligned and calibrated automatically. With animation the correspondences between input surfaces must be customisable by the animator.

8.1.2 Morphing between surfaces in 3D

There are numerous methods for defining morph sequences between topologically equivalent shapes, but very few are capable of dealing with shapes in $\mathbb{R}^3$ of differing topology. These include volumetric techniques, best represented by the distance field method of Cohen-Or et al. [1998], implicit surface methods [Desbrun and Cani, 1998, Turk and O'Brien, 1999], level set surfaces Breen and Whitaker [2001] or the Cartesian product projection method of Klimmek et al. [2007].

Volumetric methods can ignore the restrictions of surface topology inherent in manifold transition problems by instead using a volumetric representation of the interior of the surface. A raft of complex topological operations are available once in this format, giving a user enormous flexibility when defining a transition between shapes.

All volumetric techniques suffer from two main restrictions:

- The input surfaces must first be approximated by a volumetric representation incurring some error in the surface representation. Typically very sharp corners are difficult to represent accurately in a volumetric representation.
- Automated approaches may cause disconnected pieces of geometry to "appear" and "disappear" during the morph sequence.

Like volumetric representations, implicit surface techniques incur some bounded error when converting a triangle mesh input to an implicit representation. Desbrun and Cani [1998] allow for shape metamorphosis using implicit surfaces as a particle coating for a collection of morphing particles, and bears a significant similarity with volumetric methods.

Turk and O'Brien [1999] use the space-time paradigm to define a smooth morph sequence between input shapes. The variational implicit function in $\mathbb{R}^4$ bears a significant resemblance to the animation manifold introduced in Chapter 1, in that it is a space time representation encoding the entire morph sequence within a single shape. The advantage of a variational shape interpolation method is that the transformations are *smooth* rather than *linear* (which they are in our case) and complex topological transitions can be automatically identified.

The implicit surface representation of Turk and O'Brien [1999] itself is not easily modified — once the system of linear equations has been solved for the input contours, the representation is in the form of a number of coefficients of appropriate radial basis functions. This must be solved again should some coefficients change. The user has some control over this process in that intermediate shapes can be specified, and influence shapes can be applied to the shape by adding additional dimensions. The results of this method above are an aesthetically pleasing and attractively simple approach to defining morph sequences between input contours.

Klimmek et al. [2007] introduces *shadow metamorphosis* by using the analogy of the sun moving about an object, the animation sequence being represented by the shadows being cast. Rotating and projecting the Cartesian product of $\mathcal{S} \times \mathcal{T}$ results in a continuous transformation from $\mathcal{S}$ to $\mathcal{T}$. By identifying contours which remain the same throughout the sequence, the product can be greatly simplified.

Most significantly, this method is the first method to allow transformations between surfaces of differing topologies while *exactly* reproducing $\mathcal{S}$ and $\mathcal{T}$ at the start and end of the sequence. Obvious limitations of this method are the lack of user control and the presence of self-intersections during the morphing sequence. However its elegance and the unique exact reproduction of the input constraints make it an important technique for comparison.

I will present a method which will draw upon the space–time representations of Aubert and Bechmann [1997] and Turk and O'Brien [1999], and use a Delaunay based planar cross-section method similar to that of Boissonnat [1988] in order to create a representation which exactly reproduces $\mathcal{S}$ and $\mathcal{T}$ and does not suffer from self–intersections. However, the results are linear and are therefore not as aesthetically pleasing as those of Turk and O'Brien [1999]. To my knowledge, this method is the first to use a Delaunay based method to connect cross-sections in $\mathbb{R}^4$.

8.2 Terminology

An n-simplex s is defined as the convex hull of a set of $(n+1)$ affine independent points in some Euclidean space of dimension n or higher. Familiar simplices include points (0-simplices), lines (1-simplices), triangles (2-simplices) and tetrahedra (3-simplices). I will also make use of 4-simplices, known as *pentatopes*.

A full space simplex in $\mathbb{R}^n$ is an n-simplex. Any lower simplex type is not capable of filling space within $\mathbb{R}^n$. For example, triangles in $\mathbb{R}^2$ can fill an area, while points and lines cannot.

The facets of a simplex refer to the $n+1$ $(n-1)$-simplices which form the boundary of the simplex. For example, a tetrahedron has 4 triangles (2-simplices) as facets. I refer to the process of extracting a list of facets from a simplex as *downgrading* a simplex.

Simplices are grouped together into a simplicial complex. An n-complex is essentially a set of simplices of type $(n-i), i = 0 \dots n$. If the simplices in a complex are all the same type it is called *homogeneous*. The $(n-1)$-complex consisting of all the downgraded simplices of the n-complex $\mathcal{C}$ is called the *downgraded complex* of $\mathcal{C}$.

In the context of defining a morph sequence, I will refer to our source and target manifolds as $\mathcal{S}$ and $\mathcal{T}$ respectively. These are oriented and embedded in $\mathbb{R}^n$, $n = 2, 3$. The filled space between these in $\mathbb{R}^{n+1}$ is A, while its boundary, the *animation manifold* is $\mathcal{A}$.

8.3 Morph validity

In this chapter I will define methods which can be used to morph between oriented manifolds. My approach is to define an *animation manifold* $\mathcal{A}$ which is a surface in $\mathbb{R}^{n+1}$, conforming with $\mathcal{S}$ and $\mathcal{T}$. This approach only supports convex polyhedra. I will then show two methods which can be used to modify $\mathcal{A}$ such that it will support input contours which are concave, consist of multiple components, or has complicated topology.

It should be noted that there are many *correct* answers to the morphing problem. It is possible, however, define what is meant by *validity* in the context of an animation manifold:

1. The animation manifold should not *self–intersect.*

2. The animation must conform with the input contours $\mathcal{S}$ and $\mathcal{T}$. An isosurface extracted from $\mathcal{A}$ at $t = t_0$ is exactly $\mathcal{S}$ and at $t = t_1$ is exactly $\mathcal{T}$.

3. Holes within the overlap between $\mathcal{S}$ and $\mathcal{T}$ must be preserved in $\mathcal{A}$. If

   $$H = U \setminus (S \cup T)$$

   where S and T are the spaces enclosed by $\mathcal{S}$ and $\mathcal{T}$ respectively, and U is the universal set, then

   $$\text{embed}(H, t) \cap A = \emptyset$$

   for all t.

Self–intersection (validity test 1) can occur if the animation manifold is edited in some way after triangulation, but not as a result of the basic Delaunay triangulation algorithm. I will show that these can easily be tested for in Section 8.6.3.

A basic requirement for the morph sequence to be valid is that the input contours $\mathcal{S}$ and $\mathcal{T}$ (validity test 2) are actually present in the animation sequence. Using my method, these meshes will be reproduced, with the possible addition of extra vertices and facets. Validity test 3 requires that holes that are common to both $\mathcal{S}$ and $\mathcal{T}$ are preserved throughout the animation sequence.

These properties differ from the requirements defined by Bajaj et al. [1996], i.e. *tiling*, *branching* and *correspondence.* This is because *tiling* is not an issue with a Delaunay method — the facets of $\mathcal{A}$ are watertight as they are the boundary of a watertight Delaunay triangulation. *Branching* and *correspondence* are user controlled parameters, although an automated result is possible.

8.3.1 Intermediate shapes

This definition of morphing validity relates only to what can be tested geometrically, but unfortunately it does not define what is meant by the plausibility of intermediate shapes. The topology and connectivity of the input contours are unconstrained, making it

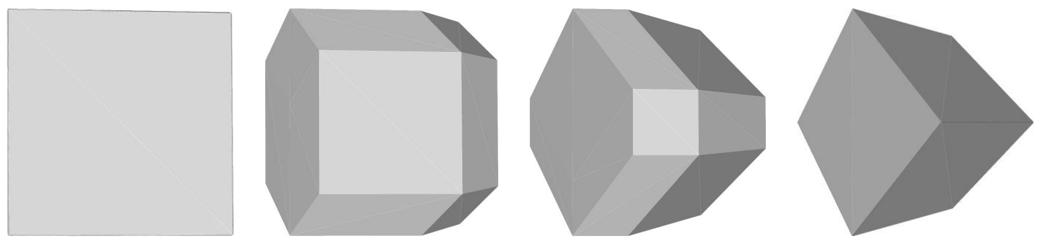


Figure 8.1: The plausibility of the intermediate shapes is difficult to assess. In this figure, a cube is connected to a rotated version of itself. A user may expect the cube to rotate from one orientation to the other, but instead the corners are cut to create the new shape.

difficult to control any particular morphing behaviour, alignment or intermediate shapes. An example of unpredictable morphing behaviour is given in Figure 8.1.

The validity criteria define geometric rather than artistic plausibility. A Delaunay triangulation of a point set yields well formed triangles, implicitly reducing thin simplices and preventing self–intersections, yielding transformations which are geometrically valid. The aesthetic issues surrounding what intermediate shapes are expected or considered plausible I leave as an area for future research, and is discussed in Section 10.1.

8.4 Morphing between convex polyhedra

In order to design an algorithm to morph between arbitrary closed polyhedra, I first consider the special case of convex polyhedra. This can be constructed by finding the convex hull of the points of $\mathcal{A}$. Instead I use a Delaunay approach which will be extended to support concave shapes and shapes of arbitrary topology.

In Chapter 5 I define a method to define a triangulation between two planar cross-sections in $\mathbb{R}^{n+1}$. An algorithm for this approach is given in Figure 5.10. Applying this method, and *extracting the boundary* yields an animation manifold $\mathcal{A}$ which defines the morph sequence between these two shapes.

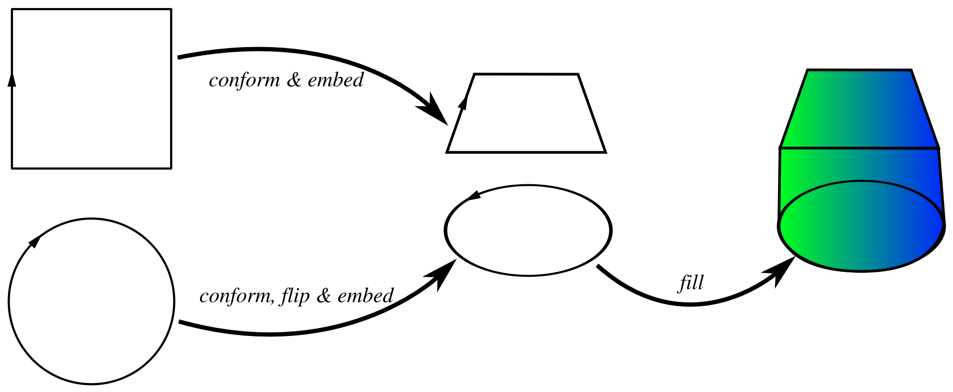


Figure 8.2: Defining the animation manifold which morphs between two convex shapes in $\mathbb{R}^2$. Note that the reason for flipping the source manifold will be made clear in Section 8.5.

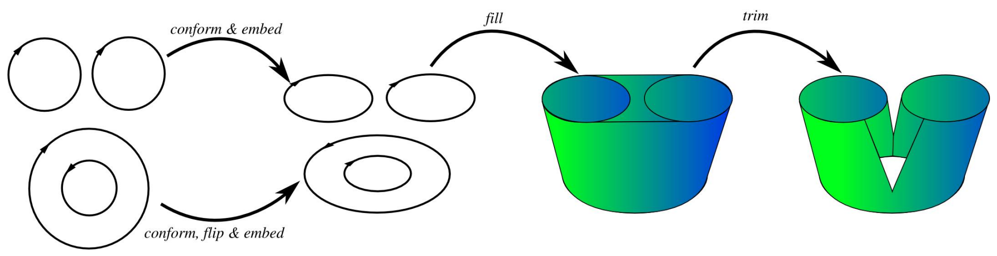


Figure 8.3: An overview of the method used to produce an animation manifold by simplex stripping.

