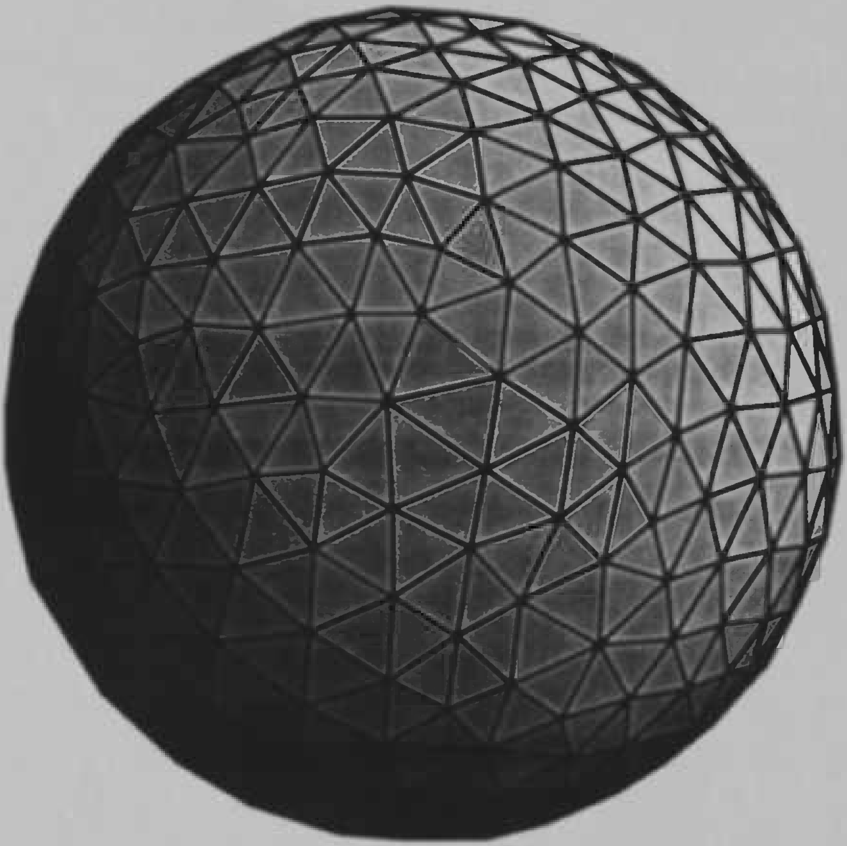
Triangulation of implicit surfaces

by E.R. Kronemijer

under supervision of Dr. G. Vegter



September 1998

Rijks*Universiteit* Groningen

RuG

Foreword

To whom this may concern

When I started studying Computer Sciences in 1991 my main goal was becoming an even better programmer then I thought I already was. During the first year I discovered that most of the subjects covered were of a mathematical or theoretical nature and not about programming. Now I am never in need of a pocket calculator and consider myself a fairly good programmer.

During the seven years at the RuG I have studied the wonders of computer science and what it can do for you. I also studied the wonders of beer and what it makes you think you can do! And now I think it is time to face the “real world”. Actually I started working some months ago, so the “real world” has started for me already.

I would like to thank my coach Gert Vegter for his time and support in helping me write this paper. I would also like to thank my parents for both their moral and financial support, as well as for being who they are. Last but not least I would like to thank you, Arijaan, for putting up with me the last year, which has not always been easy.

Groningen, September 1998

Evert Kronemeijer

1. Introduction

Computer Aided Geometric Design deals with those mathematical properties of curves and surfaces that are useful in applications like CAD-systems, fluid mechanics (jet wing design), interpolation and approximation of measured data, etc. An important issue is the triangulation of smooth surfaces.

Triangulations of surfaces are frequently used in mesh generation for finite element analysis. Moreover, many graphics (rendering) systems use triangles or polygons as primitives.

