Curvature Dependent Implicit Surface Tiling

by

Tasso Karkanis

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Curvature Dependent Implicit Surface Tiling

Tasso Karkanis
Master of Science
Graduate Department of Computer Science
University of Toronto
1999

In order to efficiently render implicit surfaces with a projective rendering system, a polygonization must be computed. Traditionally there have been two approaches to this problem: cell–based polygonization and particle methods. The former approach is fast, but can produce degenerate triangles, and requires a grid size to be specified which can miss small features. The latter approach can be computationally expensive.

This work is an implicit surface triangulator that quickly generates near equilateral triangles with sizes dependent on the local curvature of the surface. First, a seed triangle is computed. Then this polygonization is extended by adding adjacent triangles according to various rules which are based on the geometry of neighbouring triangles and an estimate of the local curvature of the surface. Finally, a set of heuristics is applied to fill the untiled seams remaining in the surface.

Acknowledgements

I thank my supervisor, James Stewart, for his guidance and assistance. I also thank the members of the Graphics Lab, especially Petros Faloutsos, Joe Laszlo, Reimar Schubert, and Glenn Tsang for their friendship and support. Finally, I thank Keith Kim for his insight, patience, interest, and assistance.

Contents

Chapter 1

Introduction

Computer rendering almost always involves the depiction of solid objects, which are typically represented by their surfaces. Applications of computer rendering are numerous, including computer aided design (CAD), scientific visualization, special effects in film and television, and digital games.

CAD is usually applied to solid objects like machinery and buildings. The representations of these objects must be very precise as the computer model is used to build the objects. The designer models the objects by defining all the surfaces and the CAD package renders realistic images of the model from this description.

Scientific visualization provides the scientist or student with insight and intuition about natural phenomena. A common application in physical chemistry is the representation of molecules. Atoms and the nuclear bonds between them are represented by surfaces to indicate the geometric structure of the molecule. These surfaces are typically isosurfaces of the electron density function.

In order to render a surface, the geometric and optical properties of the surface must be modelled. A useful representation of the geometry of a surface should easily provide information about the coordinates of points on the surface, and the normal vector at any point on the surface. The normal vector, which is perpendicular to and points away from the surface, is used for various geometric calculations, including lighting calculations to shade the surface. This thesis presents an algorithm to convert from an "implicit" representation that is easily manipulated to a "polyhedral" representation that is easily rendered.

1.1 Surface Representations

The most direct representation of a surface is a collection of polygons (planar shapes delimited by straight line segments). Polygons are simply and efficiently rendered by conventional graphics systems; the vertices of each polygon are projected onto an "image plane," the result is clipped by the border of the image, and the pixels corresponding to any visible portions are drawn. All the algorithms necessary for such a rendering system can be implemented cheaply in hardware. For example, current midrange personal computer graphics cards can render 15 million polygons per second.

Polyhedral models, which represent solid objects entirely with polygons, are especially appropriate for predominantly rectilinear objects such as buildings. Normal vectors to polygons are easily computed,

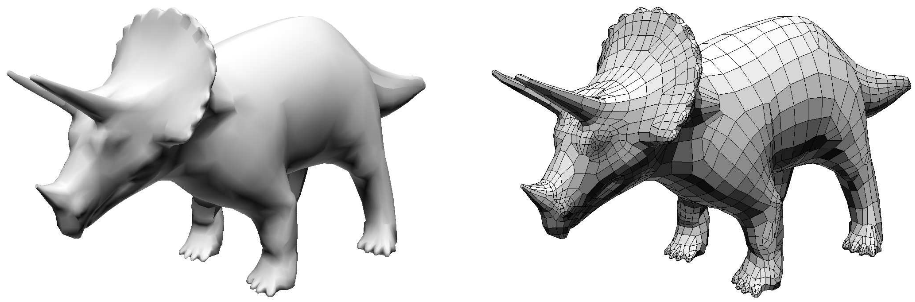

Figure 1.1: A polyhedral model of a triceratops, rendered smoothly shaded, and flat shaded with the polygons outlined. This model is used courtesy of Viewpoint Animation Engineering.

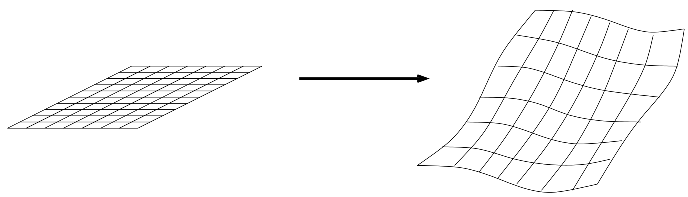

Figure 1.2: A section of the plane mapped to a surface in 3D.

and when the shape represented by the model is curved, the normal at vertices is calculated from the normals to the adjacent polygons (see Figure 1.1). Unfortunately, accurately modelling a curved surface with polygons is difficult and tedious.

A more powerful representation uses **parametric functions**. The set of points on the surface is given by 3−vector valued function of two parameters. For example, a sphere may be defined as:

$$
\begin{aligned}
x(\theta, \phi) &= R\cos(\phi)\cos(\theta), \\
y(\theta, \phi) &= R\sin(\phi)\cos(\theta), \\
z(\theta, \phi) &= R\sin(\theta)
\end{aligned}
$$

where R is the radius. By varying ϕ and θ over the parameter space $[0, 2\pi] \times [-\pi, \pi]$, all points on the sphere may be enumerated.

In practice, the components of the parametric functions are usually third order polynomials in the parameters (such as B–splines). Under these functions, a rectangular region of the parameter space maps to a 3D curved surface known as a patch. Complex surfaces are composed of seamless concatenations of patches.

A parametric representation can represent curved surfaces more accurately than polygons can. Normals can be calculated easily from the cross product of the partial derivatives, and all the points on the surface can be enumerated by varying the parameters. However, modelling with this representation is

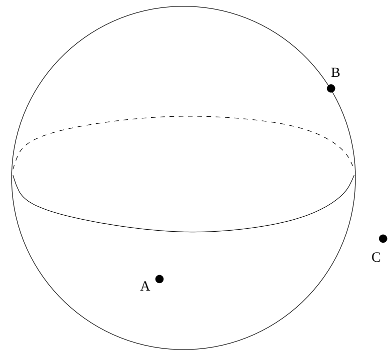

Figure 1.3: An implicit sphere model. Point A is inside the surface where f has a negative value, point B is on the surface where f is zero, and point C is outside the surface were f is positive.

not always intuitive as it involves manipulating the coefficients of the parametric function and not the surface itself. This problem represents an active area of research.

Subdivision surfaces use another representation that has received much attention recently. In this representation, the surface is defined by a small number of polygons and a set of rules that specify how to recursively subdivide these polygons. Useful sets of rules, such as the Catmull–Clark subdivision, cause this procedure to converge to a smooth surface. It has recently been shown that the Catmull–Clark subdivision scheme can also be evaluated directly like parametric surfaces.

Surfaces can also be defined with **implicit functions**. An implicit function $f(x, y, z)$ can be thought of as a scalar or potential field. A real value is assigned by f to every point in space. When this function is continuous, the set of points such that $f(x, y, z) = c$ where c is some constant defines a surface. By convention, c is taken to be zero. Points where f has a negative value are considered to be within the solid delimited by the surface and points with positive values are without. The sphere in the example above could be equivalently defined implicitly with the function:

$$f(x, y, z) = x^2 + y^2 + z^2 - R^2$$

The normal to an implicit surface is readily calculated as the gradient of the function. Implicit surfaces also lend themselves well to rendering by a technique called ray tracing. Most importantly, modelling with implicit surfaces promises to be simpler and more intuitive than with the other representations, especially when the general shape of the object being modelled is more important than high precision in the geometry. Techniques have been developed which allow the user to manipulate a surface easily, and to sculpt an implicit surface using a modelling primitive called Meta Balls [35].

1.2 Goals of this Thesis

Although the implicit surface representation is concise and useful for modelling, it does have some drawbacks. For texture mapping, a technique in which images are stretched over a surface analogous to wallpapering, a parametrization of the surface is required. Polyhedral and parametric models have natural parametrizations, but implicit surfaces do not.

Polygonal models are convenient for some techniques like collision detection because overlap of polyhedral objects is computed more easily than the intersections of parametric or implicit functions [4]. Also, real–time and hardware rendering systems accomodate only polygons. It is therefore useful if a non–polygonal surface representation can be converted to polygons. Parametric surfaces can be very easily polygonized, but polygonizing (or "tiling") implicit surfaces is much more complicated.

In this thesis, a new algorithm is presented which solves the implicit surface polygonization problem by tiling the surface with triangles. In order to represent the surface accurately, the tiling is curvature dependent; in areas of high curvature, the triangles are smaller and the resulting tiling is denser. Curvature–dependence is necessary to capture small detail on the surface while keeping the total number of triangles in the tiling low, which is important for storage and rendering efficiency.

This algorithm produces close–to–equilateral triangles, which prevent shading artifacts during rendering, and which are better for certain geometrical computations (e.g. because finite–element methods solve faster and more accurately with well–shaped triangles).

The polygonizer of this thesis has three requirements: the implicit function, a function which returns the normal to the implicit surface at a given point, and a bounding box that contains the surface. The polygonizer currently handles only differentiable surfaces which, intuitively, have no sharp edges or points.

1.3 Other Approaches

There are primarily three other approaches to the problem of tiling an implicit surface. These algorithms are compared with the new approach in the next chapter. The first approach is similar to the volume rendering algorithm with the picturesque name "marching cubes." In that algorithm, the implicit function is sampled at every point on a regular grid. Each cell of this grid is then classified according to whether the corners fall inside or outside the surface. This classification determines the polygonization within the cell.

Another approach is iterative and particle–based. A set of particles restricted by mathematical constraints is allowed to cover the surface in a procedure analogous to electrical charge spreading over the surface of a conducting solid. The constraints force the particles to lie on the surface and control the distance between and orientation of the particles. Once the configuration of the particles is quiescent, a polygonization is computed via a well–understood technique involving the Voronoi diagram of the particle set.

The third method, called shrinkwrapping, works by creating a mesh for a sphere containing the entire implicit surface. Vertices on this mesh are then iteratively moved closer to the surface, and the mesh is split if the surface consists of several components, or punctured where the surface has a hole.

1.4 Outline of the Algorithm

In broad strokes, the algorithm presented in this thesis works by finding a seed triangle of appropriate size somewhere on the surface, then recursively "growing" more triangles attached to the seed, and finally filling in the spaces left by the growing procedure.

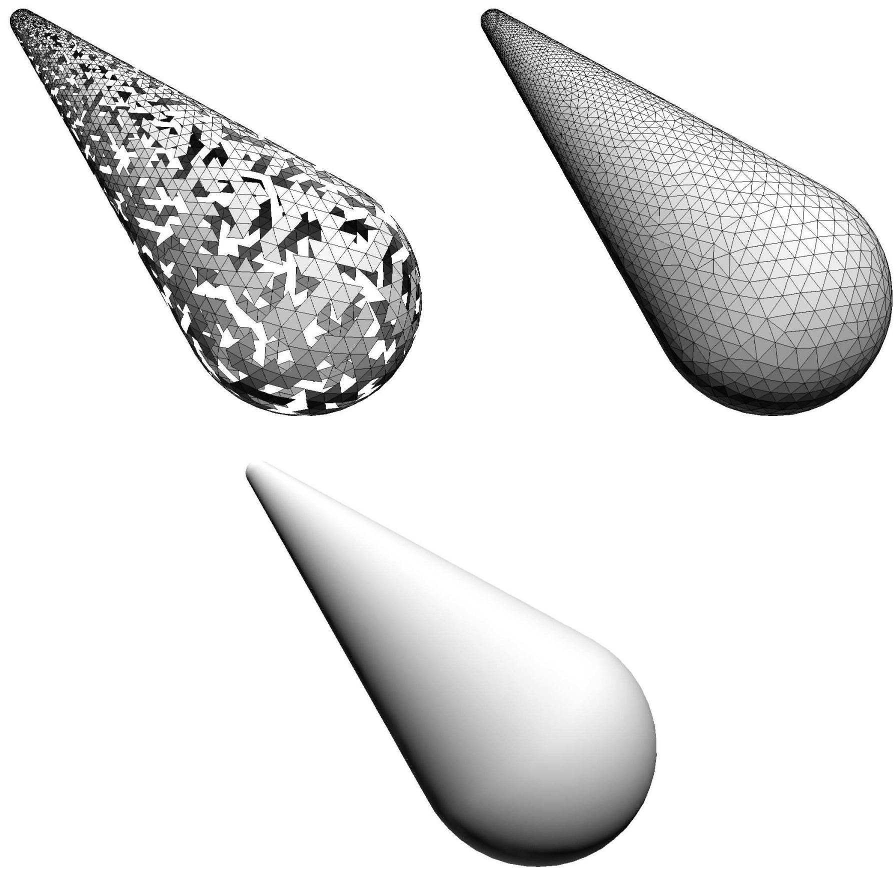

Figure 1.4: A model of a cone with a rounded apex. The three pictures show the mesh produced by the triangle growing phase, the final mesh after the gap is filled, and the final mesh with smooth shading.

To compute a seed triangle, a single point is first located on the surface. Then the curvature of the surface near this point is estimated and a triangle is formed around this point with edge lengths related to the curvature.

During the triangle growing phase, a list of triangle edges on the boundary of the tiling is maintained. For each such edge, a candidate triangle using that edge, again sized according to curvature, is computed to extend the tiling. To avoid overlapping triangles, only those candidates far enough away from existing triangles are used to extend the tiling.

The "gap" in the **tiling** (i.e. the current set of triangles) left by the growing phase can be thought of as a single non–planar polygon. The first attempt to fill the gap involved building the medial axis of this polygon. The medial axis (also known as "skeleton") of a polygon is a set of curves and lines contained in the polygon, reminiscent of the veins in a leaf. The medial axis encapsulates the structure of the polygon, so the vertices on the medial axis could be used to generate triangles to fill the gap. This approach was eventually abandoned due to its mathematical complexity.

The approach developed to fill the gap was born of experimentation and observation. Certain geometric patterns are identified in the gap. Then the gap is divided into smaller pieces and these pieces are individually triangulated with a set of heuristics.

Chapter 2

Previous Work

This chapter presents a survey of related work in the computer graphics literature. Some of this literature concerns other problems which must be solved in order for implicit surface techniques to be of practical use. Most of the previous approaches to implicit surface tiling fall into one of two categories: cell–based tilers and particle system tilers. These approaches are discussed and compared. Finally, the results of this work are briefly presented and it is compared to previous algorithms.

2.1 Related Problems

The problem of tiling implicit surfaces is surrounded by a number of similar and related problems. Interest in implicit surfaces as applied to computer graphics was sparked by Blinn's famous "blobby" paper [3]. In this paper, a scan–line algorithm for rendering implicit surfaces is presented. More importantly, solid modelling with implicit surfaces via the exponential potential is introduced. Models are created by placing "blobs" at a number of points in space. Each blob has a potential given by:

$$P_i(x, y, z) = \exp\left(\frac{B_i}{R_i^2} r^2 - B_i\right)$$

where $r = \sqrt{x^2 + y^2 + z^2}$, R_i is roughly the radius of the blob, and B_i, called the blobbiness, controls the amount of interaction of with other nearby blobs. B_i is negative, and a smaller magnitude deforms the implicit surface more towards other blobs. The implicit function is given by the sum of the potentials. This model was designed primarily to visualize molecules.

Since implicit surfaces have no natural parametrization, applying 2D textures to them is an open problem. The latest result in this area by Pedersen [21] involves the interactive generation of geodesics on the surface. The user selects two points and a geodesic passing through these points is estimated by the system. Using these geodesics, the user delimits patches on the surface which are then parametrized by the system. This system is useful for adding surface detail with texture instead of geometry, but an automatic parametrization would be useful to texture the entire surface.