In Figure 8.2, I demonstrate this method. Initially, the two oriented input surfaces $\mathcal{S}$ and $\mathcal{T}$ are made to be Conforming Delaunay. Then $\mathcal{S}'$ and $\mathcal{T}'$ are embedded in the same space, and the space between them is filled with a standard Delaunay triangulation. The animation manifold is the boundary of the resulting volume.

While simple, the method used to define a transition between two convex polyhedra is directly applicable to the method to morph between closed shapes with holes and concavities.

Concave polyhedra and polyhedra with genus higher than 0 are considerably more challenging. If the method presented in Figure 8.2 is used on a surface which is concave, the resulting manifold will not be valid as it will fill the convex hull of the combined shape, failing validity constraints 2 and 3.

In the following sections, I will introduce two methods by which a morph sequence can be created. *Simplex stripping*, which removes unwanted tets from the start or end contours, and *skeleton insertion* which appends the skeletons to each of the start and end contours before embedding them. Each produces *valid* results by our definition of validity in Section 8.3.

8.5 Morphing by simplex stripping

In this section I define a stripping algorithm which, given A and the input manifolds $\mathcal{S}$ and $\mathcal{T}$, will remove unwanted simplices lying within concave regions and holes. The results which are generated are valid by our criteria specified in Section 8.3, but results show that due to ambiguities, results may not be acceptable.

I will describe the process for any constraint manifold $\mathcal{C} \in \mathcal{S}, \mathcal{T}$. It is important for this algorithm that the orientation of $\mathcal{S}$ is *flipped*.

- Each of the full space simplices $s \in A$ is projected into the plane of $\mathcal{C}$ in $\mathbb{R}^n$. Call this non-homogeneous complex F. Note that F will fill the convex hull of the shape $\mathcal{S}$. A correspondence between $s \in A$ and the simplices in F is maintained. The projection process is described in Section 8.5.1.

- Mark each simplex $s \in A$ with the flag *unknown*.

- Starting from the oriented contour $\mathcal{C}$, use a region growing algorithm to identify *full space*, n-simplices of F which are *outside* of $\mathcal{C}$. Mark the corresponding simplices in A with flag *remove*. Mark the remaining full space simplices as *keep*.

- For each remaining $(n-i)$-simplex $k \in F$, $i = 1, \ldots, n$, determine the list K of neighbouring n-simplices. If all simplices in K are marked *remove* then this simplex is marked *remove*.

This operation is performed with both $\mathcal{S}$ and $\mathcal{T}$. After this, simplices flagged for removal in A are removed.

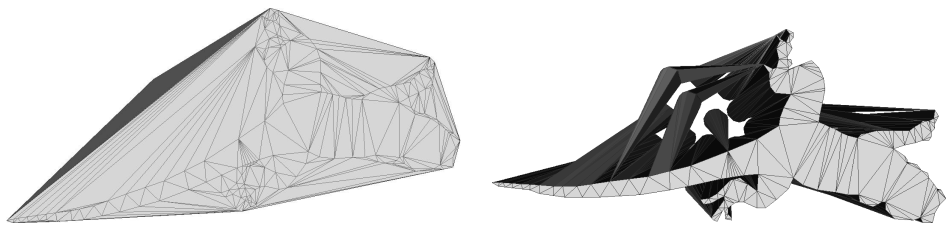


Figure 8.4: Filling and stripping. A lizard is morphed into a collection of circles. In (a) the convex hulls of both constraints are filled forming the filled region. This complex is then stripped of simplices outside the conforming constraints $\mathcal{S}'$ and $\mathcal{T}'$ to yield the animation surface in (b). This animation sequence is shown in Figure 8.7.

8.5.1 Finding the projected planar simplices

In Figure 8.5 I define a method to find the extracted simplices of A in the plane of the constraints $\mathcal{S}$ and $\mathcal{T}$.

Two important issues arise from this simple algorithm:

- The returned simplex should be an $(n-i)$-simplex, $i = 0, \ldots, n$. A full space simplex may result from this projection due to the Delaunay algorithm used (for example, in cases where the input vertices positions have a small random perterbation as a result of a Delaunay triangulation). These simplices add nothing to the morph sequence, and can be arbitrarily marked *remove*.

- If the returned simplex is not an n-simplex, then there may be many duplicates of the same simplex in the complex P in differing orientations. For example, if a tetrahedron is downgraded to four facets, each facet is unique. If each of those facets is downgraded into three edges, there will be two copies of each edge within the resulting complex. For this reason, the orientability of $(n-i)$-simplices, $i = 2 \ldots n$ cannot be determined.

Given an n-simplex $s \in A$ and the range r of indices of the constraint:

```
Append $s$ to an empty simplicial complex $P$
while (no simplex $p \in P$ has vertices all $\in r$)
    Downgrade complex $P$
end while
Remove any $p \in P$ with a vertex $\notin r$
```

Figure 8.5: An algorithm for finding the projected planar simplex from a given full space simplex s.

8.5.2 Growing holes

Using the facets of the manifold $\mathcal{C}$ as the initial active front, the algorithm grows outwards in order to identify simplices in F which are outside $\mathcal{C}$.

Each facet of $\mathcal{C}$ is added to a queue structure, called the *active front*. The simplices of F which border the facet at the front of the queue are tested for orientation. A simplex s which is outside of the facet is marked for removal and facets of s are appended to the queue.

If a facet is appended to the queue that is already in the queue (with an opposite orientation), both facets are removed. This means that the front has met itself and is cancelled.

If a facet in the queue borders no simplices in F which are oriented outside the facet, these are considered to be on the boundary and are removed. This process terminates naturally when the queue is empty.

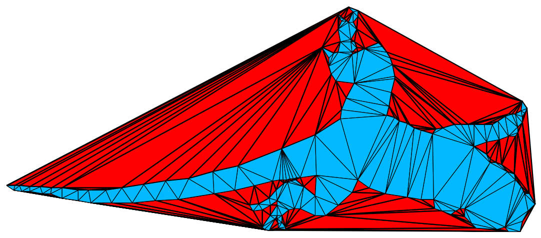


Figure 8.6: Separating the mesh region into *inside* and *outside* regions. This approach is necessary to identify holes.

8.5.3 Results and discussion

Some results of the simplex stripping algorithm are given in 2D in Figure 8.7.

Unfortunately, ambiguities arise with this approach even in $\mathbb{R}^2$. For example, if the two input contours $\mathcal{S} = \mathcal{T}$, an intermediate contour may not be the same as $\mathcal{S}$. Ambiguities such as those in Figure 8.8(b) can result in intermediate shapes having "bumps" where an edge-edge tetrahedron was not removed, or a hole where it was removed. This problem

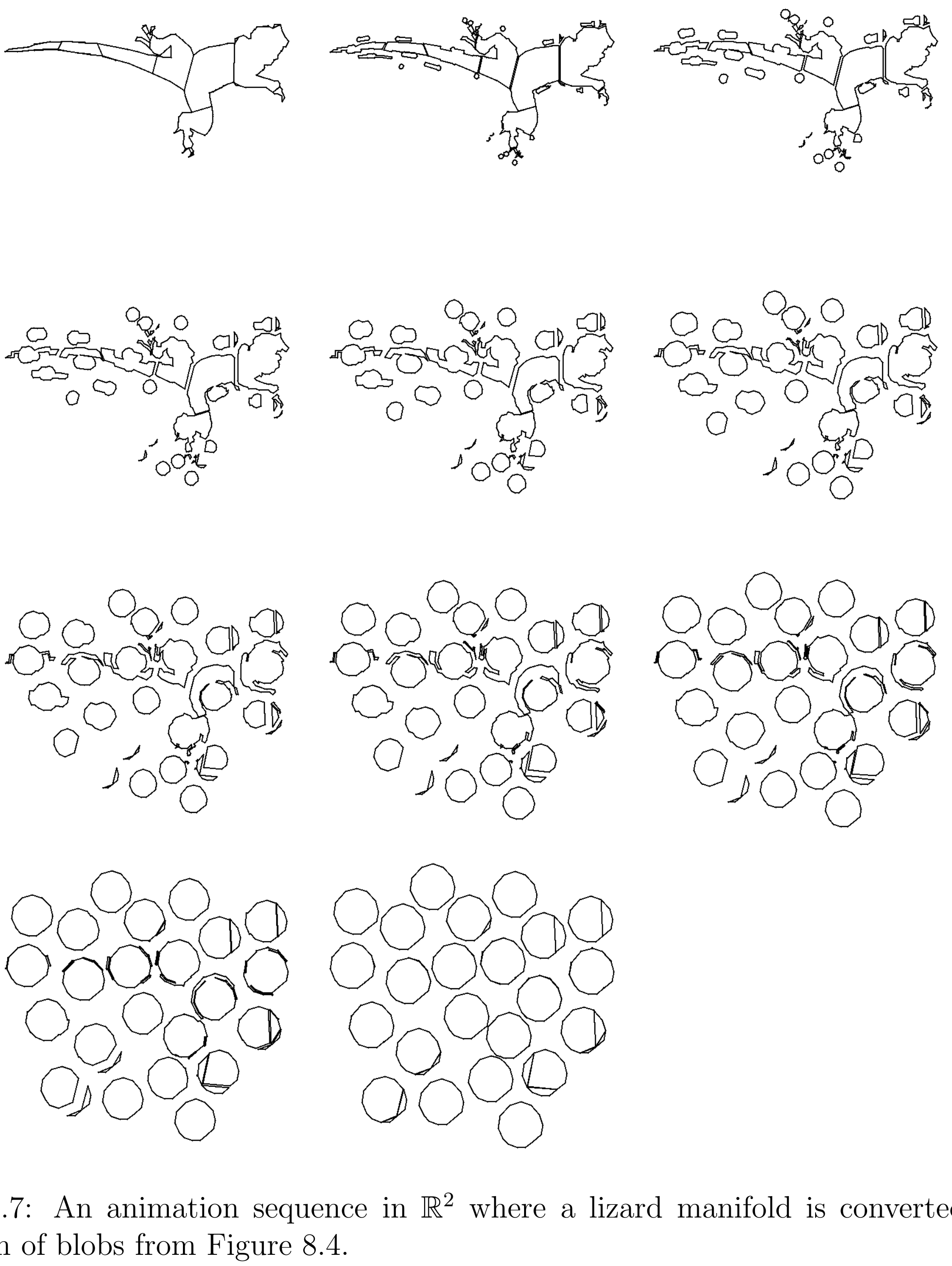


Figure 8.7: An animation sequence in $\mathbb{R}^2$ where a lizard manifold is converted into a collection of blobs from Figure 8.4.

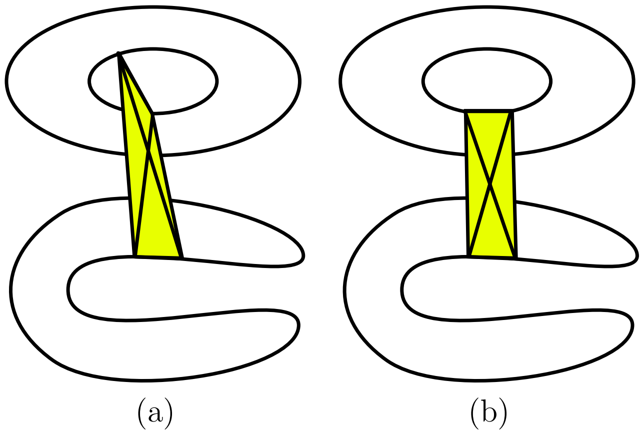


Figure 8.8: A simplex stripping ambiguity in $\mathbb{R}^2$. In (a), a simplex (in this case a tetrahedron) connects a hole to a source edge. The removal of this shape should be safe, and is marked *remove*. However in (b) a simplex connects the edge of the source contour to and edge of the target contour. According to the method for stripping, this will be marked as *unknown*. If this tetrahedron is removed, it may impact on the quality of the result, causing the transformation to bulge outwards or shrink inwards.