1.1 Problem description

This paper deals with approximating smooth implicit curves and surfaces. Implicit curves are defined by a function $f(x,y)$. The curve consists of all points (x,y) for which $f(x,y) = 0$. Implicit curves will be approximated by straight line segments, of which the endpoints lie on the curve.

An implicit surface is similarly defined by a function $f(x,y,z)$. The surface consists of all points (x,y,z) satisfying $f(x,y,z) = 0$. Implicit surfaces will be approximated by constructing a triangulation, of which the vertices lie on the surface (for finite element analysis it is important that vertices should be on the surface, or as close to it as possible).

In the sequel we assume that f is C^1, i.e. differentiable with continuous derivatives. According to the "implicit function theorem" [6], an implicit curve or surface is smooth, when $\nabla f(p) \neq 0$ for all p satisfying $f(p) = 0$. Here $\nabla f(p)$ is the gradient of f at p.

1.2 Outline of this paper

A number of techniques for polygonizing or triangulating curved surfaces (not necessarily implicit) is known in the literature. Two of these, originating from Chew [1] and Bloomenthal [2], are treated in some detail in chapter 2.

The problem of approximating implicit curves in the 2-dimensional plane is considered in chapter 3. I devised, implemented and tested a total of 4 methods. The results are presented in chapter 3.

A new, simple and fast method for triangulating smooth implicit surfaces in 3-space is introduced in chapter 4. It uses a fixed mesh size, and it produces good-quality triangulations.

Finally, chapter 5 presents the conclusions.

Appendix A covers the rootfinding methods mentioned in this paper.

2. Two existing methods

In this chapter I discuss two methods for constructing triangulations of curved surfaces: a method by Chew [1], based on the constrained Delaunay triangulation (CDT, [3]); and an octree-based method by Bloomenthal [2].

2.1 Chew : guaranteed-quality mesh generation for curved surfaces

Chew's method starts off with an initial, crude, triangulation from which the CDT is computed. A CDT is essentially a Delaunay triangulation in which some edges are fixed ahead of time.

Next, all triangles are graded. A triangle passes the grading test if it is both well-shaped and well-sized. A triangle is well-shaped if its smallest angle is greater than 30 degrees. A user supplied grading function decides whether triangles are well-sized. Any grading function is allowed, as long as making triangles smaller will eventually make them pass. Typical size criteria would be based on error estimates and/or surface curvature.

Triangles that do not pass the test are refined by adding a vertex at their circumcenter, and updating the CDT to include this new vertex. This is repeated until all triangles are graded.

To make this work on a curved surface, the notions of circumcenter and circumcircle have to be defined for curved surfaces. Chew defines these as follows: all spheres trough the three corners of a triangle have centres lying on one line (this is the line orthogonal to the triangle plane and intersecting it in its regular, 2D-circumcenter). The circumcenter is defined as the intersection of this line with the curved surface; the circumcircle is defined as the region of the surface lying inside the corresponding sphere.

However, the line may not intersect the surface at all, or more than once. Chew argues that these difficulties cannot occur if the initial triangulation is 'reasonable' in the following sense: in the region formed by the union of circumcircles of two adjacent triangles, the surface normals should not vary by more than $\pi/2$. This condition also guarantees that two adjacent triangles have consistent circumcircles: given edge *bc* with adjacent triangles *abc* and *bcd*, *d* is within the circumcircle of *abc* iff *a* is in the circumcircle of *bcd*.

Strong points of this algorithm are:

- The algorithm produces well-shaped triangles.
- Boundaries are respected; though these have to be present in the initial triangulation.
- Intersecting a line with the surface is in all practical cases a feasible operation, although it may be expensive.
- To determine if a point is inside a circumcircle –an elementary operation for building Delaunay triangulations– calculation of 3-space distance suffices.

The main disadvantage of this algorithm is that an initial, 'crude', triangulation has to be constructed first, satisfying the condition on the bounded variation of normals. Thus, a triangulation cannot be built up ‘from scratch’. Also, (crude) boundaries have to be present in the initial triangulation.