Another area of research is converting from a parametric to implicit surfaces. This conversion has theoretical interest and is desirable because certain geometric calculations like curve intersection is performed more readily in the implicit representation. Using Dixon's resultant [11], a tensor product surface of order $m \times n$ can be represented by the determinant of a $2mn \times 2mn$ matrix whose elements are linear polynomials in the Cartesian coordinates. Sederberg and Chen [23] have presented evidence that the size of this matrix can be reduced in many cases.

The problem of volume rendering is very similar to implicit surface tiling. In medical techniques such as magnetic resonance imaging (MRI) and single photon emission computed tomography (SPECT), values of some scalar field are measured at the points of a regular 3D grid. To visualize this data, isosurfaces of the field can be rendered. The seminal work in this area is the Lorensen and Cline "marching cubes" algorithm [19], although a very similar algorithm was independently published Wyvill *et al* [35]. Every cell of the 3D grid is polygonized individually. An 8–bit value is derived from the classification of the corners of the cell as lying inside or outside the isosurface. Symmetry reduces these 256 values to 14 cases, which determine the triangulation within that cell. The triangulation creates one vertex on every edge that has one end outside and one end inside the surface. The locations of these vertices are determined by linearly interpolating the values of the field on the corners. Between 0 and 4 triangles are created in each cell by this method. Coherence is exploited by analyzing the volume data in layers and the cells within a layer in rows. This method is very robust and sufficient for visualization applications, but it can produce very small triangles when the isosurfaces shears off a small corner of a cell. Also, the linear interpolation procedure to locate the vertices of the triangulation can be made more accurate when the surface to be tiled is represented implicitly rather than by the fixed–grid samples.

2.2 Other Implicit Surface Tilers

Before this work, there were three kinds of algorithms for tiling implicit surfaces. The most common approach is cell–based and is conceptually very similar to marching cubes. Another approach is a by–product of a particle–based technique for modelling with implicit surfaces. The third technique, called "shrinkwrapping" starts with a mesh covering a sphere that entirely encloses the model, and iteratively alters this mesh to fit the surface.

2.2.1 Cell–based Tilers

Cell–based implicit surface tilers resemble the marching cubes algorithm, but differ in that the function representing the surface can be evaluated at any point in space. This leads to several optimizations and enhancements which produce a better tiling.

Rather than dividing space into small cells on a fixed grid and testing the implicit function on the corners of each of these cells, two techniques were developed by Bloomenthal [4] to limit the number of evaluations of the implicit function. The first, called converged parititioning, encloses the implicit surface in an octtree. The octtree cells are subdivided if they contain some part of the surface (i.e. if some corners have positive function values and some negative), until the minimum cell size is reached. The disadvantage of this technique is that some parts of a surface may be missed by large cells.

A second space partitioning technique is called tracked partitioning [4]. A seed cell that is small with respect to the surface detail and that intersects the surface is located. Then the adjacent cells that also intersect the surface are iteratively considered. The cells are stored in an adjacency graph to ensure that the same cell is not considered twice. The disadvantage of this technique is that the user must specify a cell size that will produce an adequate tiling.

Adaptive subdivision can be applied to either converged or tracked parititioning. The cells to be polygonized are subdivided into smaller cells based on a number of geometric criteria such as the planarity of the resulting polygon and the divergence of vertex normals of the polygon. When using adaptive subdivision, more sophisticated polygonization techniques must be employed to avoid artifacts in the tiling at places where large cells are adjacent to cells that have been subdivided.

One of two types of cell geometries is usually used by cell–based tilers. Space is easily packed by cubic cells, and they are convenient for indexing schemes to keep track of the volume already considered. Some tiling algorithms, as that of Ning and Bloomenthal [20], use tetrahedral decomposition. Cubes are still used when tracking over or converging to the surface, but each cube is decomposed into five or six tetrahedra for polygonization. A 5–tetrahedron decomposition is a little more complex to use as adjacent cubes must have mirror–image decompositions to avoid artifacts in the tiling. Also, additional facets are produced the polygonization of the tetrahedra, without necessarily improving the fit of the tiling to the surface. However tetrahedral decompositions do not suffer from topological ambiguity.

Ambiguity is a problem when the cells are large with respect to the curvature of the surface. One example of topological ambiguity for cubic cells occurs when two vertices across a face diagonal are inside the surface and all other vertices of the cube are outside. In general any polyhedral cell geometry that does not have the property that any subset of the vertices is separable from the rest by a single plane gives rise to topological ambiguity. When adjacent ambiguous cubes are not polygonized consistently, holes in the tiling result. The easiest way to handle ambiguity is to include both possible polygonizations of an ambiguous cell, but this "hack" is not always appropriate. Another possibility is to polygonize the ambiguous cells in a way which chooses a consistent but arbitrary topology. Using tetrahedral cells avoids the whole problem of ambiguity.

Most cell–based polygonization techniques apply to manifold surfaces, but some work has been done by Bloomenthal and Ferguson to tile non–manifold surfaces [6]. Such surfaces are represented by functions $f : \Re^3 \rightarrow S$ where S is a finite set representing the different regions of space delimited by the function. The algorithm uses a tetrahedral decomposition. The primary contribution of this work is a set of methods that handle polygonizing the cells, given that there are more than two classifications of the corners. This polygonizer is much more complex than cell–based tilers for manifold surfaces.

2.2.2 Particle–based Methods

Much work has been done using particle–based methods with implicit surfaces. Particle systems were introduced in computer graphics by Reeves to create visual effects [22]. In the area of implicit surfaces, particle systems are usually used to control the implicit function when modelling the surface, but a polygonization can be obtained as a final step. These techniques involve some sort of constraints on the particles that force them to the surface (usually with constant density), and then a Delaunay triangulation algorithm is applied to these points to generate a mesh.

Szeliski and Tonnesen developed an approach for surface modelling which uses oriented particles [26]. Inter–particle potentials control the relative orientations and positions. This particle system is not meant to sample an implicit surface, but rather defines the surface. The position potential is such that only neighbouring particles need to be considered, and that particle density over the surface is constant. Particles are created at sample locations, and then more particles are added by "stretching" and "growing" rules, until a desired density is reached. The particles, and the surface which they represent were usually rendered as flat disks.

Figueiredo *et al* propose a similar method [10] designed specifically for implicit surface tiling. This method begins by randomly positioning points in space and settling them to the surface using the gradient of the implicit function. The points are then subjected to inter–particle forces similar to the proximity potential in Szeliski's and Tonnesen's work, but constrained in their motion to the surface by projecting the forces into the tangent plane at each point. The results are only briefly discussed, but it is mentioned that approximately 100 iterations are performed when relaxing the particles after they have been placed on the surface.

Witkin and Heckbert [34] also developed a method for shaping implicit surfaces. Their particles are either control points (which can be manipulated by the user and which affect the implicit function) or floaters (which serve only to represent the surface). Some particles are initially placed on the surface and the particle population is controlled by fissioning and killing rules. The forces on the particles are computed by inter–particle potentials, but rather than iterating over the particles to compute the forces and integrating the equations of motion individually, the full coupled linear systems are solved using matrix techniques. For 150 floaters, their system updates at 10Hz on a 100MHz RISC processor.

2.2.3 Shrinkwrapping

An interesting new method of tiling implicit surfaces was presented recently by van Overveld and Wyvill [32]. This method, called shrinkwrapping, could be loosely described as a hybrid of cell– and particle–based polygonization. First a triangulation of large sphere that encloses the surface is created. Then the vertices of this triangulation are allowed to slide down the integral curves of the gradient of the implicit function (which emanate from the surface at right angles to the tangent planes) until they arrive at the surface. The intermediate steps of this procedure represent other isosurfaces of the implicit function. At each iteration, the vertices are moved by one step of a Newton–Raphson method. This technique handles only differentiable surfaces generated by skeletons; the form of such surfaces is $V(r) = \sum_i \frac{\rho_i}{|r-R_i|}$, where ρ_i is a blending parameter, and R_i represents the distance to the closest point on the generating elements (points, lines, and polygons).

In order to represent the surface accurately, edges in the triangulation are sometimes split. A parameter β, which represents the minimum radius of curvature on the surface, must be supplied by the user. Splitting occurs whenever the edge lengths get too long according to a formula not directly related to the local curvature, but based on the normals to the isosurface at the vertices of a triangle and β.

A second paper on the shrinkwrap algorithm by Bottino *et al* [8] addresses surfaces of topological genus higher than zero. The algorithm is updated to detect critical points (where the gradient of the function vanishes) as the surface shrinks through them. The authors identify two types of critical points (see Figure 2.1). One represents a rupture where the number of components changes. The second represents a puncture, where the genus of one component of the surface changes. The triangulation of

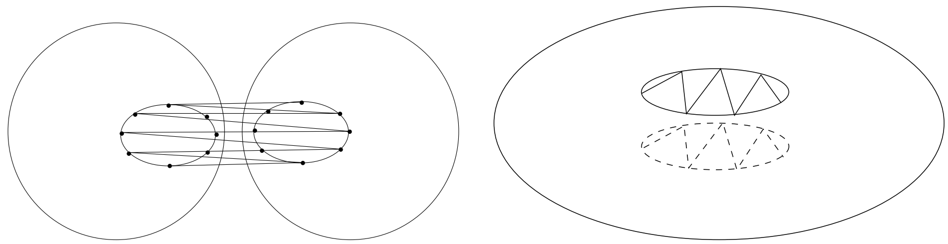

Figure 2.1: The two types topological changes (bifurcation and puncture) that a surface can undergo, and the triangulation around the critical point before the change.

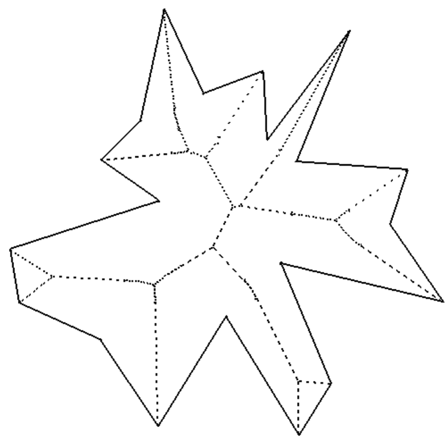

Figure 2.2: A polygon and its medial axis.

the shrinkwrap is corrected via *ad hoc* methods to account for these changes.

The updated shrinkwrap algorithm does not handle arbitrary offset surfaces. There is an implicit assumption in the algorithm that the integral curves of the gradient field which pass through a point on the desired level set all reach towards some point on an infinite sphere enclosing the level set. This is not the case when the gradient field contains a sink that is not located on the skeletal geometry, as in the surface consisting of eight blobs on the corners of a cube in Figure 6.1.

2.3 The Medial Axis

The medial axis (or skeleton) of a closed 2D curve is defined as the set of the centers of all maximal circles contained within the curve (see Figure 2.2). The set may be equivalently defined as the set of points within the curve such that there are at least two equidistant nearest points on the curve. For a volume, the medial axis is similarly defined as the set of centers of enclosed spheres.

As mentioned earlier, a medial axis transformation was used in a first attempt at filling the gap left in the triangulation by the growing phase.

Proposed by Blum in 1967 [7], the medial axis has received much attention in Computer Science literature. Since it reduces a shape to a compact representation, it is an important tool in pattern and character recognition [17]. For such applications, an approximation to the skeleton is often computed from an image using pixel–based methods [27].

The skeletons of planar polygons consist of segments of lines and parabolas. Several $O(n \log n)$ (in the number of vertices) algorithms have been developed to compute such analytical medial axes. Lee's algorithm [18], which is one of these, was implemented for this work, and will be discussed in detail later.

In 3D, Brandt computes an approximation of the medial axis by sampling a surface and extracting the skeleton from the Voronoi diagram of the samples [9]. Skeletons have been used in Computer Graphics by Ferley for surface reconstruction [12]. The skeleton of some surface is computed and the surface is reconstituted in an implicit representation.

2.4 Contribution

In this thesis, a new implicit surface tiling algorithm is presented. The new algorithm is applicable to smooth connected surfaces of any genus and generates a mesh of well–shaped triangles. The triangles are sized according to the local curvature, so that smaller triangles are created in areas of high curvature.

The new algorithm is slower to execute than the cell–based tilers because it must evaluate the curvature of the implicit surface. However, it can generate a mesh that represents a given implicit surface better than these tilers with fewer triangles. The new algorithm does not require the gradient of the implicit function to be defined in all space like the shrinkwrap algorithm does, nor does it have difficulty with critical points away from the surface.

Chapter 3

Recursive Triangle Growing

This chapter presents the first phase of the tiling algorithm. First, the problem is formally defined in terms of the inputs and outputs of the algorithm. The data structures and some self–contained subroutines necessary to implement the first phase are presented. The initialization of the tiling is then discussed, and the recursive triangle procedure that is at the heart of the first phase is detailed. Finally, the output of this phase is qualitatively described and the problem to be solved by the remainder of the algorithm is briefly discussed.

3.1 Definition of the Tiling Problem

The tiling algorithm was designed to be able to generate only G^1 surfaces. If the surface is defined by a function, rather than being defined procedurally, this restriction implies that the function must be differentiable. Intuitively, the surface cannot have any sharp edges or corners. This is a common class of implicit surfaces. Of the existing algorithms that tile implicit surfaces, only the cell–based polygonizers handle non–differentiable functions.

The other inputs to the algorithm are a function to compute the direction of the gradient of the implicit function at a given point, a bounding box that entirely contains the surface, and a curvature

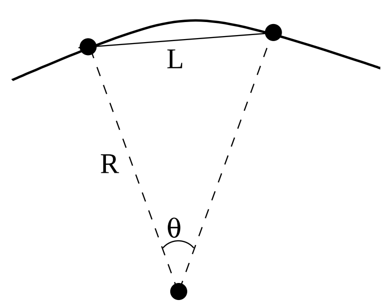

Figure 3.1: The curvature parameter ρ is defined by the desired value of $\frac{L}{R}$ where L is the length of an edge of a triangle on the surface, and R is the radius of curvature in that area of the surface. ρ is roughly equal to $\tan\theta$.

parameter. The curvature parameter, ρ, controls the fineness of the tiling. It is the desired optimal ratio of triangle edge length to local radius of curvature of the surface (Figurereffig:rho). In the same area on the surface, a higher value of ρ will produce larger triangles. A typical value for ρ is $\frac{1}{7}$.

The output of the algorithm is a mesh of triangles. The triangles are near equilateral, and are sized according to the curvature of the part of the surface that they cover. The resulting mesh has no T–vertices or holes like the meshes produced by some cell–based polygonizers.

3.2 Data Structures and Subroutines

In order to make storage and geometric operations on the mesh efficient, the tiling algorithm requires some specialized data structures and subroutines.

3.2.1 The Simplified Winged–Edge

The triangles computed by the tiling algorithm are stored in a simplified winged–edge data structure [2]. Triangles consist of three edges, each of which consists of two vertices. All edges store pointers to the one or two triangles to which they are adjacent. Edges are ordered counter–clockwise around each triangle, as seen from above the surface. A hash table that maps vertex values to edges is maintained in order to make it possible to traverse the mesh starting from vertices obtained from other data structures.

3.2.2 The Octtree of Vertices

All the vertices of the triangles are kept in an octtree. An octtree is a recursive, Cartesian axis–aligned partition of 3D space. The top node is a cube which is initialized in this application with the bounding box of implicit surface. Each node in the tree is either a leaf node which contains vertices (both geometrically and logically), or a branch node which only stores pointers to sub-nodes. As the tiling algorithm computes triangles, their vertices are entered into the octtree.

Adding a vertex is a recursive procedure. The tree is descended along the path of nodes that contain the vertex until a leaf node is reached, into which the vertex is added. If the leaf node contains too many vertices, it is subdivided into eight sub–nodes, and the vertices in the subdivided node are distributed into the sub–nodes according to their coordinate values.