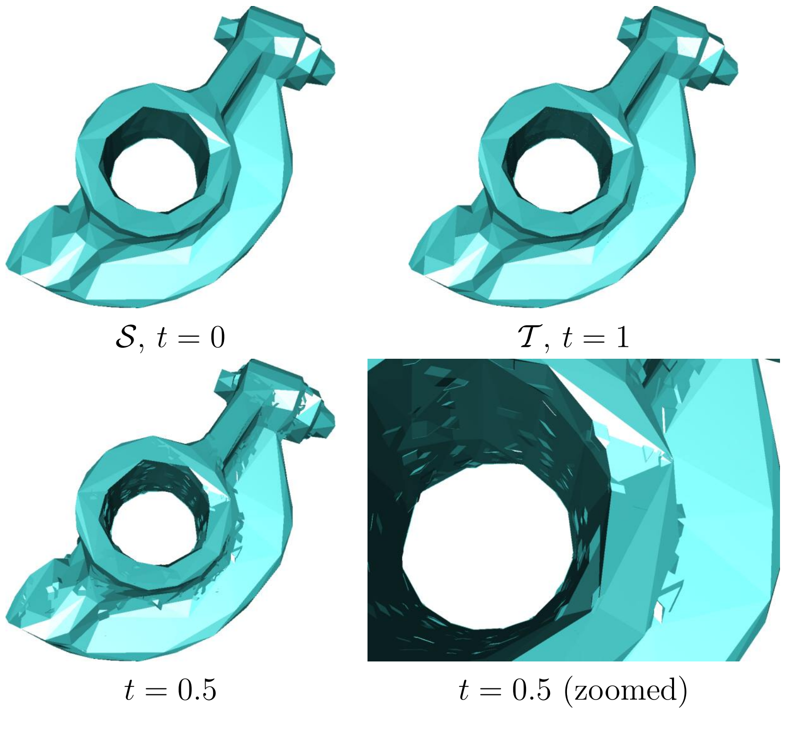


Figure 8.9: A case of simplex stripping ambiguity in $\mathbb{R}^3$. The rockerArm mesh is specified as both the source and the target model. While the animation sequence exactly reproduces the input meshes $\mathcal{S}$ and $\mathcal{T}$, the intermediate frame shows that due to stripping ambiguity, some unwanted features appear exterior to the mesh.

is exacerbated when morphing between contours in $\mathbb{R}^3$, as face–edge tetrahedra also give such ambiguities.

I resolve this by embedding the skeleton of $\mathcal{S}$ and $\mathcal{T}$ into the animation sequence. Simplex stripping is still required as a post-process to remove unwanted external simplices.

8.6 Using the skeleton

The ambiguities discussed in Section 8.5.3 only occur in concave regions. My method to resolve this problem is to connect the manifold in the concave region to its internal *skeleton*, eliminating the region. After the full space complex A has then been constructed, the vertices of this skeleton are then projected or "pushed" to a different point in time.

This gives the impression that all holes $\mathcal{S}$ are *closed* before they are transformed into the shape of $\mathcal{T}$, and similarly the holes of $\mathcal{T}$ are introduced earlier than the final instance of $\mathcal{T}$. I ensure that holes which are in both shapes are preserved in the final sequence by using a customised renderer.

8.6.1 Method overview

I follow the same approach as that given in Section 8.4, except that the initial input contours $\mathcal{S}$ and $\mathcal{T}$ are modified to contain their own external Voronoi skeleton.

Recall that $\mathrm{DT}(\mathcal{C})$ returns the Delaunay triangulation of the points of $\mathcal{C}$, $\mathrm{C}_F\mathrm{DT}(\mathcal{C})$ returns the Conforming Delaunay complex of $\mathcal{C}$ (both defined in Section 5.1.1 and $VS_{out}(\mathcal{C})$ refers to the external Voronoi skeleton (defined in Section 7.4).

1. Find the skeletons of $\mathcal{S}$ and $\mathcal{T}$, $VS_{out}(\mathcal{S})$ and $VS_{out}(\mathcal{T})$ respectively. This approach is discussed in Chapter 7.
2. Find $\mathcal{S}' = \mathrm{C}_F\mathrm{DT}(\mathcal{S} \cup VS_{out}(\mathcal{S}))$, and similarly find $\mathcal{T}'$. The skeleton may be *attached* to the input manifold — see Section 8.6.2.
3. Construct $\mathcal{F}' = \mathrm{embed}(\mathcal{S}', 0) \cup \mathrm{embed}(\mathcal{S}', 1)$.
4. Find $A = \mathrm{DT}(\mathcal{F}')$. These two steps are the same as those in Chapter 5.
5. Push vertices from $VS_{out}(\mathcal{S})$ and $VS_{out}(\mathcal{T})$ into A. This will be described in Section 8.6.3.
6. Strip simplices of A if necessary.
7. Extract boundary of A.

This process is shown in Figure 8.10.

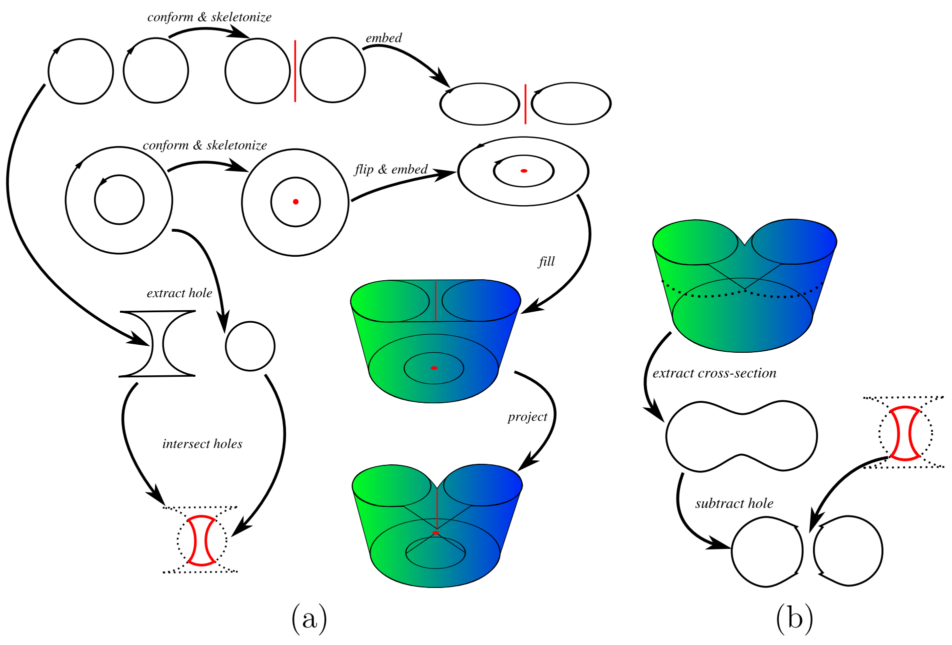


Figure 8.10: In (a) the process of defining an animation manifold in $\mathbb{R}^2$ using skeletonisation is shown. The external skeleton (shown in red) is first identified from both input shapes. The input shape along with its skeleton is then made conforming. Both these shapes are then embedded in the same space which is then filled. The vertices of the skeleton are then pushed *into* the filled shape. At the same time, the intersection of the concavities of both shapes is extracted for the renderer. In (b) the extracted hole intersection is used in combination with the final shape to render a cross section while preserving the interior holes shared by both input shapes.

8.6.2 Attaching the skeleton

The process of identifying the external Voronoi skeleton is given in Chapter 7, and the same method is applicable here. For an input manifold $\mathcal{C} \in \{\mathcal{S}, \mathcal{T}\}$, the $\mathrm{VS}_{out}(\mathcal{C})$ will be an external skeleton which is not attached to $\mathcal{C}$. This is undesirable as in order to ensure that the projection of the skeleton into the shape is successful, no simplex may span a hole in after the triangulation. An example of this situation is shown in Figure 8.11.

Using the Voronoi separable manifold which results from $\mathrm{VS}_{out}(\mathcal{C})$ as $\mathcal{C} = \{P, F\}$ in $\mathbb{R}^{n-1}$, I identify each vertex $p_i \in P$ for which the Voronoi cell ∂C_i such that $\partial C_i \cap \mathrm{VS}_{out}(\mathcal{C})$ is a $(n-i)$-simplex, with $i = 2 \ldots n$. These vertices p_i are then attached to the simplex $\partial C_i \cap \mathrm{VS}_{out}(\mathcal{C})$. If more than one vertices p_i are attached to the same simplex then these are combined to form a higher degree simplex.

For example, in $\mathbb{R}^2$, if the intersection of ∂C_i and $\mathrm{VS}_{out}(\mathcal{C})$ is a single vertex, then p_i is attached to that vertex forming an edge. In $\mathbb{R}^3$ the intersection may be a vertex or an edge. In the case of it being a vertex, then the two vertices p_i and p_j are attached to the vertex to form a facet. If $\partial C_i \cap \mathrm{VS}_{out}(\mathcal{C})$ is an edge, then the vertex p_i is attached to this edge to form a facet.

There are a number of issues with attaching the skeleton to an input manifold $\mathcal{C}$:

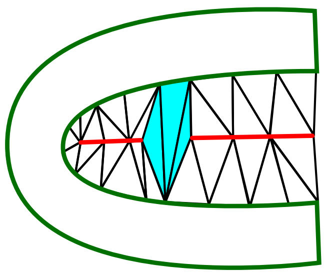


Figure 8.11: In this figure, the light blue facets connect the interior of the object to itself, bypassing its incomplete skeleton. This will given undesirable results when the embedded skeleton is pushed to a new t position.

- Attaching the skeleton will in general create *sharp dihedral angles* between $\mathrm{VS}_{out}(\mathcal{C})$ and $\mathcal{C}$, as the skeleton will, in general, bisect the corner angle. All CDT implementations will have difficulty dealing with angles that are sufficiently sharp (with *Tetgen* for example, a practical dihedral angle limitation of $5\pi/6$ radians is imposed).
- $\mathrm{C}_F\mathrm{DT}(\mathrm{VS}_{out}(\mathcal{C}) \cup \mathcal{C})$ may insert vertices into the simplex attaching $\mathrm{VS}_{out}(\mathcal{C})$ to $\mathcal{C}$. This makes Step 5 more difficult, as there are vertices which are not part of $\mathcal{C}$ nor of its associated skeleton in the resulting manifold. A method for dealing with these vertices is given in Section 8.6.3.

It is for this reason that in my implementation, $\mathcal{C}' = \mathrm{C}_F\mathrm{DT}(\mathrm{VS}_{out}(\mathcal{C}) \cup \mathcal{C})$ is first computed with $\mathrm{VS}_{out}(\mathcal{C})$ *unattached* to $\mathcal{C}$. I then compute $\mathrm{DT}(\mathcal{C}')$ and see if any simplices spanning concave regions of $\mathcal{C}$ only include vertices of $\mathcal{C}$. If this is the case, then I attach $\mathrm{VS}_{out}(\mathcal{C})$ to $\mathcal{C}$ and recompute $\mathcal{C}'$. None of the examples I tested have required this attachment procedure.

8.6.3 Pushing the skeleton

Once A has been deduced, its shape is altered by "pushing" the vertices of $\mathrm{VS}_{out}(\mathcal{S})$ and $\mathrm{VS}_{out}(\mathcal{T})$ *into* the volume A. This is applied using an iterative approach. I define the simple operator $\mathrm{push}(A, \mathrm{VS}(\mathcal{C}), t)$ which sets the last coordinate axis to t. Applying operator $\mathrm{push}(A, \mathrm{VS}(\mathcal{S}), t_0 + \varepsilon_0)$ and $\mathrm{push}(A, \mathrm{VS}(\mathcal{T}), t_1 - \varepsilon_1)$ however can cause self intersections in the resulting volume.

To avoid this, I introduce a simple test for self-intersection. Since, due to the properties of the Delaunay triangulation, the hypervolume of all enclosed simplices of A will be positive. When applying the operator $\mathrm{push}(A, \mathrm{VS}(\mathcal{C}), t + \varepsilon)$, the signed hypervolume of simplices of A containing vertices of $\mathrm{VS}(\mathcal{C})$ are tested. If the volume of any of these simplices is negative, ε is modified and it is recomputed. I use a simple binary search algorithm to find a safe location for the vertices of $\mathrm{VS}(\mathcal{C})$ in A.

Additionally, as discussed in Section 8.6.2, the simplices attaching the skeleton to $\mathcal{C}$ may be split as a result of the $\mathrm{C}_F\mathrm{DT}$ operation. In general the pushed locations of these vertices are deduced by linear interpolation.

8.6.4 Stripping unwanted simplices

As the skeleton is limited to simplices which lie within the convex hull of the initial shape (see Chapter 7), some simplices of A may span holes *at the boundary* of A. These are detected and stripped using the same process in Section 8.5. I have found that the ambiguities identified in Section 8.5 are not noticeable.

8.6.5 Rendering the result

In order to comply with validity constraint 3, some special handling is required to deal with topological preservation during the animation sequence. I use a specialised raytracer to reproduce shared holes and concave regions. This is shown in Figure 8.10(b).

During the building of the morph sequence, a "hole space" $H_\mathcal{C}$ is extracted from each input manifold $\mathcal{C} \in \{\mathcal{S}, \mathcal{T}\}$. These can be defined as

$$\mathcal{H}_\mathcal{C} = (\mathrm{CH}(\mathcal{C}) \setminus C) \cup (C \setminus \mathrm{CH}(\mathcal{C}))$$

where $\mathrm{CH}(\mathcal{C})$ is the volume enclosed by the convex hull of manifold $\mathcal{C}$, and C is the volume enclosed by $\mathcal{C}$. I refer to $\mathcal{H}_\mathcal{C}$ as the boundary of this hole. I define the hole boundary as $\mathcal{H}_\mathcal{C} = \partial H_\mathcal{C}$.

My custom raytracer then combines the final manifold boundary in $\mathbb{R}^n$, $\mathcal{A}$, with the $\mathbb{R}^{n-1}$ manifolds $\mathcal{H}_\mathcal{S}$ and $\mathcal{H}_\mathcal{T}$ using the boolean set operation $\mathcal{A} \setminus (\mathcal{H}_\mathcal{S} \cap \mathcal{H}_\mathcal{T})$.

8.7 Implementation

As in Chapter 7, the implementation of my technique makes use of several self-contained executables. Along with *wrap*, *skel* and *trim* defined in Section 7.6.6, there are several additional programs which are necessary for creating the conforming geometry with the skeleton attached, and for filling combined shapes and extracting the boundary.

- *comb* combines two input complexes.
- *embed* raises the input complex to a higher dimension by setting the final positional component to the specified input value.
- *fill* applies either a conforming fill of some input geometry using the *Triangle* package of Shewchuk [1996] for input in $\mathbb{R}^2$, the *Tetgen* package of Si and Gaertner [2005] for input in $\mathbb{R}^3$, or alternatively connects the points of any input manifold with a Delaunay triangulation using the *QHull* package of Barber et al. [1996].
- *find* identifies a mesh in a filled complex. This is used to find the conforming input manifold and skeleton from the result of the *fill* package.
- *push* accepts a t offset and a skeleton mesh as input, and applies the offset to the vertices of the skeleton in a filled manifold.

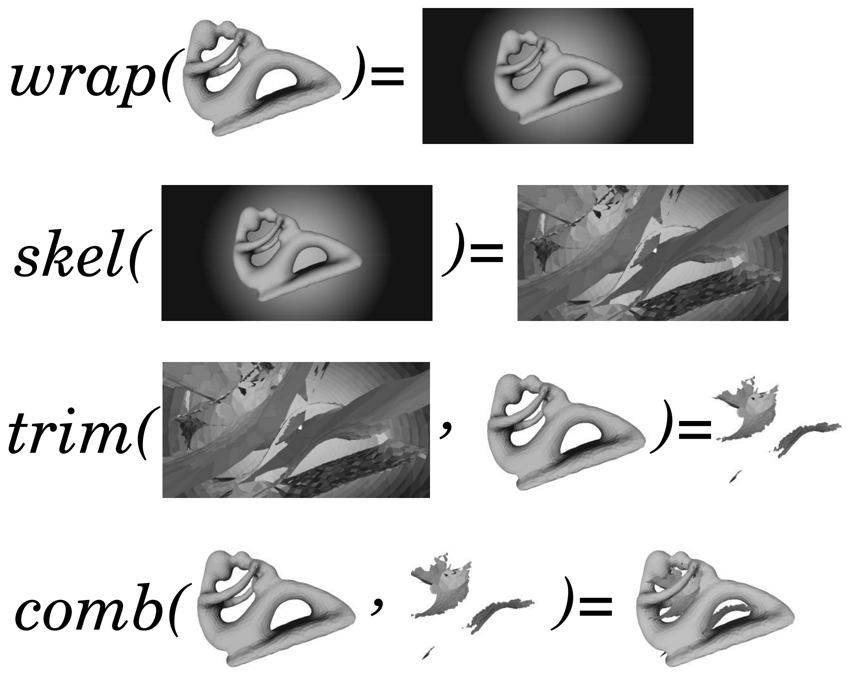


Figure 8.12: The first part of the conforming process for the *fertility* model. The external skeleton of the input mesh is found, and then combined with the input mesh.

- *strip* accepts the conforming mesh $\mathcal{M}'$ and filled shape A as input, and removes simplices of A which, when projected into the plane of $\mathcal{M}'$ lies outside of $\mathcal{M}'$.
- *bound* extracts the boundary $\mathcal{A} = \partial A$ of a full space complex.
- *hole* identifies and extracts the hole in a shape. This is used for defining the subtractive shapes $\mathcal{H}_{\mathcal{S}}$ and $\mathcal{H}_{\mathcal{T}}$, which are used with the renderer in Section 8.6.5.

The process of creating the morph sequence in Figure 8.18 is given in Figures 8.12, 8.13, 8.14 and 8.15.

The process of finding the external skeleton and attaching it to the input mesh is given in Figure 8.12. This is then wrapped and filled in Figure 8.13, and the conforming versions of the input mesh and skeleton are extracted from this filled shape.

In Figure 8.14 the filling stage of the algorithm is shown. A filled sphere model is embedded in $\mathbb{R}^4$ at $t = 0$, while the filled *fertility* mesh is embedded in $\mathbb{R}^4$ at $t = 1$. The results are then combined. In Figure 8.15 the vertices of the combined model are filled using a standard Delaunay triangulator. Due to the results of Theorem 5.2.1 the conforming geometry of our input shapes will be preserved in the resulting triangulation.

The skeleton of the *fertility* model is then pushed by some t value such that simplices do not self–intersect (in this case 0.2). Then the shape is stripped of simplices which lie outside of the input model to yield the final shape. Note that the resulting shape is filled — the boundary of this shape must be extracted using the *bound* function to be rendered.

In Figure 8.16 I demonstrate a 2D morph sequence which uses the skeleton pushing approach. In Figures 8.17 and 8.18 examples in $\mathbb{R}^3$ are shown.

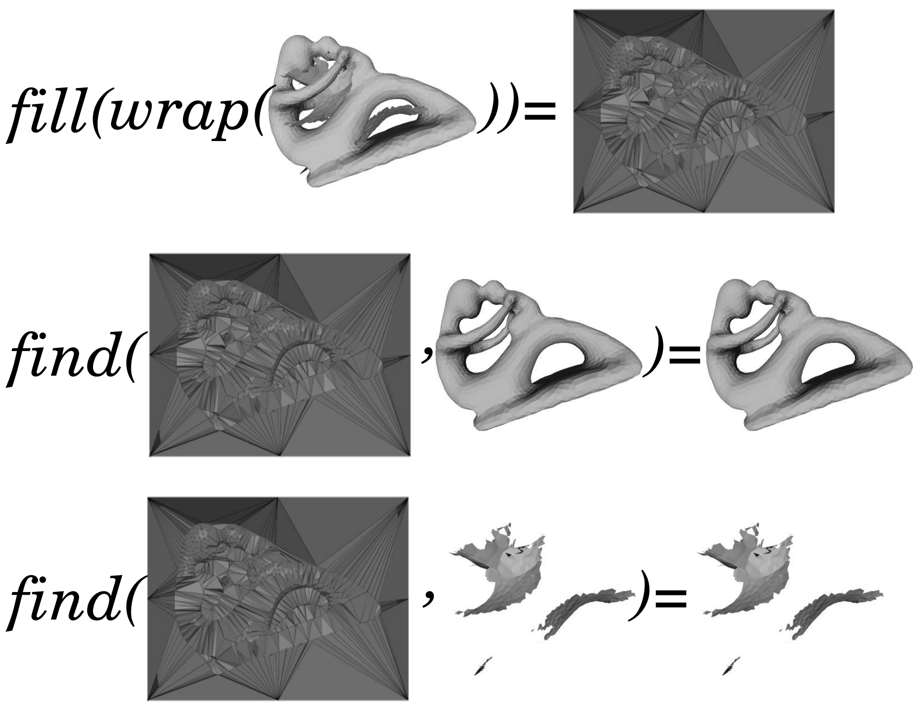


Figure 8.13: The second part of the conforming process. The combined mesh is wrapped and filled using a conforming triangulator, in this case *Tetgen*. The conforming versions of the initial mesh and the skeleton are extracted from this conforming volume.

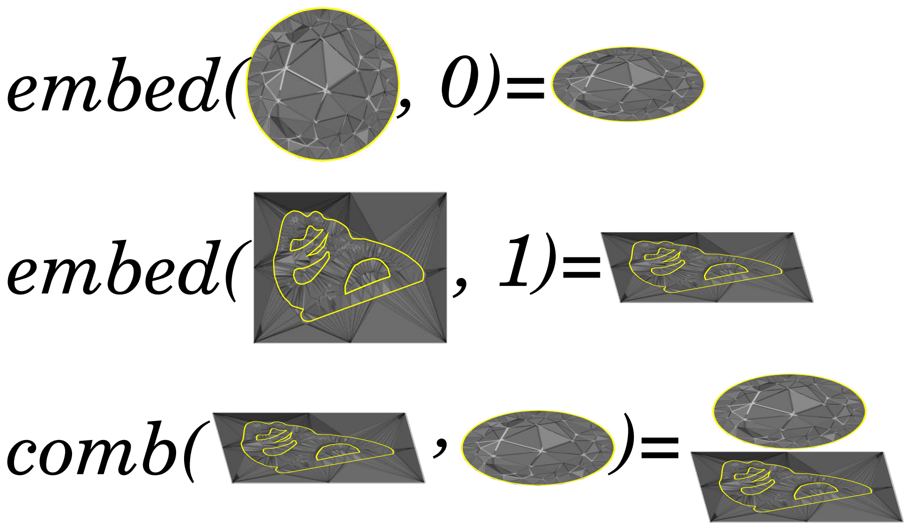


Figure 8.14: The first stage of the filling process. Both filled meshes are embedded in a higher dimensional space and then combined.