The octtree affords an efficient way of determining the subset of vertices within a given distance of a given vertex. This query is implemented by descending all those branches which geometrically contain any part of the sphere surrounding the given vertex, and then testing the vertices in the leaf nodes. This query is crucial to the tiling algorithm.

3.2.3 Triangle Proximity

The tiling algorithm requires a measure of the proximity of two triangles on the surface. The canonical definition of proximity might be zero if the triangles overlap, and otherwise the minimum distance along any geodesic of the surface between any two points on the perimeters of the triangles. Proximity is clearly not easy to compute by this definition, so a simpler one was devised.

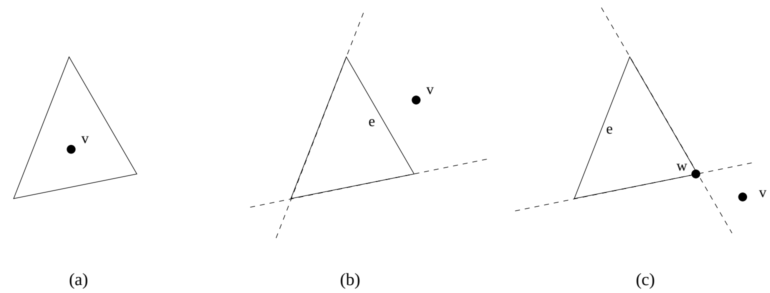

Figure 3.2: The three cases of the definition of $\mathrm{d}(v,T)$. In (a) v is contained in T, in (b) v is contained in only two positive half–spaces, and in (c), v is contained in only the half–space e^+.

Let e be an edge of a triangle T. Denote by e^+ the half–space which contains the vertex of T opposite to e and which is defined by the plane containing e and the normal to T. Further denote by $V(T)$ the set of vertices of the triangle T. The distance between two triangles T_1 and T_2 on the surface, denoted $\mathrm{d}(T_1,T_2)$, is then defined as zero if the triangles overlap and otherwise as:

$$d(T_1,T_2) = \min\left(\min_{v\in V(T_1)} d(v,T_2),\ \min_{v\in V(T_2)} d(v,T_1)\right)$$

where $d(v,T)$ is the distance of a point v from a triangle T, which will be defined shortly. Overlap is detected by testing the vertices of T_1 with the half–spaces generated by T_2 and vice versa. The two triangles do not overlap if and only if one of them has an edge e such that none of the vertices of the other triangle are contained in e^+.

Let $\mathrm{d}(u,w)$ denote the Euclidean distance between points u and w and let $\mathrm{d}(u,s)$ denote the Euclidean distance between a point u and the point closest to u on a line segment s. The distance of a point v to a triangle T, $\mathrm{d}(v,T)$, is defined as follows (see Figure 3.2):

- if v is contained in all of the half–spaces e_1^+, e_2^+, and e_3^+ of T, then $\mathrm{d}(v,T) = 0$,
- if v is contained in two of the half–spaces but not in that generated by e, then $\mathrm{d}(v,T) = \mathrm{d}(v,e)$,
- if v is contained in only one half–space, say that generated by edge e, then $\mathrm{d}(v,T) = \mathrm{d}(v,w)$ where w is the vertex of T opposite the edge e.

Since only the proximities of the triangles "close" to each other on the surface are important, further define $\mathrm{d}(T_1,T_2)$ as infinite if the dot product of the normals of T_1 and T_2 is less than zero. Note that for any two triangles T_1 and T_2, $\mathrm{d}(T_1,T_2) = \mathrm{d}(T_2,T_1)$ by this definition.

3.3 The Seed Triangle

The first step of the growing phase is the acquisition of a seed triangle. A point somewhere on the surface is required to calculate the seed triangle. This seed point is found by sampling the implicit function f in the bounding box until one point inside and one point outside the surface are located. The seed point is located by bisecting the interval between these two points until a low tolerance is reached.

Once the seed point is placed, a small equilateral triangle is created in the tangent plane to this point on the surface (which is determined by computing the normal to the surface at the seed point using the gradient function). The lengths of the edges are set to some small fraction of the length of the bounding box of the surface. Next, the vertices of the triangle are settled to the surface.

Settling a point v which lies near the surface down onto the surface is a procedure used in various places in the algorithm. The settling procedure also requires a parameter d that is an estimate of the distance from the starting point to the surface. First, an image point v' on the other side of the implicit function from v is computed, and then the distance between v and v' is bisected, maintaining two points that bracket the surface, until a certain tolerance is reached.

To find the image point, the vector

$$\hat{n} = -\frac{f(v)\nabla f(v)}{\|f(v)\nabla f(v)\|}$$

which points towards the surface from v, is computed. Then f is sampled at increasing distances, starting with d, from v along $\hat{n}$ until a point v' is found such that $f(v')$ has the opposite sign of $f(v)$.

The edges of the seed triangle are now iteratively set to the optimal size; the curvature[1] is evaluated using the vertices of the triangle, and the triangle is recomputed using the settling procedure until a tolerance is reached.

3.4 Triangle Growing

The growing procedure is initialized by placing the edges of the seed triangle into the active edge list. **Active edges** are those edges of the triangulation which are adjacent to only one triangle. At the completion of the entire algorithm, all edges will be adjacent to exactly two triangles.

Growing proceeds by iterating over the active edge list until a quiescent state is reached. Each pass applies one of two operations, which will shortly be described in detail, to the elements of the list. Odd passes through the list apply the "isosceles triangle growing" operation, and even passes apply "ear cutting". Both operations have preconditions and an operation is executed only if its conditions are met. If an operation is executed, some edges are removed from the active edge list and some new ones are added. Iteration stops when two consecutive passes produce no changes.

The isosceles triangle growing and ear cutting operations have the property that if the preconditions ever fail, they will not be met at some future time in the algorithm. Therefore, iteration over the active edge list is optimized such that each operation is attempted on each active edge only once.

3.4.1 Isosceles Triangle Growing

This operation produces a new triangle that is adjacent only to the active edge. Given a candidate active edge, a near equilateral triangle is computed by placing a point in the tangent plane of the active edge (to the side of the edge outside the current tiling) and settling this point to the surface (see Figure 3.3). The three vertices are used to estimate R, the radius of curvature, and then the point is slid closer or further away from the active edge such that the lengths of the new edges are ρR.

[1] See the Appendix for a description of curvature evaluation.

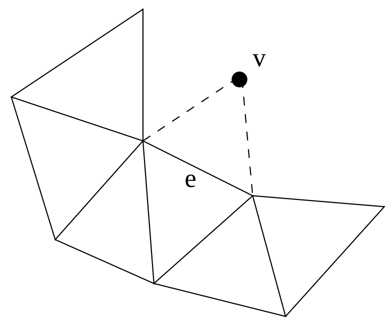

Figure 3.3: A new triangle being added off of an active edge e. A new vertex v is created to add the triangle.

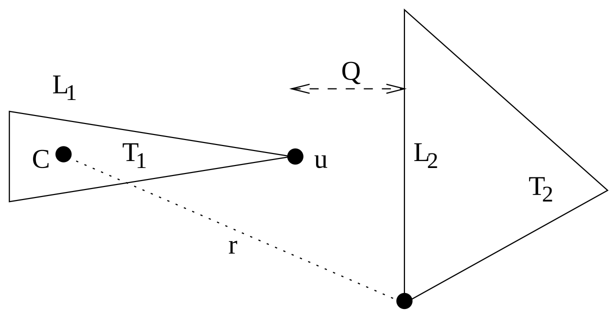

Figure 3.4: The geometry which produces the highest ratio of proximity radius r to inter–triangle proximity Q. T_1 is a narrow triangle with axis perpendicular to the closest and longest edge of T_2.

The new triangle is accepted only if certain tests are satisfied. These tests are intended to keep new triangles from overlapping or approaching existing triangles too closely. The first test discards the new triangle if the angles that the new active edges would make with their neighbours are too small. A more involved test discards the new triangle if it is too close to any other triangle.

Let T' be the new triangle and $\mathcal{T}$ be the set of existing triangles. The new triangle is accepted if it satisfies the proximity condition:

$$\forall T \in \mathcal{T}\{\mathrm{d}(T',T) > \lambda \max(L(T'), L(T))\}$$

where the function $L(T)$ denotes the longest edge of triangle T, and λ is a constant (usually set to $\frac{1}{2}$). Rather than testing the new triangle against all the existing triangles, the evaluation of the proximity condition is optimized by using the octtree to find nearby triangles and then computing the distance between the new triangle and the nearby existing ones. The center of the sphere with which the octtree is queried is located on the centroid of T'. A minimum radius r is chosen such that the sphere will contain at least one vertex of every triangle that is too close.

Figure 3.4 shows the situation which generates the maximum possible value of r. In the diagram, $L_1 = L(T_1)$ and $L_2 = L(T_2)$, and Q is the inter–triangle distance $\mathrm{d}(T_1, T_2)$. The distance from C, the centroid of T_1, to the vertex u is at most $\frac{2}{3}L_1$, so

$$r^2 < \left(\frac{2}{3}L_1 + Q\right)^2 + \left(\frac{1}{2}L_2\right)^2$$

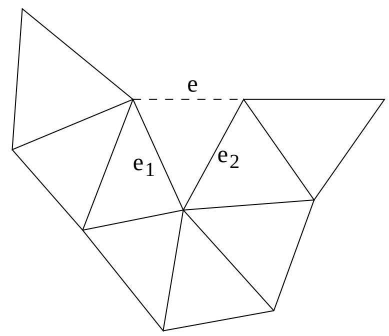

Figure 3.5: A new triangle is created by the ear cutting operation. A new edge e joins active edges e_1 and e_2 to form the new triangle.

Since the longest edge of all the existing nearby triangles is unknown, the longest edge of any triangle ever added to the mesh is maintained by the algorithm, and used to set the value of r.

Using the longest edge of all triangles makes this test very conservative. Especially in areas of high curvature, nearby triangles are small compared to the largest triangle in the mesh, and many triangles are needlessly checked.

Once the nearby vertices have been selected from the octtree, the triangles which contain those vertices and an active edge are determined from the winged–edge data structure. Any triangles adjacent to either of the vertices of the current active edge are removed from this list. These triangles have zero distance from the new triangle but should not prevent its addition to the mesh. The proximities of the remaining triangles to the new triangle are computed, and the new triangle is added to the mesh if all the remaining triangles are sufficiently distant according to the proximity condition.

Consider the result of the isosceles triangle growing operation applied to a number of colinear active edges with no other triangles nearby. A new triangle would be successfully placed at each of the active edges in a saw–tooth pattern. The spaces between these "teeth" could then be filled in with more triangles. This observation motivates the second mesh–building operation.

3.4.2 Ear Cutting

The second operation applied to active edges, known as ear cutting, is borrowed from planar polygon triangulation algorithms [13, 28]. Ear cutting is applied when two adjacent active edges have a smaller external angle than some threshold (usually set at 70 degrees). The three vertices on the two edges are filled in with a new triangle (see Figure 3.5).

This time, all triangles adjacent to the three vertices of the new triangle are discarded from the list of nearby triangles. Again the proximities of the remaining triangles to the new triangle are computed, and the new triangle is added only if the are sufficiently distant.

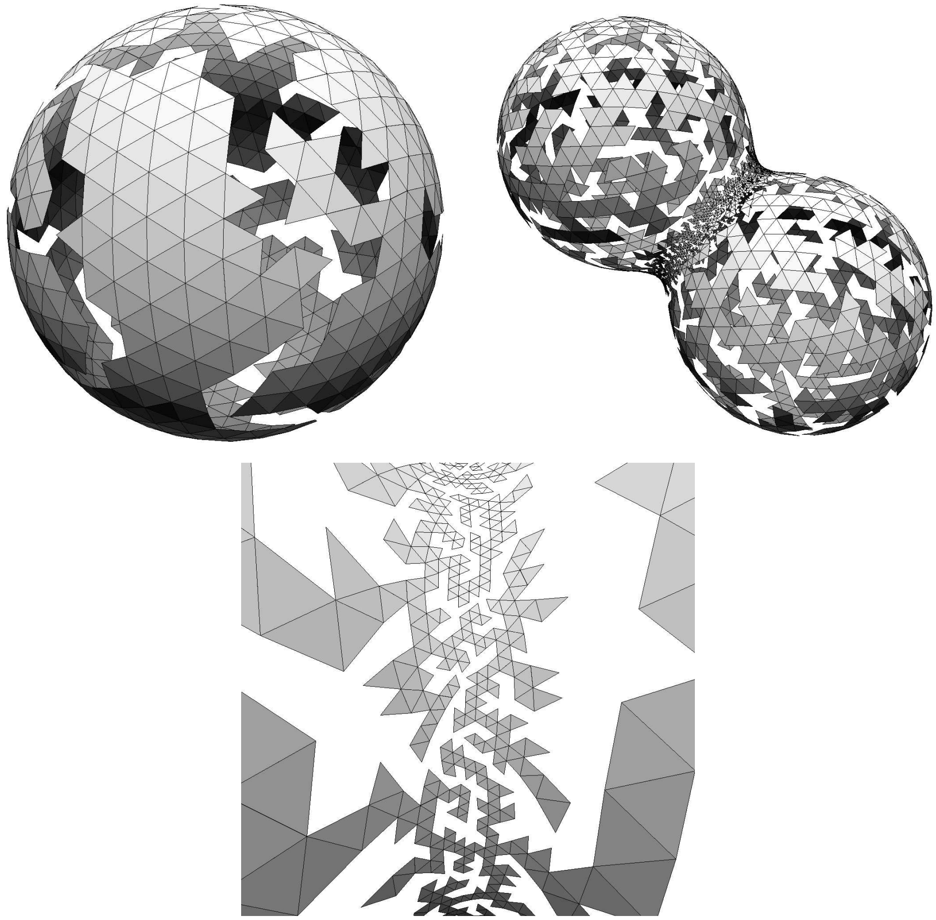

Figure 3.6: A sphere model, a peanut model, and a close–up of the peanut model. The pictures show the mesh after the triangle growing phase of the tiling algorithm.

3.5 The Gap

Ear cutting and isosceles triangle growing are geometrically complementary operations. Growing extends the perimeter of the tiling into new parts of the surface, and ear cutting fills in holes left by growing, allowing more triangles to be grown off of the now smooth parts of perimeter. A basic assumption of this algorithm is that when the quiescent state is reached, every region of the surface will be partially covered by triangles (see Figure 3.6).

Since every triangle added to the mesh after the seed triangle is edge–adjacent to an existing triangle, the tiling after the first phase is a single connected component. The uncovered part of the surface is known as the **gap**. The next two chapters deal with the problem of filling the gap.

Chapter 4

The Medial Axis of the Gap

The growing phase leaves a gap in the tiling. Due to the topological properties of the growing phase, the gap always consists of a single cycle of edges. In other words, the graph consisting of the active edges is such that every vertex has order 2.

At first glance, an *ad hoc* method for covering the gap with triangles is promising; if the tips of the fingers were located (perhaps by checking the angles between a few sequential edges), the fingers could be traversed bottom–up. Some experimentation with this idea showed that seaming the triangulations of the fingers at the bifurcations is hard to accomplish with *ad hoc* rules.

Since the gap is a (non–planar) polygon, a medial axis seemed helpful. The 2D medial axis concisely describes the structure of a polygon, and it seemed likely that the information it contains would be useful in placing vertices and edges for the triangles to fill the gap. Because the edges of the gap polygon are sized with respect to the local curvature, it was hoped that this polygon could be considered "locally planar", and that an approximation of a 2D medial axis could be computed. New vertices in the triangulation could be placed at the vertices of the skeleton, and the cells could be triangulated individually.