8.7.1 Stability and performance

This work makes use of several existing tools. *QHull* Barber et al. [1996] is used to perform Delaunay triangulations of higher dimensional space, which is robust and scales well to dimensions higher than $\mathbb{R}^3$. For conforming Delaunay triangulation in $\mathbb{R}^2$ I make use of the *Triangle* package of Shewchuk [1996]. In $\mathbb{R}^3$ conforming Delaunay triangulation

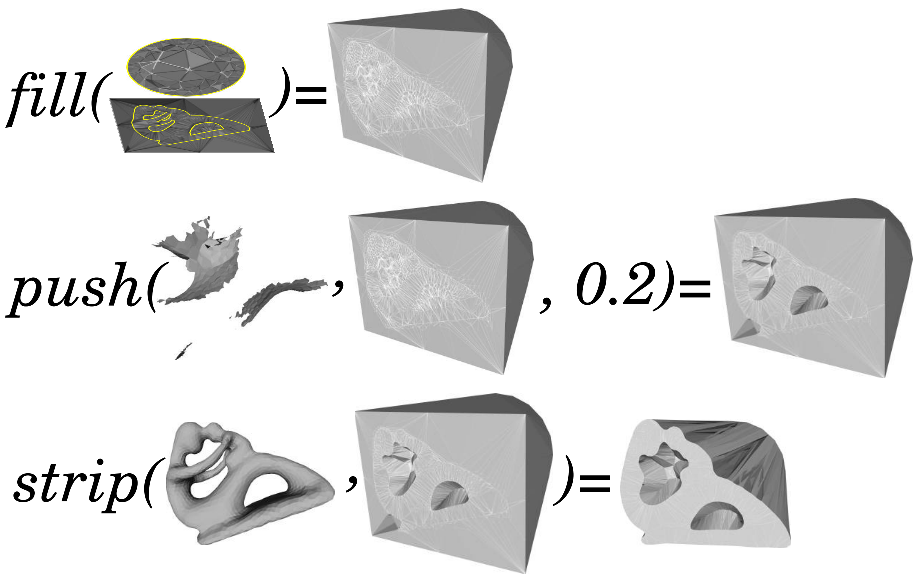


Figure 8.15: The second stage of the filling process. The combined model is filled. vertices of the external skeleton of the *fertility* model are "pushed" into the filled shape. The result is then stripped of simplices which lie outside the conforming version of the input model. Note that volumes in $\mathbb{R}^4$ are visualised by holding the $z = 0.5$ axis constant.

is performed by, to my knowledge, the only freely available conforming tetrahedral mesh generator, *Tetgen* of Si and Gaertner [2005].

While *Triangle* is robust, fast and the output is guaranteed to be conforming, *Tetgen* may produce a triangulation which is *constrained* Delaunay, but not *conforming* Delaunay. This has necessitated the implementation of a testing procedure. The vertices of the filled mesh M created by *Tetgen* are extracted. *QHull* is then executed on these vertices, creating a filled test volume V. If the result of *find*$(\mathcal{M}, V)$ is an incomplete manifold, then *Tetgen* will have failed to produce a conforming triangulation. For this reason, input models are first preprocessed using the uniform remeshing tool ReMESH of Attene and Falcidieno [2006].

The execution time of most of the executables is negligible, with the exception of *fill*, *find*, *skel* and *strip*. Most of these operations are linear or near linear, with the major time overhead being file handling. The performance of *skel* is discussed in Section 7.6.6. All running times were measured on a standard commodity PC with 2 GB RAM.

Using *fill* to find the conforming Delaunay triangulation of the 3000 vertex *fertility* mesh takes 6.6 seconds and uses a development version of *Tetgen*. Filling the combined complex of the filled *fertility* and *sphere* models, which consists of 19006 vertices in $\mathbb{R}^4$ took 12.24 seconds using *QHull*. The performance of each of these packages appears to scale linearly with the number of vertices in my experiments.

It took 559.5 seconds to *find* the updated manifold $\mathcal{M}'$ from the filled volume M, and a similar time to identify the updated skeleton. The performance is attributed to the computation of the surface normals for each of the downgraded facets of M. It should be noted that a conforming Delaunay triangulation implementation could be made to keep

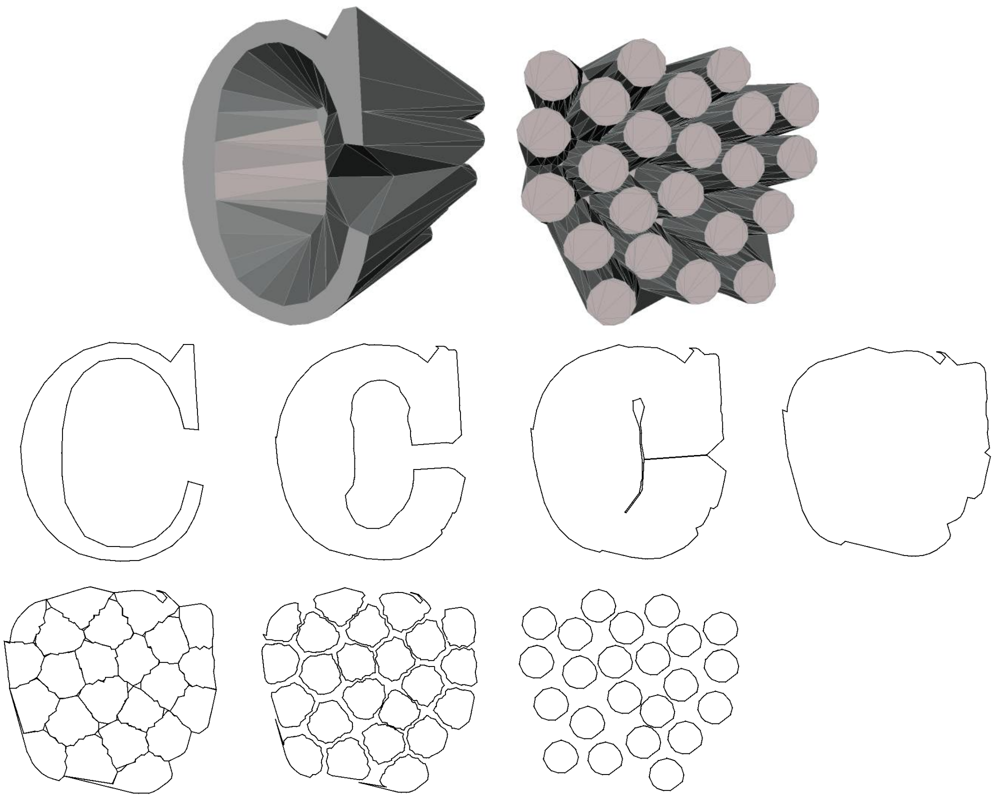


Figure 8.16: Using the skeleton to define a morph sequence. In this 2D animation sequence both shapes are morphed into each other via a genus-0 object by adding the skeleton to the input manifolds, and "pushing" the skeleton into the resulting shape. This process is described in Section 8.6.3.

track of the updated conforming manifolds, making this time consuming task unnecessary.

Stripping the 6010 unwanted simplices outside $\mathcal{M}'$ from A (170039 simplices) took 227.1 seconds. This required the projection of simplices of A into the plane of $\mathcal{M}'$. Of this, 39.9 seconds was used to grow the region necessary to identify which simplices lie outside of $\mathcal{M}'$.

8.8 Summary

In this chapter I have introduced the problem of defining smooth transformation between input manifolds by dealing with them as cross-sections of some time series. I have introduced three related methods which can be used to define morph sequences between input models, each with its own unique limitations:

- In Section 8.4 I define a method to morph between convex polyhedra by simply using the constrained Delaunay method defined in Chapter 5.
- In Section 8.5 a method for simplex stripping the result of the previous method applied to non-convex polyhedra is defined. Unfortunately this method leads to

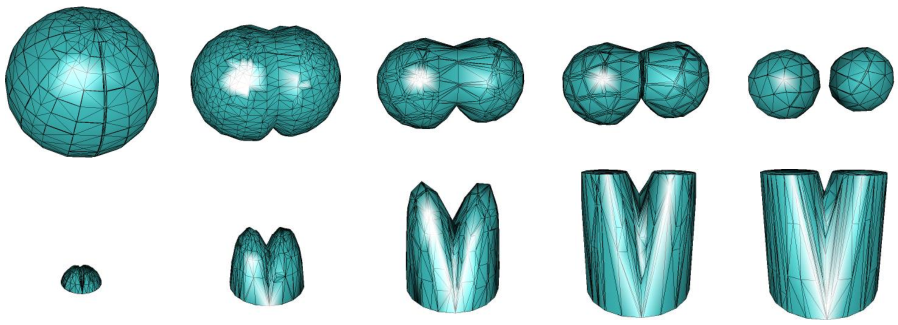


Figure 8.17: Splitting a sphere using the two sphere model and its skeleton. In the top row, t is held constant for each frame, showing the actual animation sequence. In the second row, z is held constant, giving us a visualisation of the "trouserlegs of time".

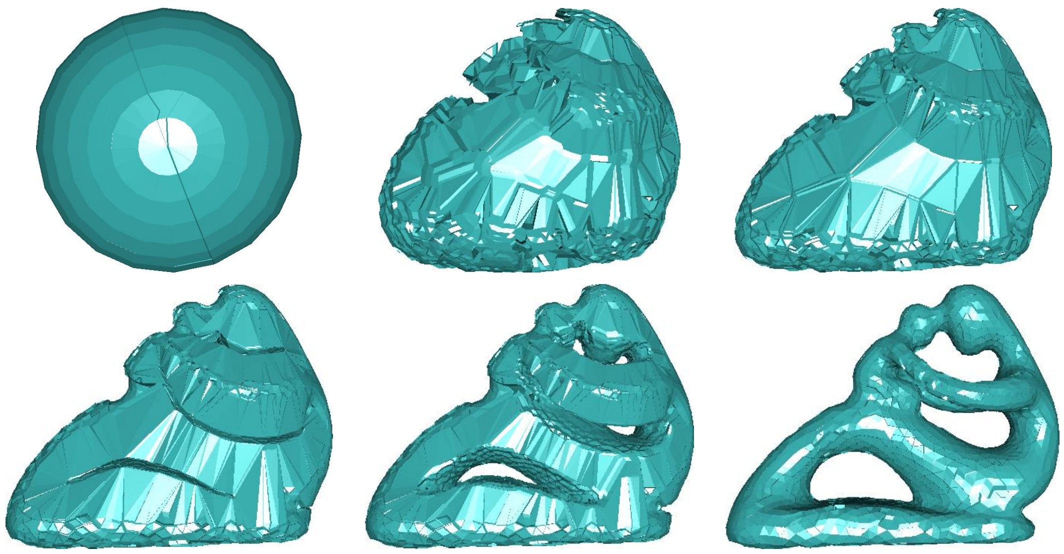


Figure 8.18: This figure demonstrates the conversion from the sphere to the fertility model, which is genus 3. The topology is altered when the skeleton is introduced between the third and fourth frame.

ambiguities which cannot be accurately identified by observing the correspondence of the final hypervolume A with the input manifolds $\mathcal{S}$ and $\mathcal{T}$.

- In Section 8.6 I modify the simplex stripping approach by inserting or attaching a Voronoi skeleton to the input mesh to the input manifolds $\mathcal{S}$ and $\mathcal{T}$ before building the full space manifold A, and then pushing the skeletons into A. The result is that at some stage of the morph sequence the mesh will be a genus-0 object, regardless of the input geometry. This is corrected in the rendering step by subtracting shared holes from both objects from the rendered result.

In this chapter I have made use of the *fertility* model acquired from the AIM@SHAPE model repository (`http://www.aimatshape.net`), and the rocker arm model from the INRIA GAMMA project
(`http://www-c.inria.fr/gamma/download/`).

Chapter 9

Summary

In this dissertation I have presented the animation manifold, and argued for its use as a viable and unified structure for representing and modelling topological alteration of animation sequences. I have presented methods by which they may be constructed, deformed, visualised and rendered.

9.1 Construction

Animation manifolds can be directly constructed using either the prism based keyframe stitching approach of Section 2.2, or using the conforming Delaunay based volume filling method of Chapter 8.

Keyframe stitching utilises a simple prism building (sometimes called "tent–pegging") approach to iteratively build a keyframe sequence. It requires an exact point to point correspondence of vertices between different keyframes, and is therefore relevant to converting from existing animation sequences to animation manifolds.

The morphing approach is useful for connecting two arbitrary closed surface manifolds without vertex correspondences. My approach is to initially identify a Voronoi skeleton for each surface using the approach presented in Chapter 7, and then attach it to the surface. The skeletonised surfaces are then connected using the conforming Delaunay triangulation algorithm of Chapter 5. The skeleton is then projected into the resulting higher dimensional structure in order to achieve branching behaviour. A customised CSG raytracer is then used to render the result in a manner that preserves holes through both surfaces.

9.2 Deformation

In Chapter 4 I adapted the Laplacian editing technique to the problem of deforming animation manifolds. I have explored three methods which are relevant in the higher dimensional context: the basic equal weight approach of Taubin [1995], the graph Laplacian of Zhou et al. [2005] and the dual mesh Laplacian of Au et al. [2006], and have adapted these to higher dimensional mesh editing.

Animation manifolds typically have an open boundary as they are constructed by appending consecutive keyframes. In Section 4.4 I derive boundary conditions for both primal and dual graph Laplacian editing techniques. In the primal case the graph Laplacian weights are used, while in the dual case a reflected face is incorporated into the dual Laplacian system.

A problem shared by all manifold editing techniques is polygon density of the deformed mesh after significant extrusion. In Section 4.5 I make use of a novel offline adaptive refinement technique to refine the surface after a deformation has been applied. An online adaptive subdivision algorithm is also proposed.

9.3 Rendering

Both of the traditional surface rendering techniques apply to animation manifolds. In Section 2.5 a higher dimensional ray tracing solution is introduced. It offers an offline rendering solution with a high quality result. I discovered that the use of a higher dimensional spacial partitioning structure, called the hypertree, significantly improves rendering performance.

A method for real-time iso-surface extraction is presented in Section 2.6 which draws from previous literature in the field of tetrahedral volume visualisation. This method can be implemented in graphics hardware to produce an interactive rendering solution for animation sequences.

Visualising animation manifolds can be challenging. An independent axis visualisation is shown in Figure 2.7, which may help animators to predict editing behaviour. However a practical interface for working in higher dimensions is an open problem.

Chapter 10

Conclusions and future work

There are some general areas of development which may help transform animation manifolds into a viable medium for representing animating scenes:

10.1 Morphing

There are several areas relating to the problem of the shape transformations of Chapter 8 which would benefit from further research. Most of these improvements relate specifically to the problem of aligning the surfaces through deformation and user guidance.

10.1.1 Delaunay techniques

It is clear from Chapter 5 that the problem of constrained Delaunay triangulation in dimensions higher than 3 is still an open problem. In fact, the angle limitations of CDT implementations in $\mathbb{R}^3$ show that this still warrants further investigation. A practical and robust algorithm and implementation of a general dimensional constrained Delaunay triangulation algorithm developing on from the work of Shewchuk [1998] would be of great interest.

Another possible extension to the method described in Chapter 5 is the use of *guiding shapes*. The skeleton which I use for smoothly converting input shapes to genus 0 can be thought of as an example of a guiding shape which is embedded in the Delaunay volume. I have only used external skeletons in my experiments, although internal skeletons, and possibly alternative intermediate shapes could also be deduced which would alter the morphing path. This additional level of flexibility is an interesting avenue for further work.

10.1.2 Alternative connection methods

Initially I attempted to resolve the problem of connecting planar cross–sections by attempting to generalise the contour connecting approach of Bajaj et al. [1996] (amongst others). This method "walks" around the input contours, building a surface connecting the two, and identifying special surface behaviours such as branching. However, while in

2D curves have a natural ordering, which indicates in what direction to "walk". Unfortunately the orientation of a 2-simplex is meaningless in $\mathbb{R}^4$.

The main advantage of the contour walking approach is the relaxation of the Delaunay condition, allowing the user to specify alignment requirements as connectivity constraints, for example ensuring that the ears of a bunny transform into the ears of the horse. In order to ascertain the orientation of the 2-simplex in $\mathbb{R}^4$ it may be possible to employ *geometric algebra*, a special form of Clifford algebra. This is an open question.

10.1.3 Alignment by deformation

Alignment of surfaces is currently restricted to rigid body transformations (rotations, uniform scaling, translation), as more sophisticated deformations to either the source or target models may result in a failure of the Delaunay condition of the undeformed model. For example, if we are transforming from a bunny into a torus, we might try to extrude the bunny dramatically and make it into a ring in order to align it with the torus. However due to the linear paths connecting the two shapes, the animation manifold of the unfolded bunny to the torus will almost certainly contain self intersections. While these can be detected, it is not clear how these may be corrected.

Sabin [2007] suggests that deformations based on a global best fit *Möbius transformations* of the form $f(z) = (az + b)/(cz + d)$ for $a, b, c, d \in \mathbb{C}$ could be used. This type of transformation is unique in that it is *circle* preserving in $\mathbb{R}^2$, and would therefore theoretically not violate the Delaunay property. Such a system would allow the user to specify arbitrary point correspondences, and the alignment would take the form of a best fit Möbius transformation.

10.2 Geometric tools

In this dissertation I have made use of and adapted a variety of geometric tools which are typically applied to surface meshes. There are a number of areas in which these approaches could be improved.

10.2.1 Deformation

While the Laplacian editing paradigm which I have used is the most general approach, there are other published methods, such as detail transfer and sketch based interfaces which would have interesting applications to animation manifolds. For example, a user could specify that some surface detail be added to an animation sequence by using a detail transfer function such as a repeated geometric texture.

The geometric extrusion method defined in Section 4.5 is simplistic, but also opens up some considerably more interesting application areas. Texture synthesis, and more recently *geometry synthesis* are techniques which propagate surface feature detail. Using some combination of the surface detail transfer approach, a geometric synthesis method and a geometric extrusion technique would be a powerful tool for propagating surface properties in deformed regions, such as the bumps on the leg of the armadillo model.

A general problem with Laplacian editing, and indeed many deformation techniques, is their insensitivity to general structure, typically characterised by a skeleton. The only exception is the method of Zhou et al. [2005] which takes the contained volume of the shape rather than any internal skeleton. A skeleton–aware Laplacian editing platform would be of considerable interest, and may be possible by using a skeletal structure as part of the Laplacian system which ensures that skeleton edges are not excessively compressed or stretched.

The most important shortcoming of the methods presented in Chapter 4 is the modelling paradigm for the user interface. I have found that using the anchor and handle specification method on animation manifolds to be counter-intuitive and challenging to use, and could be improved with and automatic method to deduce these handles and anchors. The development of an intuitive editor for higher dimensional deformation, such as the technique of Brandel et al. [1998], would be particularly useful in building a commercial grade application for editing animation manifolds.

10.2.2 Subdivision

Smoothing the animation manifold affects both the quality of the frames extracted from the animation sequence, as well as smoothing the path of motion, as can be seen in Figure 4.6.

Throughout this dissertation I have made use of the approximating tetrahedral subdivision method and implementation of Schaefer et al. [2004], which produces C^1 surfaces in general. One problem inherent in standard, stationary subdivision methods is their susceptibility to artifacts such as those identified by Sabin and Barthe [2002]. An example of a lateral artifact is shown in Figure 10.1.

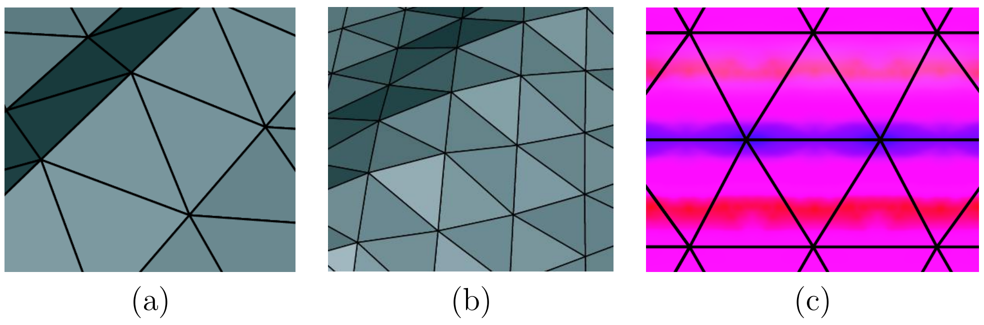


Figure 10.1: An example of lateral artifacts in subdivision in $\mathbb{R}^3$. A single ridge in a triangle mesh (a) is subdivided with $\sqrt{3}$-subdivision (b). The mean curvature in the limit is plotted in (c). The irregularity of the curvature pattern about the ridge is the lateral artifact.

While the development of a sophisticated non-stationary subdivision scheme which eliminates (or at least minimises) artifacts in the smoothed animation manifold, alternative smoothing and refinement approaches could also be explored for smoothing general dimensional manifolds.

10.3 Visualisation

One of the most difficult aspects of working with manifolds in $\mathbb{R}^4$ is visualising its behaviour. Interaction is even more challenging, as while a rendered scene is inherently 3D, editing on a standard commodity PC is typically restricted to two dimensions (in the plane of the display). This problem is not isolated to animation manifolds, as existing animation specification packages will suffer from similar problems.

Certain features have been developed which assist animators in defining the paths of animation. Vertex path visualisation, for example, is particularly useful to display and edit motion. This feature has an analogy on animation manifolds in the form of a geodesic between two points which the user deems to be representing the same feature. The development of these features is an important area for future work, not just for the improvement of animation manifolds, but for animation visualisation in general.

10.3.1 Eliminating jagged edges

The linearity of the animation manifold structure may result in some rendering artifacts, such as jagged edges, or unnatural shortening or lengthening of features in interpolated frames. Jagged edges can, to a degree, be disguised using the smooth vertex normals presented in Section 2.3 as can be seen from Figure 2.5, but the silhouette may still appear jagged. A method to determine the sampling density of keyframes necessary to avoid these jagged edges would be of keen interest.

An alternative approach could be to connect a very dense sampling of keyframes and apply a simplification algorithm such as a higher dimensional generalisation of the method of Garland and Heckbert [1997]. This approach would naturally simplify "flat" regions (regions which do not contribute a significant feature to the animation) while maintaining animation features, forcing features to follow the flow of the animation and minimising distortion.

10.3.2 Alternative rendering solutions

Programmable graphics hardware offers several avenues for alternative methods for rendering animation manifolds in real-time. While I have not implemented them, these approaches would improve surface smoothness, while maintaining an interactive framerate, and represent exciting avenues for future work.

- Subdivision is one of the "killer applications" for programmable graphics hardware, promising smoother surfaces without overburdening the bottleneck between graphics card and system memory. Shiue et al. [2005] introduced a viable subdivision kernel for use with current GPU's. A GPU program combined with the hardware isosurface extraction method of Section 2.6 is feasible, assuming that sufficient instructions are available on the graphics card.

- Real-time rendering of Bezier tetrahedra was demonstrated by Loop and Blinn [2006] and it promises exciting prospects for the rendering of smooth isosurfaces from volumetric tetrahedral data sets in real-time.

List of Symbols

$\mathcal{A}$	An animation manifold, and the boundary of the volume A.
$\mathbf{b} = \{\gamma_i\}$	Where $i = 1 \dots n$. Barycentric coordinates representing a position in a polygon from its n points.
CDT	A constrained Delaunay triangulation operator. Given a point P and a constraint set $\mathcal{X}$ this returns a constrained Delaunay triangulation.
$\mathrm{C}_F\mathrm{DT}$	A conforming Delaunay triangulation operator. Given a point P and a constraint set $\mathcal{X}$ this returns a conforming Delaunay triangulation.
CH	An operator returning the convex hull of a set of points.
DT	The Delaunay triangulation operator. Given a point set P, this returns a Delaunay triangulation.
$\mathcal{M}$	A manifold consisting of a simplicial n-set in $\mathbb{R}^{n+1}$.
$\mathbb{R}^n$	An n-dimensional space with real valued coordinates.
VS_{in}	The internal Voronoi skeleton.
VS_{out}	The external Voronoi skeleton.
$\mathcal{X}$	A possibly non-homogenous simplicial set which is used as a set of constraints.