4.1 The Medial Axis of Planar Polygons

The medial axis (or skeleton) of a simple planar polygon P is the set of points Q such that each $q \in Q$ is internal to P and has at least two nearest points $p_1 \neq p_2$ on the perimeter of P such that $|q-p_1| = |q-p_2|$.

The points of Q line on straight lines and parabolas. These curves are defined by two edges of P,

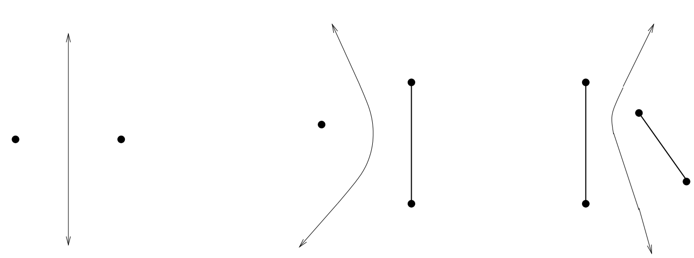

Figure 4.1: A point–point, point–segment, and segment–segment bisector.

two vertices of P, or an edge and a vertex of P (see Figure 4.1). For example, the two adjacent edges of every convex vertex of P will generate a part of the medial axis which lies on the angular bisector of the edges.

The medial axis algorithm developed by D. T. Lee [18] was selected for implementation. This algorithm would be exact if computers had arithmetic with infinite precision, and it was claimed to be simpler to implement than its predecessors. It runs in $\mathrm{O}(n \log n)$ time in the number of vertices of the input polygon.

4.1.1 Definitions

The curves in the medial axis are generated by **elements**, which are either points or segments. **Points** are reflex vertices (those with internal angle greater than π) of the input polygon. The remaining vertices are called **convex** and are not considered elements. The edges of the polygon are **segments**. A **chain** is a maximal connected sequence of elements around the perimeter of the polygon that does not contain a pair of segments adjacent at a convex vertex. The perimeter of the polygon is divided into disjoint chains, bounded at convex vertices.

The locus of points equidistant from a pair of elements (e_1, e_2) is called the **bisector** of e_1 and e_2, and is denoted $\mathrm{B}(e_1, e_2)$. When e_1 and e_2 are both points, the bisector is an infinite straight line. When one of the elements is a point, and the other is a segment, the bisector generally consists of three curves: two rays and a part of a parabola. When a bisector contains several curves, they meet such that a tangent vector is defined along the entire curve. The bisector is always a curve of infinite extent in both directions.

Every element e has an associated **cell**, $\mathrm{C}(e)$, which is the set of points on the interior of the polygon that is closer to e than to any other element. The cell $\mathrm{C}(e)$ is bounded by e (when e is a segment) and by bisectors (which may be thought of as the edges of the cell), each of which are generated by e and some other element. The union of the cells of all elements plus the cell boundaries covers the interior of the polygon. The set of cells corresponding to all elements is called the Voronoi diagram of the elements.

4.1.2 Computing Bisectors

In the following discussion, lines perpendicular to segments and passing through their endpoints will be used to divide the plane into regions. Suppose v is an endpoint of a segment s. Let n_v be the perpendicular through v and let N_v^+ denote the half–space delimited by n_v that contains the other endpoint of s. Let N_v^- be the half–space complementary to N_v^+. Also, if s has endpoints v_1 and v_2, denote $N_s = N_{v_1}^+ \cap N_{v_2}^+$.

The bisector of two points p_1 and p_2 is the line containing the midpoint of p_1 and p_2 and having a tangent orthogonal to $p_1 - p_2$.

When a point p and a segment s are colinear, the bisector of these elements is simply the bisector of p and the endpoint of s closest to p. When p and $s = (v_1, v_2)$ are not colinear, the plane may be divided into three regions by n_{v_1} and n_{v_2}. In the center region N_{v_1,v_2}, $\mathrm{B}(p, s)$ consists of the parabola having p as the focus and a directrix coincident with s. In the other two regions, $N_{v_1}^-$ and $N_{v_2}^-$, the bisector is $\mathrm{B}(p, v_1)$ and $\mathrm{B}(p, v_2)$, respectively. A special case occurs when p is coincident with one of the endpoints

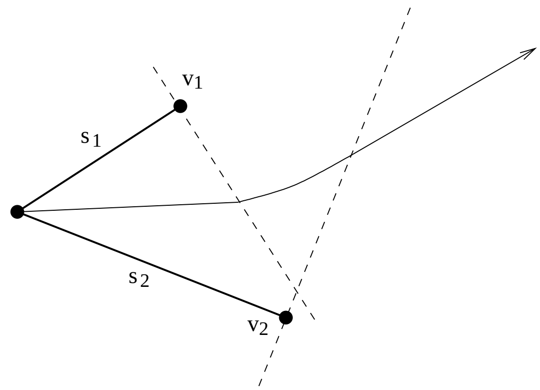

Figure 4.2: A segment–segment bisector for two segments with a common end–point.

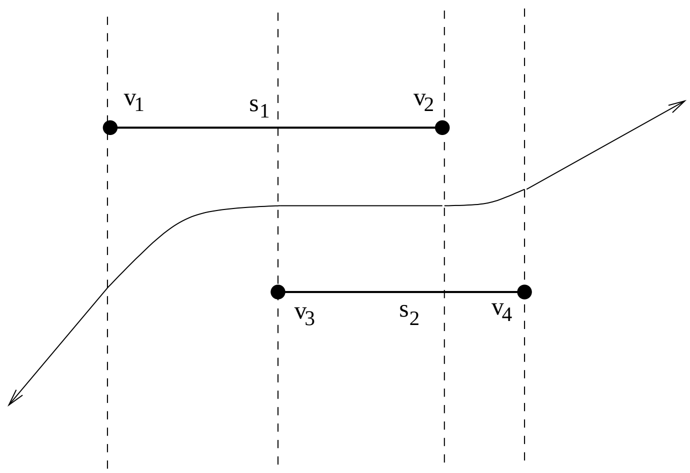

Figure 4.3: The bisector of two parallel segments.

of s. In that case, the $\mathrm{B}(p, s)$ is defined as the perpendicular to s passing through p (this is useful for the medial axis algorithm described in the next section).

Bisectors of the form $\mathrm{B}(s_1, s_2)$ may be divided into several sub–cases. The first case occurs when s_1 and s_2 have a common endpoint as in Figure 4.2. If s_1 and s_2 are the same length, $\mathrm{B}(s_1, s_2)$ is just the angular bisector of s_1 and s_2. Now assume without loss of generality that s_1 is shorter than s_2. Let v_1 be the lone vertex of s_1 and v_2 be the lone vertex of s_2. In $N^+_{v_1}$, $\mathrm{B}(s_1, s_2)$ is the angular bisector of s_1 and s_2. In $N^-_{v_1} \cap N^+_{v_2}$, the bisector is the parabola generated by v_1 and s_2, and in N^-_2 it is $\mathrm{B}(v_1, v_2)$.

The second case occurs when s_1 parallel to s_2. If the segments are also colinear, $\mathrm{B}(s_1, s_2)$ is the point–point bisector of the pair of endpoints, one on s_1 and one on s_2, that is closest. In general, suppose that s_1 and s_2 are viewed in a coordinate system such that they are both parallel to the x–axis. Label the endpoints of s_1 from left to right by v_1 and v_2 and the endpoints of s_2 similarly as v_3 and v_4 (Figure 4.3). If the region $N_{s_1} \cap N_{s_2}$ is non–empty, the bisector in this region will be a straight line parallel to s_1 and passing through the midpoint of a perpendicular connecting s_1 and s_2. In the region $N_{s_1} - N_{s_2}$, the bisector is a parabola with directrix coincident with s_1 and having the nearer of v_3 and v_4 as focus (and similarly for the region $N_{s_2} - N_{s_1}$). This region may be disjoint, in which case the bisector

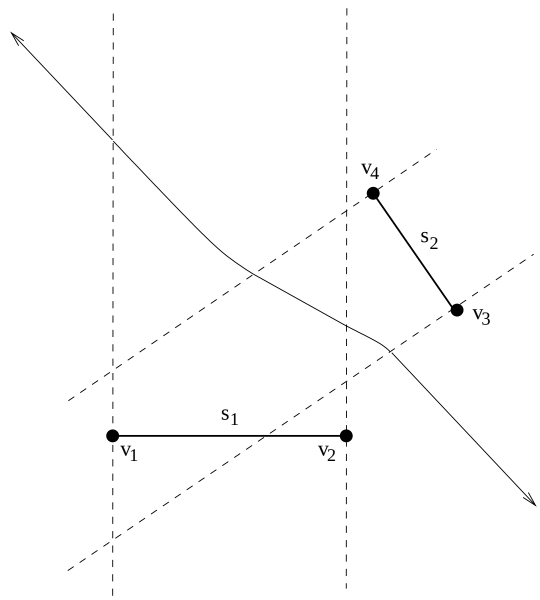

Figure 4.4: The bisector of two arbitrarily oriented segments.

is a different parabola in each component. Also, in any non–empty region $N^-_{q_1} \cap N^-_{q_2}$ where $q_1 \in \{v_1, v_2\}$ and $q_2 \in \{v_3, v_4\}$, the bisector is given by $\mathrm{B}(q_1, q_2)$. In practice, the bisector for such a configuration can be computed by sorting the endpoints of s_1 and s_2 from left to right, starting with the left–most region, and moving through the different regions to the right. Such a bisector can have up to five component curves. Special cases occur when any of the perpendiculars to s_1 and s_2 are coincident.

The most difficult case is the general case where s_1 and s_2 do not share an endpoint and when their extensions meet at a point. In this case, the plane can be divided into nine regions by the perpendiculars to s_1 and s_2, which is illustrated in Figure 4.4. The center region, defined by $N_{s_1} \cap N_{s_2}$, is a parallelogram, and is the only finite region. It also has the property that the bisector could pass through this region twice, whereas it will pass through every other region at most once. In the center region, the bisector lies on the angular bisectors of the extensions of s_1 and s_2. In the remaining regions, the bisector is either a point–point bisector or a parabola. Conceptually, the bisector can be computed by finding a starting region which contains an infinite length component curve, and tracing the curve through the different regions until the other infinite component is reached. A starting region is always defined by two endpoints, q_1 and q_2, on different segments such that the line through them partitions the plane such that both the remaining endpoints lie in the same half–space. The region defined by the perpendiculars associated with q_1 and q_2 will contain an infinite component curve of the bisector. Bisectors of this form can have at most seven component curves.

4.1.3 Overview of the Medial Axis Algorithm

The algorithm starts by computing the elements of the input polygon and grouping them into chains. This can be done by considering the edges and vertices of the polygon in counter–clockwise order, starting with any convex vertex. The edge after this vertex will be the start of a chain. If the next vertex is also

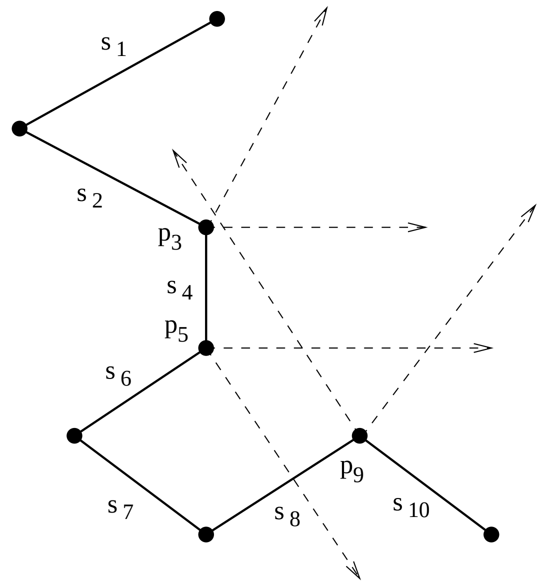

Figure 4.5: Some of the edges of a polygon and the corresponding initial cells. Recall that convex vertices are not elements.

convex, the first chain is completed. If instead the vertex is reflex, it is added to the chain as a point, and the next edge is added as a segment. The chain is continued until a convex vertex is reached.

Once the chains have been determined, the initial cells are computed. The initial cell corresponding to an element with two neighbours on the chain simply has two straight–line bisectors defined by the element and its neighbours. Elements on the ends of a chain only have one bisector in their cells. One end of the bisectors is truncated to lie at the perimeter of the polygon, so that the remaining piece lies on the "inside" of the polygon. Throughout the algorithm, the cells are maintained as a list of bisector pieces, in counter–clockwise order around the perimeter of the cell. The bisector pieces are called **cell edges**.

The chains and cells associated with the elements now form the initial **components** of the medial axis (see Figure 4.5). The algorithm proceeds by merging adjacent components pairwise. The order of merging is irrelevant except for optimization considerations. During merging, the cells in the components are updated as described in detail later. When the last two components have been merged, the result is the Voronoi diagram of the elements. The medial axis is obtained from the Voronoi diagram by deleting those bisectors that have a point as one of their generating elements.

Before being merged, cells from two components may overlap. The **component bisector** is a continuous curve that separates the plane into two regions such that points on each region are closest to the component in that region. The component bisector consists of pieces of element bisectors.

To merge the components, the component bisector is used to clip the affected cells in both components. The affected cells are those which contain a piece of the component bisector.

4.1.4 Computing Component Bisectors

During the course of the algorithm, component bisectors are only ever computed for two components that are adjacent (i.e. that share a vertex). The computation of the component bisector starts at this common vertex. For the purpose of this computation, the component bisector should be thought of as an oriented curve, starting at the point shared by the adjacent elements and extending into the polygon.

Let C_1 and C_2 denote the elements of the left and right components, respectively, and let $e_1 \in C_1$ and $e_2 \in C_2$ be the pair of those elements that are adjacent at a vertex v_0 (note that the final two components to be merged will have *two* such pairs). See Figure 4.6 where e_1 is element 6 and e_2 is element 7.

The component bisector will be constructed as a sequence $\{b_0, b_1, \ldots, b_k\}$ of intersecting bisector pieces. Let $v_i = b_{i-1} \cap b_i$ for $0 < i \leq k$. Initialize the current bisector b_0 to be $B(e_1, e_2)$. Initialize the current left cell as $C_\ell = C(e_1)$ and the current right cell as $C_r = C(e_2)$.

There now follows an iterative procedure to calculate the next bisector piece b_{i+1} given b_i, v_i, C_ℓ, and C_r; b_i is intersected with the edges of C_ℓ, tested in order clockwise from v_i, until an intersection is found. b_i is also intersected with the edges of C_r, tested in order counter–clockwise from v_i, until an intersection is found. v_{i+1} is assigned the closer of the two intersections (measured by arc–length along b_i, starting from v_i).

Suppose v_{i+1} lies on an edge of the current left component cell C_ℓ. To the other side of this edge is another cell C_{adj}, also of the left component. The bisector piece b_i is terminated at v_{i+1} and the current left cell is changed to C_{adj}. If v_{i+1} lies on an edge of the current *right* component cell, a similar procedure is applied.

The above procedure is applied iteratively until no intersection is found in either cell, at which point the last element of the sequence of bisector pieces b_k has been computed. b_k is semi–infinite unless C_1 and C_2 are the final two components to be merged, in which case b_k is truncated at the vertex other than v_0 that is in common to C_1 and C_2.

4.1.5 Clipping Components

Once the component bisector has been computed, the cells that were intersected have to be updated, and then the merge is complete. This clipping operation can be done cell by cell. A cell is said to be **open** if it contains cell edges that extend to infinity. All initial cells are open, but in the final Voronoi diagram all cells are closed.

The clipping operation must handle open and closed cells properly. In general, the cell edges that were intersected by the component bisector must be truncated at the intersection point, and the bisector pieces that were generated by the element associated with the cell become new cell edges.