Glossary

A

animation manifold ($\mathcal{A}$) A n-manifold embedded in $\mathbb{R}^{n+1}$ for which one of the dimensions is time. Rendering an isosurface extracted from the animation manifold with the time component fixed yields a frame from the animation sequence. It will typically be referred to as $\mathcal{A}$ in the text., p. 12.

B

barycentric coordinates ($\mathrm{b} = \{\gamma_i\}, i = 1 \ldots n$) A point within a polygon can be defined as a weighted sum of the vertices of the polygon. These weights are called barycentric coordinates. If the polygon is convex (or a simplex) then for points within the polygon, all weights are *positive* and *sum to unity.*, p. 65.

bisector (b_{ij}) A bisector b_{ij} is a set of points equidistant from two point p_i and p_j. These form the boundaries of Voronoi cells., p. 70.

C

conforming Delaunay triangulation ($\mathcal{M} = \mathrm{C}_F\mathrm{DT}(P, \mathcal{X})$) A triangulation $\mathcal{M}$ of a point set is conforming Delaunay if for a set of input constraints $\mathcal{X}$, $\mathcal{X} \subset \mathcal{M}$ *and* each simplex in $\mathcal{M}$ satisfies the Delaunay condition. It may also be made to fulfil some prerequisite quality criteria., p. 53.

connected components A connected component of a manifold is one from which any other vertex in that component can be reached by traversing edges on the manifold. Thus a triangle mesh consisting of two disconnected spheres has two connected components., p. 16.

constrained Delaunay triangulation ($\mathcal{M} = \mathrm{CDT}(P, \mathcal{X})$) A triangulation $\mathcal{M}$ of a point set is constrained Delaunay if for a set of input constraints $\mathcal{X}$, $\mathcal{X} \subset \mathcal{M}$. Note that these constraints may cause one or more simplices of $\mathcal{M}$ to violate the Delaunay condition. It may also be made to fulfil some prerequisite quality criteria., p. 14.

constructive solid geometry (CSG) A shape representation which consists of a hierarchy of set operations applied to geometric primitives., p. 13.

D

Delaunay condition A simplex s formed from a point set is locally Delaunay if no point lies within the circumsphere bounding s. A simplex is *strongly* Delaunay if no point lies on or inside the circumsphere bounding s., p. 53.

Delaunay triangulation ($\mathcal{M} = \mathrm{DT}(P)$) A triangulation of a point set is Delaunay if each simplex satisfies Delaunay condition., p. 14.

G

geodesic The shortest path between two points in space. In this context, that space is a manifold., p. 117.

H

hypertree A general dimensional equivalent of the quadtree structure., p. 21.

I

iso-contour An extracted $n-1$-manifold extracted from a n-manifold from a particular cross-sectional plane. The cross-sectional plane for an animation volume is typically the time plane., p. 16.

iso-surface A iso-contour which is a surface., p. 16.

K

keyframe A keyframe in animation is a rigid contour which defines the start and/or end of an interpolating animation sequence., p. 11.

M

Manhattan distance For an edge $(x_0, y_0), (x_1, y_1)$ the Manhattan distance is an approximation of the edge length given by $|x_1 - x_0| + |y_1 - y_0|$. This removes the need for any multiplication or square root operations., p. 44.

manifold ($\mathcal{M}$) An orientable space in which the neighbourhood of every point is topologically equivalent to a disc. A simplicial mesh $\mathcal{M} = \{P, \mathcal{F}\}$, consisting of point set P and simplices $\mathcal{F}$ is referred to as a *piecewise manifold.* A manifold may be closed or may have a boundary., p. 16.

medial axis The medial axis of a closed surface is the set of centres of empty balls which touch the surface at more than one point. It is a topological skeleton., p. 71.

metaballs An implicit representation for surfaces, introduced by Wyvill et al. [1986], which support complex topological and geometric operations. Results from using this method are traditionally "blobby-looking"., p. 13.

Q

quadtree An iterative spacial partitioning and sorting algorithm for scenes in $\mathbb{R}^2$. Objects or facets in the scene are assigned to spacial cells, the cells are recursively subdivided if necessary and the process is repeated., p. 21.

R

ray tracing A rendering technique where rays originating from the viewer are traced and the interactions with objects and light sources in the scene are accumulated to produce an image., p. 21.

S

simplex An n-simplex is defined as the convex hull of a set of $(n+1)$ affine independent points in some Euclidean space of dimension n or higher. A simplex is said to be *full space* if it fills space, i.e. is an n-simplex in $\mathbb{R}^n$. The number prefix is referred to as the *type* of the simplex. Edges, triangles and tetrahedra are examples of simplices., p. 94.

simplicial complex A simplicial n-complex $\mathcal{K}$ is a set of simplices with the property that any face of a simplex in $\mathcal{K}$ is also in $\mathcal{K}$., p. 53.

simplicial set A simplicial n-set is a set of simplices with maximum type n. This differs from the simplicial complex in that faces of a simplex need not be present in the set. It is set to be *homogenous* if it only consists of n-simplices. It is typically represented by a caligraphic symbol (e.g. $\mathcal{M}$) if it represents a boundary, or a Roman symbol (e.g. M) if it is a full space simplicial set., p. 119.

skeleton A skeleton of a contour is a simplified structure which is topologically equivalent to the contour. The medial axis is an example of a skeleton.

surface deformation A feature sensitive method technique for surface editing. Often user interaction is modeled after intuitive real–world sculpting metaphors such as clay modelling., p. 13.

T

topological alteration Altering the topology of a manifold which may result in the change of its topological genus., p. 11.

topological genus The topological genus (family) of a space is derived from the first Betti number, which is the maximum number of cuts that can be made without dividing the space into two pieces., p. 122.

topology Informally, topology describes *the properties of and the nature of space.* This includes the compactness, connectedness and countability.

tweening Interpolating between frames of an animation, typically a hand drawn cartoon., p. 26.

V

Voronoi cell (C_i) The space C_i surrounding a point $p_i \in P$ such that any point in C_i is *closer* to p_i than any other point in P according to some distance function., p. 69.

Voronoi diagram A structure containing the spatial partitioning of a point set P based on the Voronoi cells. It is the dual of the Delaunay triangulation of P., p. 14.

Voronoi separable A mesh $\mathcal{M}$ is Voronoi separable if facets of the manifold $\mathcal{M}$ which intersect the Voronoi cell boundary ∂C_i contain the vertex p_i., p. 73.

Voronoi skeleton (VS) The set of bisectors of the Voronoi diagram of a Voronoi separable manifold $\mathcal{M}$ which lie within the *convex hull* of $\mathcal{M}$ and do not intersect the faces of mesh $\mathcal{M}$. This is further separated into the *internal* and *external* skeletons (denoted by VS_{in} and VS_{out} respectively) which indicates whether the portion of the skeleton lies inside or outside of $\mathcal{M}$., p. 74.

Voronoi vertex A vertex at which at least three bisectors intersect., p. 70.

Bibliography

Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gätner. A novel type of skeleton for polygons. *Journal of Universal Computer Science*, 1(12):752?–761, 1995.

Tomas Akenine-Möller and Eric Haines. *Real-time rendering.* AK Peters, 2 edition, July 2002.

Marc Alexa. Recent advances in mesh morphing. *Computer Graphics Forum*, 21(2): 173–196, 2002.

N. Amenta, S. Choi, and R. Kolluri. The power crust, unions of balls, and the medial axis transform. *Computational Geometry*, 19(2–3):127–153, 2001.

Alexis Angelidis, Marie-Paule Cani, Geoff Wyvill, and Scott King. Swirling-sweepers: Constant volume modeling. In *Proceedings of Pacific Graphics*, oct 2004.

Dominique Attali, Jean-Daniel Boissonnat, and Herbert Edelsbrunner. Stability and computation of medial axes: a state of the art report. In T. Möller, B. Hamann, and B. Russell, editors, *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration.* Springer–Verlag, Mathematics and Visualization, 2007. URL `http://cgal.inria.fr/Publications/2007/ABE07`.

M. Attene and B. Falcidieno. ReMESH: An interactive environment to edit and repair triangle meshes. In *Shape Modeling Internationl*, pages 271–276. IEEE Computer Society Press, 2006.

O.K.-C. Au, C.L. Tai, L. Liu, and H. Fu. Dual laplacian editing for meshes. *IEEE Transactions on Visualization and Computer Graphics*, 12(3):386–395, May-June 2006.

F. Aubert and D. Bechmann. Animation by deformation of space-time objects. *Computer Graphics Forum*, 16(3):57–66, September 1997.

Chandrajit L. Bajaj, Edward J. Coyle, and Kwun-Nan Lin. Arbitrary topology shape reconstruction from planar cross sections. *Graphical models and image processing: GMIP*, 58(6):524–543, 1996. URL `citeseer.ist.psu.edu/bajaj96arbitrary.html`.

C.B. Barber, D.P. Dobkin, and H.T. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, 22(4):469–483, December 1996. http://www.qhull.org.

Gill Barequet, Michael T. Goodrich, Aya Levi-Steiner, and Dvir Steiner. Straight-skeleton based contour interpolation. In *The Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 119–127, 2003.

J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

H. Blum. A transformation for extracting new descriptors of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*. MIT Press, 1967.

Jean-Daniel Boissonnat. Shape reconstruction from planar cross sections. *Comput. Vision Graph. Image Process.*, 44(1):1–29, 1988. ISSN 0734-189X.

Jean-Daniel Boissonnat and Monique Teillaud, editors. *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, 2007.

M. Bóo, M. Amor, M. Doggert, J. Hirche, and W. Strasser. Hardware support for adaptive subdivision surface rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on on Graphics hardware*, pages 33–40, 2001.

M. Botsch and L. Kobbelt. An intuitive framework for real-time freeform modeling. In *Proceedings of SIGGRAPH*, pages 630–634, 2004.

S. Brandel, D. Bechmann, and Y. Bertrand. Stigma: a 4-dimensional modeller for animation. In *Workshop on Animation and Simulation, Eurographics*, September 1998.

D. Breen and R. Whitaker. A level-set approach for the metamorphosis of solid models. *IEEE Trans. on Visualization and Computer Graphics*, 7(2):173–192, 2001.

Hamish Carr, Torsten Moller, and Jack Snoeyink. Artifacts caused by simplicial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):231–242, March/April 2006.

CGAL. CGAL user and reference manual, 2007. URL `http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/packages.html`. CGAL Editorial Board.

L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pages 274–280. ACM, May 1993.

P. Cignoni, C. Montani, and R. Scopigno. Dewall: a fast divide and conquer delaunay triangulation algorithm. *Computer-Aided Design*, 30(5):333–341, April 1998.

Daniel Cohen-Or, Amira Solomovic, and David Levin. Three-dimensional distance field metamorphosis. *ACM Transactions on Graphics*, 17(2):116–141, 1998.

David Cohen-Steiner, Éric Colin de Verdiére, and Mariette Yvinec. Conforming delaunay triangulations in 3d. *Special issue on the 18th annual symposium on computational geometry*, 28(2–3):217–233, June 2004.

Tim Culver, John Keyser, and Dinesh Manocha. Accurate computation of the medial axis of a polyhedron. In *Symposium on Solid Modeling and Applications*, pages 179–190, 1999.

Tim Davis. Sparse matrix algorithms research at the university of florida. http://www.cise.ufl.edu/research/sparse/, February 2006.

Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, and Alan H. Barr. Dynamic real-time deformations using space and time adaptive sampling. In Eugene Fiume, editor, *Proceeding of SIGGRAPH*, pages 31–36. ACM Press / ACM SIGGRAPH, 2001.

Boris N. Delaunay. Sur la sphre vide. *Izvestia Akademia Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7:793–800, 1934.