4.2 Medial Axis Implementation

The medial axis algorithm, along with a driver program, was implemented in roughly 11,000 lines of C++ code. The most complex parts of the algorithm and therefore the most difficult to implement are the element bisector computation, the component bisector computation, and cell clipping. Another

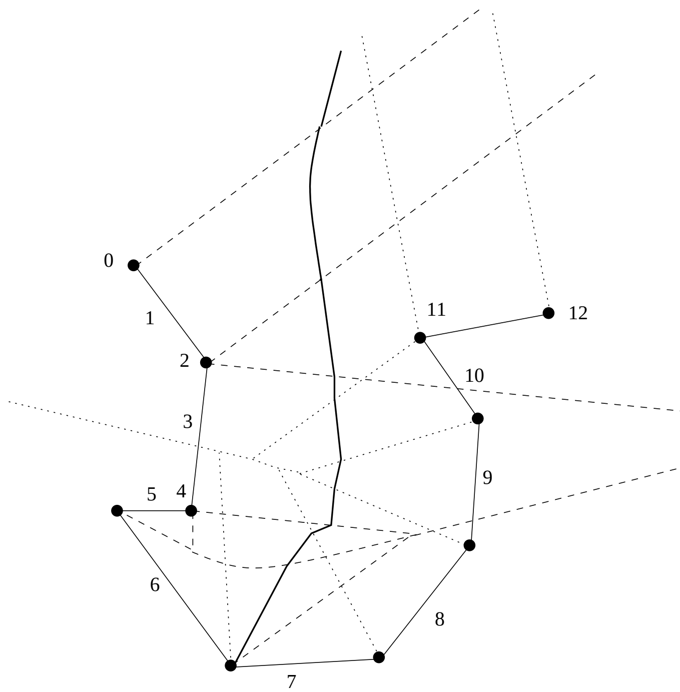

Figure 4.6: A component bisector cutting through the cells of two components. The elements $\{0, 1, 2, 3, 4, 5, 6\}$ belong to one component, and $\{7, 8, 9, 10, 11, 12\}$ belong to the other.

interesting part of the implementation is the computation of intersections of the element bisectors. However, the most time was spent fighting issues of numerical stability.

4.2.1 Computing Intersections

To ease the computation of intersections, the bisector curves were represented parametrically in the implementation. Lines and parabolas were represented as:

$$\begin{aligned} \vec{L}(t) &= \vec{n}t + \vec{p} \\ \vec{P}(t) &= (t\cos\theta - ct^2\sin\theta + x_0, t\sin\theta + ct^2\cos\theta + y_0) \end{aligned}$$

were $\vec{n}$ is the tangent vector to the line and $\vec{p}$ is a point on the line. The parameter c is twice the focal length and θ and (x_0, y_0) specify a rotation and translation from the natural coordinate system in which the bottom of the parabola lies on the origin and has an axis coincident with the y–axis.

Line–line intersections are straightforward, and line–parabola intersections are not much more difficult, requiring the solution of an equation quadratic in the parameter. Parabola–parabola intersections are a little more difficult; substitution yields a fourth order polynomial in the parameter, which was solved numerically using a Newton–Raphson derivative algorithm.

4.2.2 Numerical Stability

The most time consuming aspect of implementing this algorithm involved numerical stability. The algorithm makes many decisions based on geometric tests such as angle comparisons, dot product comparisons, and length comparisons. Once a decision is made, subsequent computations frequently depend on that decision, and must be made consistent with it.

An example of this dependency is found in the computation of the element bisector. If both elements are segments, they are tested for parallelism. To perform this test, the dot product of their direction vectors is compared to one. If the absolute value of one minus the dotproduct is within some non–zero ϵ, the segments are considered parallel.

Suppose two segments are not quite parallel by this definition. Then all the later calculations, such as the intersections of the perpendiculars to the segments (which define the nine regions of the plane in which the bisector is defined by different generators), have to be guaranteed to work. To some extent, this can be done by having two versions of functions such as the one to compute the intersection of two lines; one that uses ϵ and returns an intersection only if the lines are not considered parallel, and one that ignores this case and always returns an intersection.

The component bisector exemplifies a more problematic case of numerical instability in this algorithm. The problem is related to the fact that the first intersection with the left and right cells is chosen to define the next piece of the component bisector. These intersections can occur arbitrarily close together on the component bisector, making it difficult to order them reliably. Due to the fact that some of the intersections are computed numerically, it is possible that the closest intersection, which should have been used, will be discarded and the component bisector will go off in the wrong direction. About 0.2% of randomly generated test cases of polygons with 100 vertices exhibited this problem. Conceptually, some sophisticated code could have been written to detect these cases, but this was not attempted.

4.3 Application of the Medial Axis

A 2D medial axis is not directly useful to the problem of filling the gap in the tiling left by the growing phase because the gap polygon is not planar. Two adaptations of this concept to 3D were attempted. The first is an extension of the 2D algorithm to work on the gap polygon. This was thought possible because the gap polygon is locally planar. When this generalization failed, the feasibility of computing a complete 3D skeleton was investigated.

4.3.1 Generalization to Locally Planar Polygons

In order to accomodate non–planar input polygons, some aspects of the medial axis algorithm have to be modified. The non–trivial changes were made to bisector computation and bisector intersection.

The existing bisector computation code was reused. Point–point bisectors and point–segment bisectors were computed by transforming the elements onto the $x-y$ plane. For segment–segment bisectors, the best approximating plane for the four points was determined, the segments were projected into this plane, and the 2D bisector was computed for the projected segments. In 3D, bisectors were represented as a 2D bisector and a 4×4 transformation matrix that transforms the 2D coordinates, extended to 3D by adding a null third coordinate, into the 3D world coordinates.

To compute the “intersection” of two such bisectors, the orthogonal extrusion of the first was intersected with the second. This seems like a reasonable technique as the planes containing bisectors are locally close in proximity and close to parallel when the generating elements are close together. This surface–curve intersection produces a coupled system of equations in three variables: the parametric variables for the two curves and the extrusion parameter. For example, consider the case of finding the intersections of two parabolas $P_1(t)$ and $P_2(t)$. To simplify the computation, the parabolas are represented in the natural coordinate systems, so:

$$\vec{P_1}(t) = M_1 M_1^n \begin{pmatrix} t \\ c_1 t^2 \\ 0 \end{pmatrix}$$

where M_1^n transforms the natural coordinates of the parabola into the coordinates of the plane into which the two elements were projected, and M_1 transforms the coordinates this plane into world coordinates. Then the intersection of an extruded parabola with another parabola is given by:

$$\begin{aligned} \vec{P_1}(t_1, s) &= \vec{P_2}(t_2) \\ M_1 M_1^n \begin{pmatrix} t_1 \\ c_1 t_1^2 \\ s \end{pmatrix} &= M_2 M_2^n \begin{pmatrix} t_2 \\ c_2 t_2^2 \\ 0 \end{pmatrix} \\ \begin{pmatrix} t_1 \\ c_1 t_1^2 \\ s \end{pmatrix} &= (M_1 M_1^n)^{-1} M_2 M_2^n \begin{pmatrix} t_2 \\ c_2 t_2^2 \\ 0 \end{pmatrix} \end{aligned}$$

This system can easily be decoupled by substitution and solved as a fourth order polynomial similar to the 2D case. The solutions to this system represent points on the second parabola which are in a sense closest to the first parabola. The closest point on the first parabola to the solution point was taken as the intersection point in the coordinate system of the first parabola.

Unfortunately, this approach did not work as well as expected. The numerical problems discussed earlier were exacerbated by the approximations inherent in this generalization from 2D to 3D. Also, while the gap polygon may be locally planar in the sense that any three edges in sequence along the perimeter are close to coplanar, during the course of the medial axis algorithm elements quite far away from each other may have their bisector computed. Therefore, using the bisector for the segments projected into an approximating plane is not always a reasonable approximation. This bisector may not appear in any cell of the final Voronoi diagram, but since it is used in intermediate steps, it can cause the computation to fail.

4.3.2 The 3D Medial Axis

The possibility of computing a true 3D medial axis was investigated. This investigation quickly became mired in mathematical difficulties. In 3D, the cells corresponding to the elements of the polygon are volumes delimited by three types of surfaces: planes, orthogonal extrusions of parabolas as discussed above, and saddle surfaces.

The saddle surfaces are generated around non–coplanar segments. Such segments present the most difficult case of the bisector surface and will be discussed in detail. For each pair of such line segments, there exists a coordinate system in which these segments lie on the lines:

$$\begin{aligned}\vec{L_1}(t) &= t\hat{x} \\ \vec{L_2}(t) &= t(\cos\theta, \sin\theta, 0) + c\hat{z}\end{aligned}$$

In words, the first segment is chosen to lie on the x–axis, and the second segment lies in the plane $z = c$ where c can be chosen to be positive. Now choose an arbitrary point p and set the distance of the nearest point to p on L_1 equal to the distance of the nearest point to p on L_2. This defines the surface:

$$S(x, y) = \frac{1}{2c}[\sin^2\theta(x^2 - y^2) - 2xy\cos\theta\sin\theta + c^2]$$

If $s = (v_1, v_2)$ is now a segment in 3D, the notation from the 2D case can be directly extended to describe the regions of the bisector surface. n_{v_1} is now a plane that passes through v_1 and has a normal coincident with $v_2 - v_1$. $N^+_{v_1}$ denotes the half–space delimited by n_{v_1} and containing v_2.

Bisector surfaces generated by non–coplanar segments divide 3D space into nine regions, similar to segments in 2D. The planes perpendicular to the segments have coplanar normals, so they define a parallelepiped which is infinite in one of its axes. Unlike the bisector curve, the bisector surface is necessarily defined in all nine regions. In the center region, it is given by the saddle surface. In the regions separated from the center by a plane, it is defined by a point and a line, and is therefore an orthogonally extruded parabola. In the corner regions, it is defined by two points and is therefore a plane.

An algorithm that computes a 3D medial axis must calculate the intersection curves of bisector surfaces to define the volumes of the cells associated with elements. This is difficult when the bisector surfaces involve saddle surfaces or extruded parabolas. Even when using the natural coordinate systems, setting parametric representations of the surfaces equal produces a coupled system three equations of total second order in four variables. This system cannot generally be decoupled.

Research into the application of the medial axis to help in tiling the gap was abandoned at this point.

4.4 Conclusions

The medial axis is an interesting reduction that is potentially useful for geometric analysis as is required to tile the gap. A simple generalization of the planar medial axis to 3D is not feasible because the approximations and assumptions made in this approach are not accurate enough.

The canonical 3D medial axis is difficult to compute because the intersections curves of the bisector surfaces cannot be calculated analytically. The points on these curves could be computed numerically. However, even if such a procedure were not prohibitively expensive for the application, it is not clear how the results could be used to compute a medial axis.

Numerical skeletons of 3D shapes have recently developed [9] and applied to reconstruction of shapes [12]. Also, Hoff [15] *et al* developed an algorithm to compute 3D Voronoi diagrams numerically using graphics hardware. These algorithms are not directly applicable to this problem, because they do not supply the cells corresponding to the segments. Perhaps a similar approach could be developed to produce a result useful in solving this problem.

Chapter 5

Proximity

This chapter presents a technique entitled "proximity" that was successfully used to tile the gap and complete the polygonization algorithm. After the failure of the complex skeleton attempt, proximity is a refreshingly "low tech" approach. It is motivated by the observation that the line segment between a vertex and its nearest neighbour across the gap is often a good candidate for an edge in the triangulation of the gap.

At the core of this technique is the proximity data structure. Filling the gap is implemented with a number of operations which make use of the proximity data structure and some simple geometric information such as the angle between adjacent edges.

5.1 The Proximity Data Structure

While the medial axis is a complete description of a shape, the proximity data structure only associates the nearest neighbour (formally defined later) with every vertex. Computing this data structure is simple and efficient and has none of the numerical problems associated with the medial axis. The input to the routine that computes the proximity data structure is simply the list of vertices in counter–clockwise order around the gap.

5.1.1 Definitions

The nearest neighbour relationship is carefully defined so as to make the line segments between a vertex and its nearest neighbour a good segment in the triangulation.

For a vertex v on the gap, let N_v be the normal to the implicit surface at v, let v_{pred} be the vertex preceding v on the boundary of the gap, and let v_{succ} be the vertex succeeding v (recall that the vertices are ordered counter–clockwise).

Every vertex v on the gap has two **associated planes**, P_v^1 and P_v^2. P_v^1 embeds v, v_{pred}, and N_v, and is oriented such that the normal to the plane points into the gap. P_v^2 is defined similarly except that it embeds v_{succ} instead of v_{pred} (see Figure 5.1).

A vertex u is said to be **above** P_v^1 if it is contained in the half–space into which the normal of P_v^1 points, and is farther from P_v^1 than one tenth the distance between v and v_{pred}. The definition of "u is

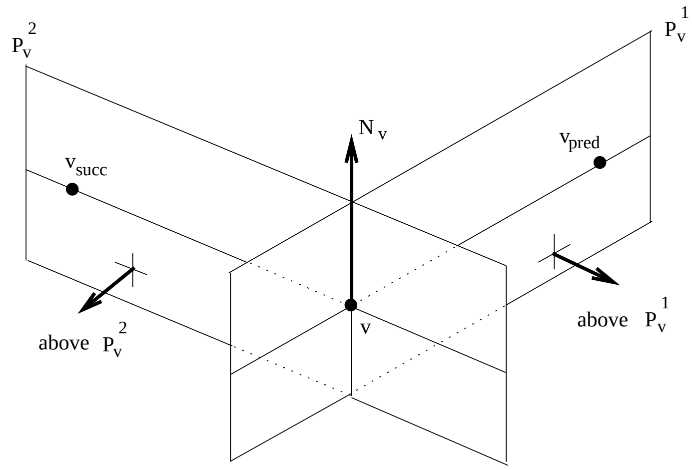

Figure 5.1: A concave vertex on the gap and its associated planes.

above P_v^2" is symmetric.

A vertex v on the gap is **convex** if the interior angle that it makes with v_{pred} and v_{succ} is less than π and **concave** otherwise.

Every vertex v has an associated set of neighbours which consists of other vertices on the gap. If v is convex its **neighbours** are those vertices which are above P_v^1 and P_v^2, and if it is concave its neighbours are those which are above one or more of P_v^1 and P_v^2. Note that the set of neighbours may be empty. This case can occur when there is a small internal angle between the two edges on the gap containing v. Note further that if v_1 is a neighbour of v_2, the converse is not necessarily true.

The **closest neighbour** v_c to a vertex v is that neighbour of v which has the shortest Euclidean distance to v. This relationship is denoted $v \rightarrow v_c$. Note that the closest neighbour may not exist, and that this relationship is not necessarily commutative.

A pair of vertices v_1 and v_2 is said to be **bridge** if $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_1$. A bridge is denoted $v_1 \leftrightarrow v_2$.

Note that the set of neighbours of a vertex v is defined so as to exclude the successor and predecessor of v, even though they may be geometrically close to v. This is done because edges to these vertices already exist in the gap. See Figure 5.2 for an illustration of the closest neighbour relationship.

The use of the associated planes in the definition of the neighbours of v eliminates many of the vertices that are not visible from v. A vertex u is *visible* from another vertex w if there exists a shortest path along any geodesic joining u and w that does not intersect the orthogonal projection onto the surface of any edge on the gap. Visibility is difficult to compute and the associated planes are a first order approximation. Excluding invisible vertices from the set of neighbours helps to prevent edges that overlap existing triangles from being considered for new triangles.

5.1.2 Computing Proximity

To compute the proximity data structure, the vertices of the gap are first inserted into an octtree. The vertices are also stored in a doubly–linked list. The predecessor and successor of each vertex are associated to it via pointers, and the associated planes and internal angle of each vertex are calculated.