Mathieu Desbrun and Marie-Paule Cani. Active implicit surface for animation. In *Graphics Interface*, pages 143–150, June 1998.

Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. *Proceedings of SIGGRAPH*, pages 317–324, 1999.

T. K. Dey and S. Goswami. Tight cocone: A water tight surface reconstructor. In *ACM Symposium of Solid Modeling Applications*, pages 127–134, 2003.

T. K. Dey and W. Zhao. Approximate medial axis as a voronoi subcomplex. In *ACM Symposium of Solid Modeling Applications*, pages 356–366, 2002.

N. Dyn, D. Levin, and J.A. Gregory. 4-point interpolatory subdivision scheme for curve design. *Computer Aided Geometric Design 4*, pages 257–268, 1987.

H. Edelsbrunner. The union of balls and its dual shape. In *ACM Symposium on Computational Geometry*, pages 218–231, 1993.

Herbert Edelsbrunner. *Geometry and Topology for Mesh Generation.* Cambridge University Press, 2001.

Herbert Edelsbrunner and Raimund Seidel. Voronoi diagrams and arrangements. *Discrete and Computational Geometry*, 1:25–44, 1986.

Herbert Edelsbrunner and Nimish R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15(3):223–241, March 1996.

Eric Fausett, Alexander A. Pasko, and Valery Adzhiev. Space-time and higher dimensional modeling for animation. In *IEEE Computer Animation*, pages 140–145, 2000.

Adam Finkelstein, Charles E. Jacobs, and David H. Salesin. Multiresolution video. In *Proceedings of SIGGRAPH*, pages 281–290, 1996.

J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 2^{nd} edition, July 1997.

James Gain and Neil Dodgson. Adaptive refinement and decimation under free-form deformation. In *Eurographics UK*, pages 13–15, 1999.

Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. *Proceedings of SIGGRAPH*, pages 209–216, 1997.

B. Geiger. Three-dimensional modeling of human organs and its application to diagnosis and surgical planning. Technical Report RR-2105, INRIA, 1993. URL `citeseer.ist.psu.edu/geiger93threedimensional.html`.

A. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

A. Glassner. Spacetime raytracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, March 1988.

N. I. M. Gould, Y. Hu, and J. A. Scott. Complete results from a numerical evaluation of sparse direct solvers for the solution of large, sparse, symmetric linear systems of equations. Technical report, Numerical Analysis Group, CCLRC, ftp://ftp.numerical.rl.ac.uk/pub/reports/ghsNAGIR20051r1.pdf, December 2005.

N. Grislain and J. Shewchuk. The strange complexity of constrained delaunay triangulation. In *Proceedings of the Fifteenth Canadian Conference on Computational Geometry*, pages 89–93, August 2003.

Brian Guenter. Efficient symbolic differentiation for graphics applications. *ACM Transactions on Graphics*, 26(3):108, 2007.

I. Guskov, W. Sweldens, and P. Schröder. Multiresolution signal processing for meshes. In *Proceedings of SIGGRAPH*, pages 325–334, 1999.

H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH*, pages 99–108, 1996.

Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk, and Jack Snoeyink. Streaming computation of delaunay triangulations. In *Proceedings of SIGGRAPH*, pages 1049–1056, 2006.

T. Kay and J. T. Kajiya. Ray tracing complex scenes. In *Proceedings of SIGGRAPH*, pages 269–278, July 1986.

S. Kircher and M. Garland. Progressive multiresolution meshes for deforming surfaces. In *Proceedings of ACM Symposium on Computer Animation*, pages 191–200, 2005.

Scott Kircher and Michael Garland. Editing arbitrarily deforming surface animations. In *Proceedings of SIGGRAPH*, pages 1098–1107, 2006.

Allison W. Klein, Peter-Pike J. Sloan, Adam Finkelstein, and Michael F. Cohen. Stylized video cubes. In *Proceedings of ACM Symposium on Computer Animation*, pages 15–22, July 2002.

B. Klimmek, H. Prautzsch, and N. Vahrenkamp. Shadow metamorphosis. *Computing*, 79 (2–4):2007, April 2007.

Leif Kobbelt. Interpolatory subdivision on open quadrilateral nets with arbitrary topology. *Computer Graphics Forum*, 15(3):409–420, August 1996.

Leif Kobbelt, Swen Campagna, Jens Vorsatz, and Hans-Peter Seidel. Iteractive multi-resolution modeling on arbitrary meshes. *Proceedings of SIGGRAPH*, pages 105–114, July 1998.

D. T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Parallel Programming*, 9(3):219–242, June 1980.

P. Leinhardt. Subdivision of n-dimensional spaces and n-dimensional generalized maps. In *Symposium on Computational Geometry*, pages 228–236. ACM, 1989.

Yaron Lipman, Olga Sorkine, David Levin, and Daniel Cohen-Or. Linear rotation-invariant coordinates for meshes. In *Proceedings of SIGGRAPH*, pages 479–487, 2005.

Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. In *Proceedings of SIGGRAPH*, pages 664 – 670, 2006.

H. A. Lorentz, A. Einstein, H. Minkowksi, and H. Weyl. *The Principle of Relativity: A Collection of Original Memoirs.* Dover, 1952.

M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and Mathematics III*, pages 35–57, 2003.

Gary L. Miller, Steven E. Pav, and Noel J. Walkington. When and why Ruppert's algorithm works. In *Proceedings of the 12th International Meshing Roundtable*, pages 91–102. Sandia National Laboratory, September 2003.

M. Murphy, D. M. Mount, and C. W. Gable. A point-placement strategy for conforming delaunay tetrahedralizations. *International Journal of Computational Geometry and Applications*, 11(6):669–682, 2001.

Andrew Nealen, Matthias Mller, Richard Keiser, Eddy Boxerman, and Mark Carlson. Physically based deformable models in computer graphics: State of the art report. *Computer Graphics Forum*, 25(4):809–836, 2005.

R. Ogniewicz and M. Ilg. Voronoi skeletons: Theory and applications. In *Computer Vision and Pattern Recognition*, pages 63–69, June 1992.

John Page. Thales' theorem. http://www.mathopenref.com/thalestheorem.html, January 2008.

U. Pinkall and K. Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Math*, 2(1):15–36, 1993.

Frank Reck, Carsten Dachsbacher, Marc Stamminger, Günther Greiner, and Roberto Grosso. Realtime isosurface extraction with graphics hardware. In *Proceedings of Eurographics (short paper)*, pages 1–4, 2004.

J. Ruppert and R. Seidel. On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra. In *Proc. 5th Annual Symposium on Computational Geometry*, pages 380–393. ACM, 1989.

Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

M. A. Sabin and L. Barthe. Artifacts in recursive subdivision surfaces. In *Proceedings of the Fifth International Conference on Curves and Surfaces (St. Malo)*, pages 353–362, June 2002.

Malcolm Sabin. Personal communication. Computer Laboratory, Cambridge, August 2007.

S. Schaefer, J. Hakenberg, and J. Warren. Smooth subdivision of tetrahedral meshes. In *Symposium on Geometry Processing*, pages 147–154. Eurographics/ACM SIGGRAPH, 2004.

E. Schönhardt. Über die zerlegung von dreieckspolyedern in tetraeder. *Mathematische Annalen*, 98:309–312, 1928.

A. Shamir and V. Pascucci. Temporal and spatial level of details for dynamic meshes. In *Proceedings of ACM Sypmposium on virtual reality software and technology*, pages 77–84, 2001.

Damian J. Sheehy, Cecil G. Armstrong, and Desmond J. Robinson. Shape description by medial surface construction. *IEEE Trans. Vis. Comput. Graph.*, 2(1):62–72, 1996.

A. Sheffer and V. Krayevoy. Pyramid coordinates for morphing and deformation. In *3D Data Processing, Visualization and Transmission*, pages 68–75, 2004.

Dinggang Shen and Christos Davatzikos. Measuring temporal morphological changes robustly in brain mr images via 4-dimensional template warping. *NeuroImage*, 21(4): 1508–1517, April 2004.

Dinggang Shen, Hari Sundar, Zhong Xue, Yong Fan, and Harold Litt. *Lecture Notes in Computer Science*, volume 3750/2005, chapter Consistent Estimation of Cardiac Motions by 4D Image Registration, pages 902–910. Springer Berlin / Heidelberg, September 2005.

Jonathan Richard Shewchuk. Constrained delaunay tetrahedralizations and provably good boundary recovery. In *Eleventh International Meshing Roundtable (Ithaca, New York)*, pages 193–204. Sandia National Laboratories, September 2002.

Jonathan Richard Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering (ACM Workshop on Applied Computational Geometry)*, pages 203–222. Springer-Verlag, Berlin, May 1996.

J.R. Shewchuk. *Delaunay Refinement Mesh Generation.* PhD thesis, Carnegie Mellon University, 1997. Available as Techn.Rep. CMU-CS-97-137.

J.R. Shewchuk. A condition guaranteeing the existence of higher-dimensional constrained delaunay triangulations. In *Proc. 14th Annual Symposium on Computational Geometry*, pages 76–85. ACM, 1998.

Le-Jeng Shiue, Ian Jones, and Jörg Peters. A realtime GPU subdivision kernel. In *Proceedings of SIGGRAPH*, pages 1010 – 1015, July 2005.

H. Si and K. Gaertner. Meshing piecewise linear complexes by constrained delaunay tetrahedralizations. In *Proceedings of the 14th International Meshing Roundtable*, pages 147–163, September 2005.

Hang Si. Personal communication. e–mail, March 2007.

Julian Smith. Personal communication. Computer Laboratory, Cambridge, August 2007.

Olga Sorkine. State of the art report: Laplacian mesh processing. In *Proceedings of Eurographics*, pages 53–70, 2005.

Olga Sorkine and Daniel Cohen-Or. Least-squares meshes. In *Proceedings of Shape Modeling International.* IEEE Computer Society Press, 2004.

Olga Sorkine, Yaron Lipman, Daniel Cohen-Or, Marc Alexa, Christian Rössl, and Hans-Peter Seidel. Laplacian surface editing. In *Symposium on Geometry Processing*, pages 179–188. Eurographics/ACM SIGGRAPH, 2004.

Abhijit Sovakar and Leif Kobbelt. API design for adaptive subdivision schemes. *Computers and Graphics*, 28(1):67–72, 2004.

G. Taubin. Dual mesh resampling. In *Pacific Graphics*, pages 94–113, 2001.

Gabriel Taubin. A signal processing approach to fair surface design. *Proceedings of SIGGRAPH*, pages 351–358, August 1995.

Greg Turk and James O'Brien. Shape transformation using variational implicit functions. In *Proceedings of SIGGRAPH*, pages 335–342, August 1999.

J. Vleugels and M.H. Overmars. Approximating voronoi diagrams of convex sites in any dimension. *Int. J. of Comp. Geom. and Appl.*, 8:201–221, 1998.

Wolfram von Funck, Holger Theisel, and Hans-Peter Seidel. Vector field based shape deformations. In *Proceedings of SIGGRAPH*, pages 1118–1125, 2006.

A. Witkin and M. Kass. Spacetime constraints. In *Proceedings of SIGGRAPH*, pages 159–168, 1988.

G. Wyvill, C. McPhetters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2:227–234, 1986.

Shin Yoshizawa, Alexander Belyaev, and Hans-Peter Seidel. Skeleton-based variational mesh deformations. In *Eurographics*, pages 3–7, 2007.

Kun Zhou, Jin Huang, John Snyder, Xinguo Liu, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Large mesh deformation using the volumetric graph laplacian. In *Proceedings of SIGGRAPH*, pages 496–503, July 2005.

D. Zorin, P. Schröder, and W. Sweldens. Interactive multiresolution mesh editing. In *Proceedings of SIGGRAPH*, pages 259–268, 1997.

Dennis Zorin, Peter Schröder, and Wim Sweldens. Interpolating subdivision for meshes with arbitrary topology. In *Proceedings of SIGGRAPH*, pages 189–192, 1996.