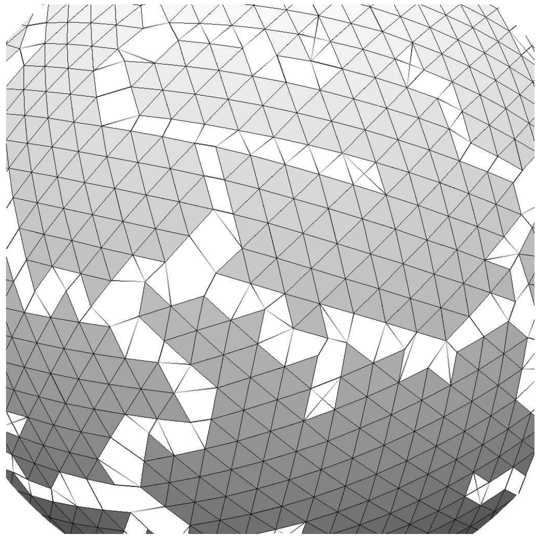

Figure 5.2: The gap with the closest neighbour relationships on a sphere. The thick uniformly coloured lines between two vertices u and v indicate $u \leftrightarrow v$. A dark line that fades to a lighter colour from a vertex u to a vertex v indicates $u \rightarrow v$.

The nearest neighbour is then computed for each vertex. Rather than finding the closest of the neighbours directly, the octtree is used to make the search sub–linear. A radius r is initialized to twice the distance between v and its predecessor. Then vertices within this distance of v are determined from the octtree. If this set contains neighbours, the nearest of these is selected. Otherwise, the radius is doubled, and the procedure is repeated. Iteration stops if a vertex is found or if r has become large enough to include all vertices (in which case no nearest neighbour exists).

5.1.3 Operations on Proximity

In order to implement the gap filling operations (which will be discussed later) the proximity data structure must support removal and insertion of vertices. Naively, these operations could be implemented by updating the octtree and the doubly–linked list and recomputing all of the closest neighbour relationships.

A more efficient implementation recomputes the nearest neighbours of only those vertices which are affected by the operation. For this purpose, the proximity data structure maintains a maximum radius. This radius is the maximum of the lengths of the edges of the gap polygon and the distances between the closest neighbours found to date.

When a vertex v is removed, it is deleted from the doubly–linked list and from the octtree. Then all the vertices within the maximum radius around v are determined from the octtree, and their nearest neighbours are recomputed. This procedure is equivalent to recomputing the nearest neighbours of all vertices in the sense that the same closest neighbour relationships will result.

To add a vertex v, it is inserted into the doubly–linked list and the octtree, and all vertices within the maximum radius of v have their nearest neighbours recomputed. However, a case exists in which the result of this procedure is not identical to recomputing all nearest neighbours (see Figure 5.3). When a

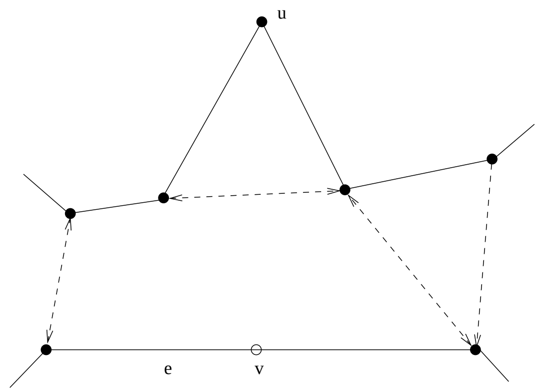

Figure 5.3: Before the edge e is split, vertex u has no nearest neighbour. When v is added, the relationship $u \to v$ should be created. The dashed lines indicate the proximity relationships.

vertex u exists in the data structure which has no closest neighbour before the addition of v, and which has v in its set of neighbours, the association $u \to v$ would not be made if u is far enough from v.

Fortunately, this case does not cause problems for the filling operations. Since the edges adjacent to u must necessarily have a small interior angle, the association $u \to v$ would not be used by any operations. Also, the distance between u and v would likely be large in comparison to other nearest neighbour relationships, so (u, v) would not be a suitable edge for filling the gap.

5.2 Filling Operations

The algorithm that fills the gap is based on a number of operations that act on a proximity data structure. Most of these operations generate triangles that lie in the gap and are added to the mesh covering the surface. Some of the operations add or remove vertices from the proximity. Other operations split the proximity into two or more components. The algorithm therefore deals with a number of independent proximity structures which collectively represent the part of the surface still to be covered by triangles.

Those operations which create new triangles use estimates of curvature of the surface to determine the lengths of new edges. The algorithm attempts to make new edges of optimal length, which is the product the curvature parameter ρ and the local radius of curvature.

At the start of the gap filling algorithm, a proximity data structure of all the vertices of the gap is computed and placed into an empty queue. The algorithm proceeds by extracting the items in this queue one after another. Filling operations are applied to an item P until it is **consumed** (i.e. the corresponding gap is completely triangulated). When the queue is empty, the gap is entirely filled.

The filling operations observe a certain precedence. If the preconditions of an operation are met by a proximity P, the operation is applied. Otherwise, the preconditions of the operation with the next highest precedence are evaluated for P, until an applicable operation is found. When an operation is applied, and P is not yet consumed, the operation with highest priority is attempted again. Iteration continues until P is consumed.

The operations are described below in order of precedence.

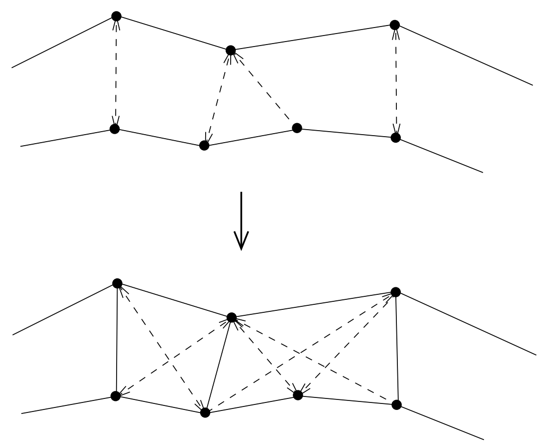

Figure 5.4: A section of the gap being handled by "subdivision on bridges." The triangle edges and the nearest neighbour relationships are shown before and after the operation.

5.2.1 Polygon Filling

The precondition for this operation is that P consists of exactly three or four vertices. The polygon filling operation consumes P. For three vertices, a triangle using the vertices is added to the mesh. For four vertices, the quadrilateral is split into two triangles. If the quadrilateral has a concave vertex v, it is split along the line joining v with the vertex opposite it.

If the quadrilateral is convex, it could be divided along either diagonal to produce two triangles. The diagonal is selected to maximize the minimum of the "qualities" of the two triangles, where the quality of a triangle is measured as the ratio of the smallest to largest edges.

5.2.2 Subdivision on Bridges

The second operation consists of subdividing a proximity into components. The proximity is divided into smaller proximities along bridges (Figure 5.4). The precondition of this operation is that at least one bridge exists.

The subdivision starts by identifying all the bridges $u \leftrightarrow v$ in the proximity such that two or more vertices separate u from v in both clockwise and counter–clockwise order. The components on the right and on the left of each bridge are then traversed once to enumerate their vertices by moving from one vertex to the next in counter–clockwise order around the perimeter, and crossing the bridge whenever a vertex that is on a bridge is reached.

When this operation is applied to a proximity, new proximities are computed for each component and added to the list, and the original proximity is removed from the list and deleted. If a bridge $v_1 \leftrightarrow v_2$ is longer than 150% of the local optimal edge length, the bridge is split by adding a vertex on the midpoint of v_1 and v_2, and settling this point to the surface.

5.2.3 X Filling

An X–**sequence** in a proximity is a sequence $S = \{v_1, v_2, v_3, v_4\}$ of adjacent vertices with the properties:

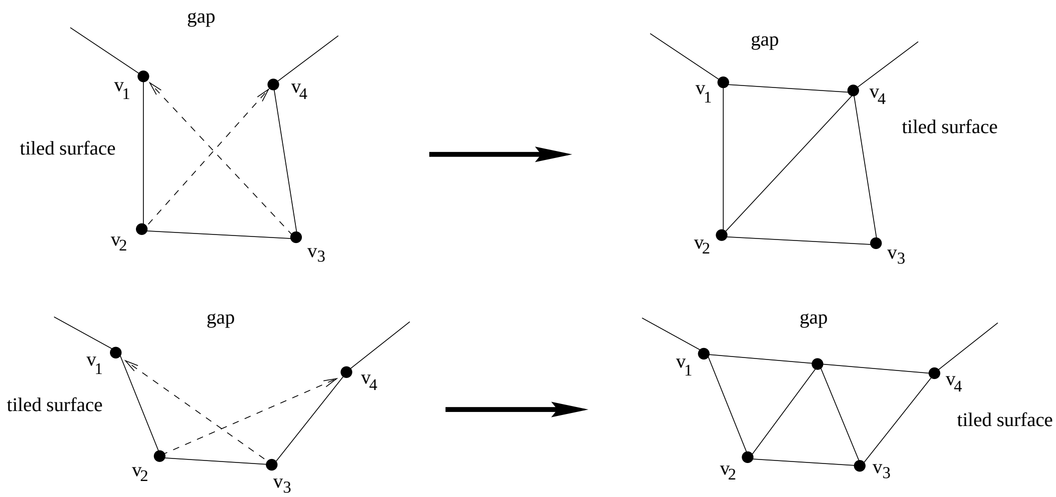

Figure 5.5: Proximity X's being filled with and without the addition of another vertex. The nearest neighbour relationships are indicated with dashed lines.

- $v_2 \rightarrow v_4$,
- $v_3 \rightarrow v_1$, and
- neither v_2 nor v_3 is the closest neighbour of any vertex not on S.

The appropriateness of the name should be clear from Figure 5.5.

The precondition of the X filling operation is the existence of an X configuration in P. The operation consists of identifying all of the X configurations in the proximity and adding either two or three triangles to fill them. Since the filling of an X changes the closest neighbour relationships of some of the remaining vertices in P, the X configurations are identified and processed one after another.

For each configuration, the distance between v_1 and v_4 is checked against the local optimal edge length. If the distance between the vertices is more than 150% of optimal, a vertex v is generated on the midpoint between v_1 and v_4. Three triangles in a fan configuration around v are added to the mesh, v_2 and v_3 are removed from P, and v is added to P between v_1 and v_4.

If the distance between v_1 and v_2 is not too large, the vertices on S are covered by the best pair of two triangles (as in the polygon filling operation), and the vertices v_2 and v_3 are removed from P.

This operation is motivated by two observations. The vertices of an X frequently occur such that the quadrilateral containing them is close to square. Splitting this quadrilateral therefore generates good triangles. Also, the restriction that no other vertex of the proximity may have v_2 or v_3 as nearest neighbour implies that these vertices are relatively isolated and that the X can be triangulated independently.

5.2.4 Ear Filling

This operation is similar to ear cutting in the first phase. In this context, an **ear** consists of a sequence $S = \{v_1, v_2, v_3\}$ such that $v_1 \leftrightarrow v_3$ and no vertex of P has v_2 as its closest neighbour.

The precondition for the ear filling operation is the existence of an ear configuration in P. Ears are filled much the same as X's. The distance between v_1 and v_3 is split if it is too long. If it is split, two

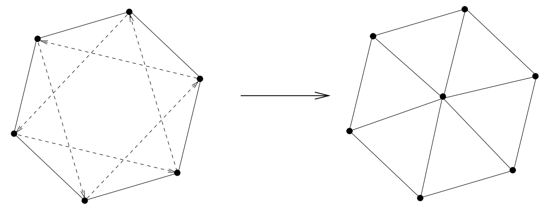

Figure 5.6: A convex polygon before and after the convex polygon filling operation is applied.

triangles are added to the mesh using the diagonal containing v_2 and the new vertex. Otherwise, just one triangle is added. In either case v_2 is removed from P, and if two triangles were added to the mesh the new vertex is inserted into P between v_1 and v_3.

The triangles created by filling ears are generally not as close to equilateral as those produced from X configurations. Therefore, only the first ear found is filled in the hope that the changes introduced into P will make one of the higher precedence operations applicable.

5.2.5 Convex Polygon Filling

The convex filling operation closes P, like the quadrilateral filling operation. The precondition is that every vertex of P is convex. A new vertex v is placed at the average of the vertices of P and settled to the surface, and a fan of triangles is created around this point to cover all the portion of the surface represented by P (see Figure 5.6). P is consumed by this operation.

Adding a vertex in the middle of the P and triangulating it with a fan may generate triangles a little smaller than optimal, but they will be well–shaped. Note that the number of vertices in a convex proximity cannot be too large because the distance across the gap is limited by the triangle growing phase, and because the higher precedence filling operations will tend to reduce the number of vertices in P before this operation is considered.

5.2.6 Relaxed Ear Filling

A relaxed ear has the same properties as an ear except that either $v_1 \to v_3$ or $v_3 \to v_1$ is acceptable rather than requiring that $v_1 \leftrightarrow v_3$. The relaxed ear filling procedure is otherwise the same as ear filling. Note that it is applied only if the operations described above are not applicable.

5.2.7 Concave Vertex Bisection

If none of the other operations apply, P must have at least one concave vertex. Concave vertex bisection has no other precondition and is therefore always applied when none of the other operations' preconditions are met.

The vertex v_1 that has the largest interior angle is selected from P, and a corresponding vertex v_2 in the proximity is found such that the line segment (v_1, v_2) most closely bisects the interior angle at v_1. P is split into two components along this line segment. If the distance between these vertices is more than 150% of the optimal edge length the edge is split by settling the midpoint to the surface. The two new components are added to the proximities list and P is deleted.

5.3 Results and Conclusions

The different filling operations are not used with the same frequency. The higher priority operations, like X filling and subdivision on bridges, occur much more often than relaxed ear filling and concave vertex bisection. For example, a sphere model with ρ (the ratio of optimal edge length to radius of curvature) set to $\frac{1}{7}$ produced a gap with 383 vertices. A total of 216 operations were required to fill this gap: 108 polygons filled, 22 subdivisions, 57 X's filled, 41 ears filled, and 4 convex polygons filled. The remaining operations are required when the surface has areas of rapidly changing curvature.

The proximity method performs well in practical tests even though it has not been proven to always be able to fill the gap. The tests performed to validate the method will be described in detail in the next chapter.

A problem similar to that of filling the gap was addressed by Turk and Levoy [31]. Their "mesh zippering" algorithm was developed to merge to triangular meshes created from range scanner data. The meshes from different view points have overlapping areas. The triangles of mesh A that completely overlap mesh B are deleted. Then mesh A is "clipped" against the boundary of mesh B in a process which adds new vertices and splits overlapping triangles.

The clipping process can result in arbitrarily small or narrow triangles. Turk and Levoy correct the mesh in a post–process that collapses narrow triangles and retriangulates the areas surrounding very small triangles.

It is not clear that their technique can handle bifurcations in the overlapped regions, which is an important aspect for the gap filling problem. Also, since mesh zippering uses a post–process to correct the quality of the mesh rather than addressing this issue directly, it is possible that the proximity technique generates a higher quality mesh.

Chapter 6

Results

The algorithm described in this thesis, hereafter called the "creep algorithm" because of the way in which the mesh grows during the triangle growing phase, was implemented in 15,000 lines of C++ code. The implementation was tested with a number of models, which are depicted in Figure 6.1. The two phases of the algorithm are discussed qualitatively and quantitatively below, and the resulting triangulations are analyzed and compared to a standard implicit surface polygonizer [5].

6.1 The Test Models

The cube, peanut, sphere, and torus models are composed of blobs using Blinn's potential [3]. The cone model is defined procedurally by dividing space into three regions by two parallel planes with normals along the axis of the cone. In the top and bottom regions, the implicit surface is a section of a sphere, and in the center region, it is a section of a cone.

The pretzel and spiral models were built using a simple implicit surface modelling primitive. A B–spline is used to define a curve, which is then approximated by a number of line segments. The Blinn potential is analytically integrated over the line segments and the implicit function is the superposition of these potentials.

6.2 Comments on Triangle Growing

It was found that accurate curvature measurement is critical to the triangle growing phase (see Figure 6.3). If the curvature is not measured accurately, small triangles may be placed close to large ones and this situation will be handled badly by the proximity phase, which implicitly expects the local density of vertices on both sides of the gap to be similar. Unfortunately, accurate curvature computation is costly in terms of the execution time of the algorithm.

The creep algorithm cannot handle a high rate of change of curvature. If the peanut model is changed so that it looks more like two spheres connected by a thin strand, the area on one sphere from which the strand emanates can be triangulated improperly during the triangle growing phase. When this happens, a triangle is created which covers this area, so the strand and the other sphere are never reached. Note

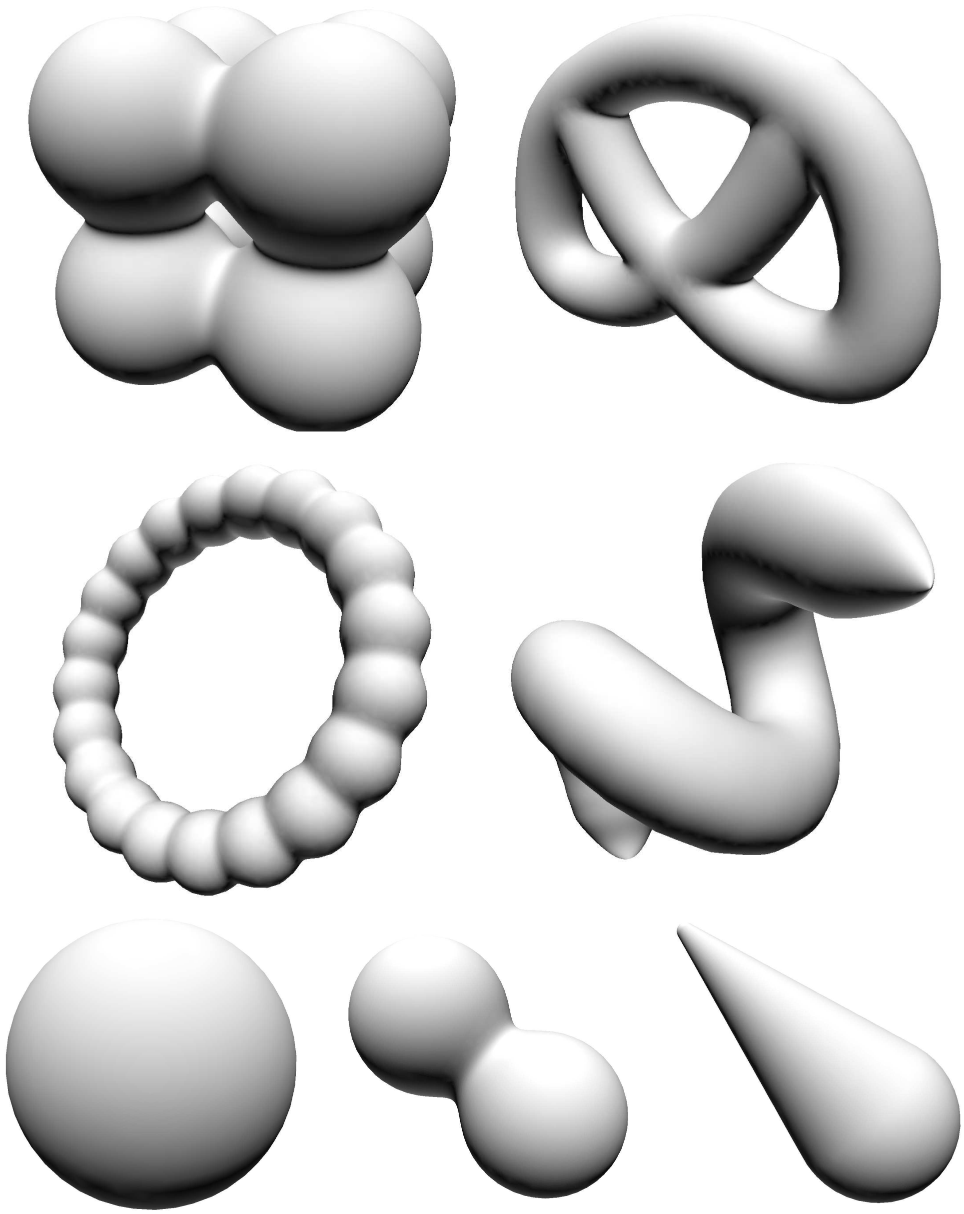

Figure 6.1: The models used to test the creep algorithm: cube (of eight spheres), pretzel, torus, spiral, sphere, peanut, and cone. These images are generated from the results of the algorithm with a ρ value of $\frac{1}{7}$.

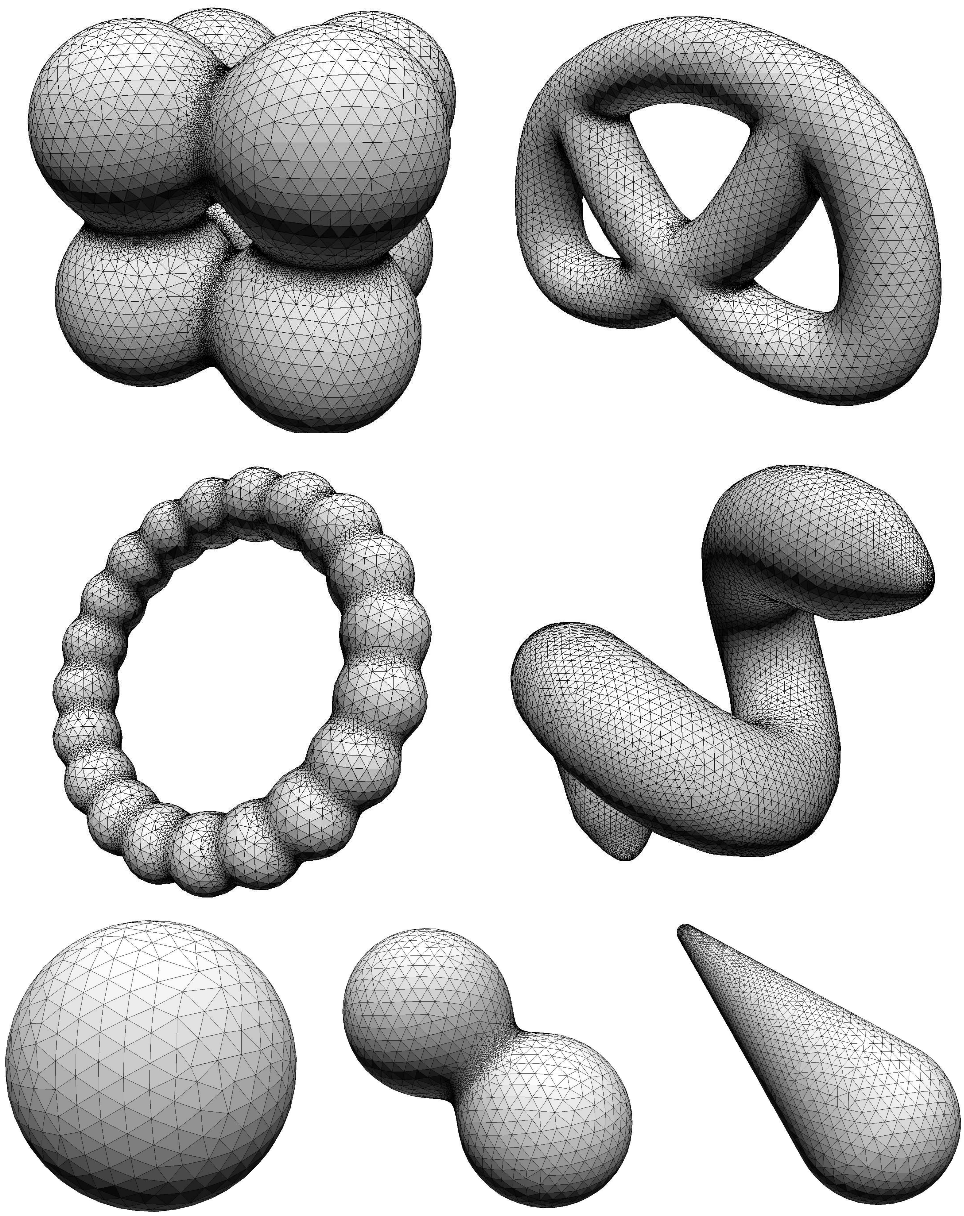

Figure 6.2: Images of the models with showing the mesh.

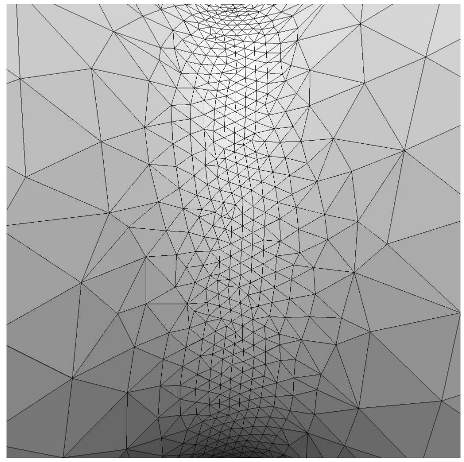

Figure 6.3: A close–up of the high curvature area of the peanut model. Accurate curvature measurement is necessary during the triangle growing phase to create appropriately sized triangles.

	Cone	Cube	Peanut	Pretzel	Sphere	Spiral	Torus
Triangles	6,362	56,292	6,546	40,924	1,476	18,072	46,064
Total Ops.	1,167	13,301	1,446	8,503	228	3,931	10,285
Quadrilaterals	540	4,891	551	3,474	106	1,631	3,680
Subdivisions	102	1,214	141	805	22	369	884
X's	248	3,024	338	1,840	57	847	2,574
Ears	256	3,389	344	2,032	37	959	2,539
Convex Polys.	21	677	60	315	6	76	543
Relaxed Ears	0	98	9	33	0	44	57
Bisections	0	8	3	4	0	5	8

Table 6.1: A summary of proximity filling operations (see Section 5.2) for the test models.

that cell–based tilers that follow the surface instead of sampling all space can have the same problems.

The number of iterations over the active edge list is small for all the test models. This number seems to depend on the model as well as on number of polygons in the triangulation. The sphere requires only two iterations when covered with 1,500 triangles, but five iterations when covered with roughly 12,000 triangles. The peanut requires three iterations when covered with 6,500 triangles. The cube model has the most triangles at 56,000 but only requires five iterations.

6.3 Comments on Proximity

As Table 6.1 attests, the distribution of proximity operations is relatively stable across the different test models. The total number of proximity operations required to complete the triangulation is approximately one fifth of the final number of triangles. The operations in the last three rows, which do not control the shapes and sizes of the triangles they produce as tightly as the other operations, do not need to be used very often.

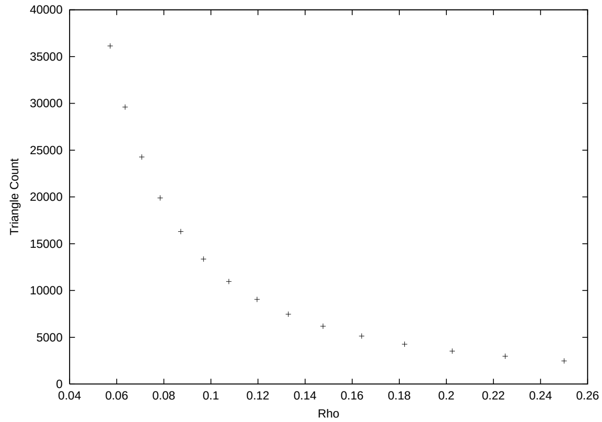

Figure 6.4: A plot of triangle count vs. ρ for the peanut model.

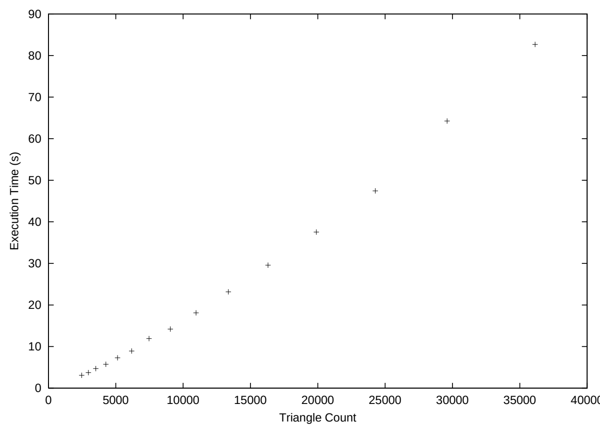

Figure 6.5: A plot of execution time vs. triangle count for the peanut model.

The proximity phase generally requires less execution time than the triangle growing phase. Proximity is faster because it requires fewer implicit function, normal, and curvature evaluations than the first phase.

6.4 Triangle Counts and Timings

Since the curvature parameter ρ is proportional to the optimal edge length of the triangles in the mesh produced by the creep algorithm, and since the area covered by a triangle is proportional to the square of the edge length, the number of triangles required to cover a surface is proportional to $\frac{1}{\rho^2}$.

Figure 6.4 shows a plot of triangle count versus ρ for the peanut model. Regression analysis of a log–log plot of triangle count versus ρ has a slope of -1.83 and a linear correlation of 0.9997. The inverse square relationship is therefore verified, and in this sense, the algorithm works as designed.

The total execution time of the creep algorithm (Table 6.2) is determined by a number of factors including the curvature parameter ρ, the surface area and amount of curvature of the surface, and the cost of evaluating the implicit function. Figure 6.5 shows a plot of execution time versus number of

	Cone	Cube	Peanut	Pretzel	Sphere	Spiral	Torus
Phase 1	12.3	130.7	7.62	530.5	0.77	90.3	65.4
Phase 2	1.4	58.9	1.97	56.7	0.26	11.7	43.7
Total	13.7	189.6	9.59	587.2	1.03	102.0	109.1

Table 6.2: The timings, in seconds, of the creep algorithm applied to the test models with $\rho = \frac{1}{7}$.

triangles for the peanut model with different meshes being created by varying ρ (other models had similar plots). Execution was timed on a 500MHz Pentium III machine running Linux. Since many of the parts of the algorithm use searches which would be $\mathrm{O}(\log n)$ on randomized data for every polygon, the execution time is slightly superlinear in the number of polygons.

The Bloomenthal polygonizer [5] is much faster than this implementation of the creep algorithm. Whereas creep took 180 seconds to generate a mesh of 56,292 triangles for the cube model, the Bloomenthal tiler required less than three seconds to produce a similarly sized mesh. However, comparing meshes of the same size is not fair to the creep algorithm, as demonstrated in the next section.

6.5 Mesh Quality

In order to compare the output of the creep algorithm to the Bloomenthal polygonizer, two measures of the "quality" of a mesh were developed.

The first measure is intended to express how well a triangulation captures the geometry of the implicit surface. The error in the approximation of a surface S with a mesh can be expressed as:

$$E = \oint_S d^2 \hat{n} \cdot d\vec{S}$$

where $\hat{n}$ is the unit normal vector to S and d is the perpendicular distance of the nearest point on the triangulation to the surface. This error was computed by stochastic sampling. For each sample, a triangle was selected randomly from the mesh by weighting the triangles with their areas. Then a point was selected uniformly randomly on the triangle. The nearest point on the surface was computed by intersecting the line generated by the point on the triangle and gradient of the implicit function at this point with the surface.

Table 6.3 compares the error for meshes generated by creep with meshes of the same or more triangles generated by the Bloomenthal tiler. See Figure 6.6 for a visual comparison. In each case, every triangle was sampled at least 10 times on average. Note that the error values for different models cannot be meaningfully compared as they depend on the size of the models in their object coordinates, and on the number of polygons in the mesh.

According to the measure above, the quality of the meshes produced by the creep algorithm is better than those produced by the Bloomenthal tiler in all cases except the cube. In the other cases, the Bloomenthal tiler required between 1.5 and 2 times the number of triangles to produce a mesh of equal quality. However, even when producing twice as many triangles, the Bloomenthal algorithm is far faster than the creep algorithm.

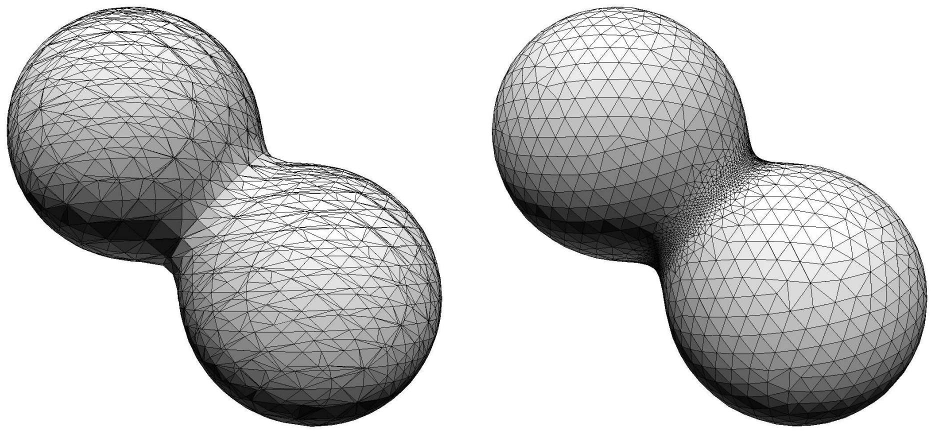

Figure 6.6: Meshes generated by the Bloomenthal tiler and the creep algorithm. Both meshes have approximately 6,500 triangles.

Model	Triangles	Creep Error	Bloom. Error
cone	6,400	1.54×10^{-4}	3.21×10^{-4}
cube	56,300	9.76×10^{-4}	5.08×10^{-4}
peanut	6,500	4.13×10^{-4}	7.14×10^{-4}
pretzel	15,100	2.11×10^{-3}	6.12×10^{-3}
sphere	1,500	1.94×10^{-3}	1.60×10^{-2}
spiral	11,400	3.30×10^{-5}	5.91×10^{-4}
torus	13,600	6.17×10^{-2}	8.97×10^{-2}

Table 6.3: A comparison of the error of meshes produced by creep and the Bloomenthal tiler. The triangle counts for each pair of meshes differed by at most 2%.

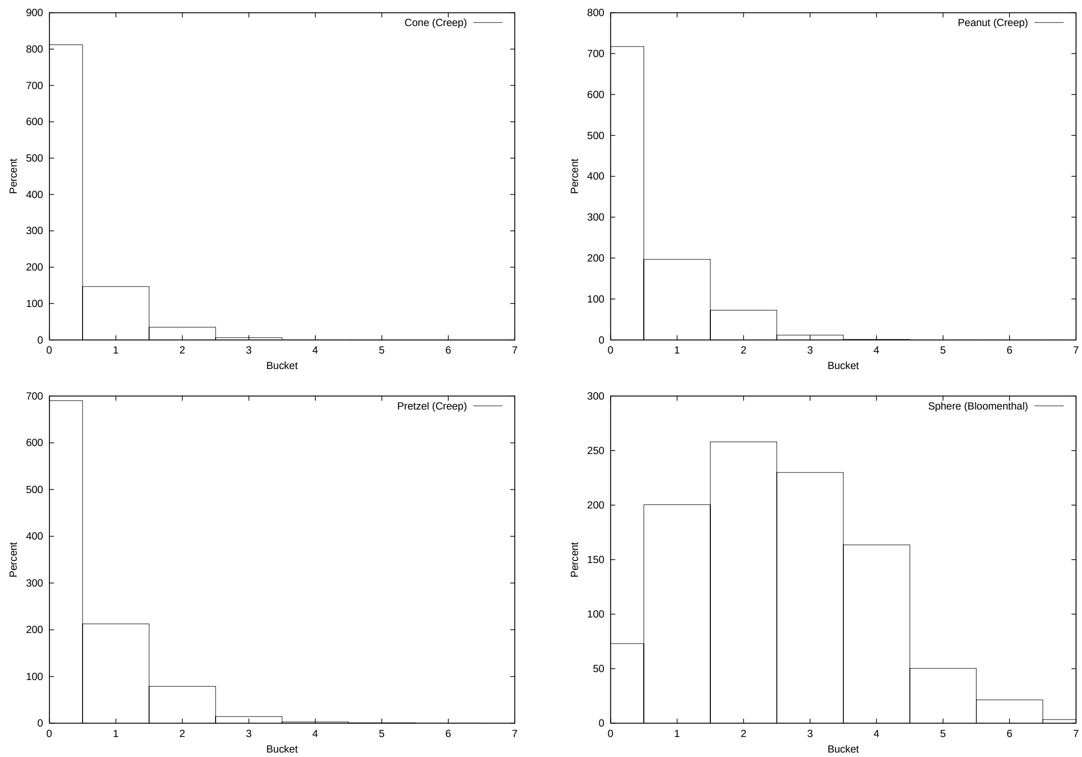

Figure 6.7: Graphs of absolute deviation angle distribution. The first bucket contains all triangles with a maximum deviation from the optimal 60° of less than 15°. The next is 15° to 30° and so on.

The second measure of the quality considers the shape of the triangles produced. The largest absolute deviation of any internal angle from 60° is computed for each triangle, and the triangles are sorted into buckets based on this angle.

Figure 6.7 shows bar graphs of the result. The graph for the Bloomenthal tiler is based on a tiling of the sphere model, but graphs for the other models look very similar. The creep algorithm clearly generates triangles that are much closer to equilateral.

Chapter 7

Conclusions

A new implicit surface polygonizer has successfully been designed and implemented. The polygonizer produces a mesh without T–vertices, unlike some adaptive cell–based tilers. The mesh consists of high quality, near–equilateral triangles. The size of the triangles is adaptive, so that more detail is placed where required in regions of high curvature. By one error measure, a mesh produced by the new algorithm fits implicit surfaces better than an equally sized mesh produced by a standard polygonizer [5].

Because it performs estimates of curvature and requires the gradient of the implicit function, the new algorithm executes about two orders of magnitude slower than the standard polygonizer. However, the new algorithm can generate meshes of about 100,000 triangles in a few minutes on standard contemporary hardware, and these meshes are of higher quality.

The new algorithm handles only G^1 surfaces. It shares this restriction with other sophisticated polygonizers like the shrinkwrap algorithm [8], but not with some cell–based polygonizers.

The new algorithm is not as robust as the cell–based polygonizers. It can fail on surfaces which have a very high rate of change of curvature along a geodesic or in areas where the solid represented by the surface is very thin. The cell–based polygonizers will produce a mesh on such surfaces, but that mesh may fail to capture the topology and geometry of the surface correctly.

7.1 Future Work

The approach of this thesis could be extended to include non–G^1 surfaces. Edges and points on the surface would have to be identified and perhaps triangles could be generated around these features so that the rest of the surface could be tiled with the algorithm described here. The algorithm could similarly be extended to non–manifold surfaces.

The linear dependence of triangle size on the local radius of curvature should be investigated further. In a sense, such a dependence produces the best–fitting triangulation, but this is not necessarily desirable for applications in computer graphics. Perhaps the triangle size should have a lower bound because the amount of detail produced by linear dependence on radius of curvature is not required.

The technique developed to fill the gap in the second phase of this algorithm could be applied to other problems. One such problem is the production of triangular meshes from laser range finder data.

When making a complete model of an object, several data sets are required in order to cover the object from all sides. The proximity technique could be used in one step of seaming triangulations computed from these data sets.

The triangulation as produced by the creep algorithm still sometimes has narrow triangles. In order to correct this, the curvature computed when generating the triangles could be used to create a post–process which uses the relaxation techniques to regularize the mesh.

Appendix A

Curvature

Suppose a surface S is defined by $\vec{x}(u,v)$ and P is a point that lies on S. Let $\hat{n}$ be the normal to S at P and let C be some curve embedded in S that passes through P. In the parameter space of S, C is defined by $u = u(s)$ and $v = v(s)$ where s is the arc length parameter.

The second fundamental form of S is:

$$\kappa_{\hat{n}} = L(u,v)\left(\frac{\mathrm{d}u}{\mathrm{d}s}\right)^2 + 2M(u,v)\frac{\mathrm{d}u}{\mathrm{d}s}\frac{\mathrm{d}v}{\mathrm{d}s} + N(u,v)\left(\frac{\mathrm{d}v}{\mathrm{d}s}\right)^2$$

where

$$\begin{aligned} L &= \hat{n} \cdot \frac{\partial^2 \vec{x}}{\partial u^2} \\ M &= \hat{n} \cdot \frac{\partial^2 \vec{x}}{\partial u \partial v} \\ N &= \hat{n} \cdot \frac{\partial^2 \vec{x}}{\partial v^2} \end{aligned}$$

The second fundamental form is invariant under parameter transformations. Since it is only dependent on the direction vector of a curve embedded in S, all curves that have the same tangent at P will produce the same $\kappa_{\hat{n}}$.

The second fundmantal form is equal to the component, κ, of the curvature of C, along $\hat{n}$. The curvature of a curve at a point is the reciprocal of the radius of the largest osculating sphere at that point. A geodesic of a surface may be defined as a curve on the surface that has curvature vector entirely along the normal to the surface.

A.1 Estimating Curvature

To estimate the radius of curvature at a point P of an implicit surface, the radius of curvature of several geodesics passing through the point is computed, and the smallest of these is selected as the result. Since the geodesics are difficult to compute (especially without a parametrization), the geodesics are assumed to locally lie in the planes containing the normal vector and the tangent vector at P.

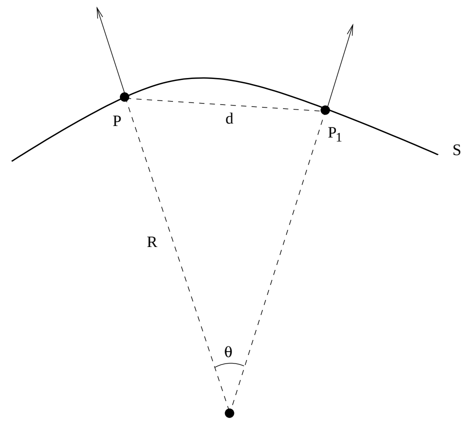

Figure A.1: Estimating the radius of curvature via the isosceles triangle method.

In order to estimate the radius of curvature of the geodesic, the normal to the surface is computed at P and at a nearby point P_1 on the geodesic. The angle between these vectors, θ, is taken as the odd angle of an isoceles triangle, and the distance between P and P_1, d, is taken as the length of the edge of the triangle opposite to θ (see Figure A.1). The length R of the equal edges is computed from θ and d and taken as the radius of curvature.

Bibliography

[1] Eugene L. Allgower and Phillip H. Schmidt. An algorithm for piecewise-linear approximation of an implicitly defined manifold. *SIAM Journal on Numerical Analysis*, 22(2):322–346, April 1985.

[2] Bruce G. Baumgart. Winged edge polyhedron representation. Technical Report CS-TR-72-320, Stanford University, Department of Computer Science, October 1972.

[3] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[4] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[5] Jules Bloomenthal. An implicit surface polygonizer. In Paul Heckbert, editor, *Graphics Gems IV*, pages 324–349. Academic Press, Boston, 1994.

[6] Jules Bloomenthal and Keith Ferguson. Polygonization of Non-Manifold implicit surfaces. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 309–316. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[7] H. Blum. A transformation for extracting new descriptors of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Forms*, pages 362–380. MIT Press, Amsterdam, 1967.

[8] Andrea Bottino, Wim Nuij, and Kees van Overveld. How to shrinkwrap through a critical point: an algorithm for the adaptive triangulation of iso-surfaces with arbitrary topology. In *Implicit Surfaces '97 Proceedings*, pages 53–72, October 1997.

[9] Jonathan W. Brandt and V. Ralph Algazi. Continuous skeleton computation by Voronoi diagram. *Computer Vision, Graphics, and Image Processing. Image Understanding*, 55(3):329–338, May 1992.

[10] Luiz Henrique de Figueiredo, Jonas de Miranda Gomes, Demetri Terzopoulos, and Luiz Velho. Physically-based methods for polygonization of implicit surfaces. In *Proceedings of Graphics Interface '92*, pages 250–257, May 1992.

[11] A. L. Dixon. The eliminant of three quanitcs in two independent variables. *Proceedings of London Mathematical Society*, 6(49):473–492, 1908.

[12] Eric Ferley, Marie-Paule Cani-Gascuel, and Dominique Attali. Skeletal reconstruction of branching shapes. *Computer Graphics Forum*, 16(5):283–293, December 1997.

[13] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 7(4):175–179, June 1978.

[14] M. Held. Efficient and reliable triangulation of polygons. In *Computer Graphics International*, 1998. `www.cosy.sbg.ac.at/~held`.

[15] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized voroin diagrams using graphics hardware. In Alyn Rockwood, editor, *SIGGRAPH 99 Conference Proceedings*, Annual Conference Series, pages 277–285. ACM SIGGRAPH, Addison Wesley, 1999.

[16] Devendra Kalra and Alan H. Barr. Guaranteed ray intersections with implicit surfaces. *Computer Graphics*, 23(3):297–306, July 1989.

[17] L. Lau, S.-W. Lee, and C. Suen. Thinning methodologies: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(9):869–885, 1992.

[18] D. T. Lee. Medial axis transform of a planar shape. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-4:363–369, 1982.

[19] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[20] Paul Ning and Jules Bloomenthal. An evaluation of implicit surface tilers. *IEEE Computer Graphics and Applications*, 13(6):33–41, November 1993.

[21] H. K. Pedersen. Decorating implicit surfaces. *Computer Graphics*, 29(Annual Conference Series):291–300, 1995.

[22] W. T. Reeves. Particle systems – A technique for modeling a class of fuzzy objects. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 359–376, July 1983.

[23] T. W. Sederberg and F. Chen. Implicitization using moving curves and surfaces. *Computer Graphics*, 29(Annual Conference Series):301–308, 1995.

[24] Jos Stam. Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 395–404. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[25] Barton T. Stander and John C. Hart. Guaranteeing the topology of an implicit surface polygonization for interactive modeling. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 279–286. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[26] Richard Szeliski and David Tonnesen. Surface modelling with oriented particle systems. *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 185–194, 1992.

[27] Charles C. Tappert, Ching Y. Suen, and Toru Wakahara. The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-12(8):787–808, August 1990.

[28] Robert E. Tarjan and Christopher J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17(1):143–178, February 1988.

[29] David Love Tonnesen. *Dynamically Coupled Particle Systems for Geometric Modelling, Reconstruction, and Animation*. PhD thesis, University of Toronto, 1998.

[30] Greg Turk. Re-tiling polygonal surfaces. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 55–64, July 1992.

[31] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*, Computer Graphics Proceedings, Annual Conference Series, pages 311–318. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[32] C. W. A. M. van Overveld and B. Wywill. Shrinkwrap: an adaptive algorithm for polygonizing an implicit surface. Technical Report 93/514/19, University of Calgary, March 1993.

[33] Luiz Velho. *Piecewise Descriptions of Implicit Surfaces and Solids*. PhD thesis, University of Toronto, 1994.

[34] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 269–277, 1994.

[35] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.