2.2 Bloomenthal : Polygonization of implicit surfaces

Bloomenthal surrounds the surface with an octree of cubes, at whose corners the implicit function is sampled. The surface is defined by the zero set of a continuous function *f*, or, more generally, the set of points that separate positive *f* from negative *f*. A cube intersects the surface if it contains a corner for which *f* evaluates positively and a corner for which *f* evaluates negatively.

The octree can be constructed converging to the surface, or tracking it. When converging to the surface, the octree is constructed from an initial root cube. The root cube should be large enough to contain the complete surface to be triangulated. It then converges to the surface by recursively subdividing those cubes that intersect the surface. The subdivision process halts when a pre-set recursion depth is reached.

When tracking the surface, an initial 'seed cell' is established that intersects the surface, and is small compared to surface detail. New cells propagate along existing cell edges that intersect the surface.

Eventually, for each cell a polygonal representation of the surface is constructed, with polygon vertices lying on cell-edges, and polygon edges lying in the cell faces. Surface vertices are computed for each cell-edge having opposite-signed corners, by bisection along the edge. These surface vertices are connected along the faces to form polygons. The polygons may be subdivided into triangles to form a triangulation.

Unfortunately, the 'positive' corners (at which *f* evaluates positively) and the 'negative' corners of a cube cannot always be separated by a single plane. When this problem arises, Bloomenthal proposes to partition the cube into 12 tetrahedrons (introducing one new vertex at the centre of the cube), and to construct polygonal representations of the surface from these tetrahedrons. Since for a tetrahedron, positive and negative corners can always be separated by a single plane, this always yields a consistent representation of the surface. However, a different choice of partitioning will result in a different representation of the surface.

Advantages of this algorithm are:

- It works for very general functions.
- The quality of the approximation can be controlled by the recursion depth.
- The algorithm can be made adaptive, by making the recursion depth dependent on surface curvature.

Problems connected to this algorithm are:

- There is no control over triangle shapes. They may get very small or skinny.
- Even if *f* evaluates to the same sign in all cube corners, the surface may still penetrate the cube. Since such cubes are not processed further, this will result in a gap in the surface representation.
- The tracked version needs a 'seed cell'.
- Boundaries must always coincide with cube faces.

3. Approximating implicit curves in the plane

3.1 Problem

An implicit curve in the *xy*-plane is formed by those points (x,y) satisfying the equation $f(x,y) = 0$, for some C^1 function f. When f is a differentiable function and $\nabla f(x_0,y_0) \neq 0$ for all (x_0,y_0) satisfying $f(x_0,y_0) = 0$, the implicit curve will be smooth. The problem we intend to solve is, given such a function, approximate the corresponding curve by connected straight-line segments.

3.2 Four methods

As a starting point I have taken a 2-dimensional version of the octree-method as described by Bloomenthal [2]. As mentioned in chapter 2, the converging octree method will cause 'holes' in the approximated surface when a cube intersects the surface while f evaluates to the same sign in all its corners. The two-dimensional version, using squares instead of cubes, suffers from the same problem; this is illustrated in figure 3.1.

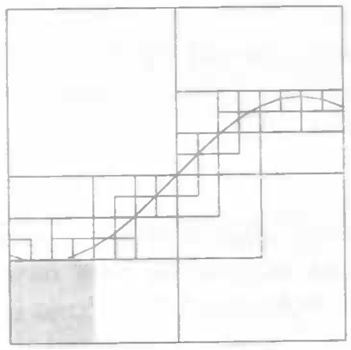


Figure 3.1 Part of a sine-curve. The shaded square was not subdivided because its corners all evaluate to the same sign.

Increasing precision, that is increasing recursion depth, does not solve this problem, because the problem can occur at any depth. It is possible to check whether pieces of the curve are missing: each segment should have two neighbours, unless it is a boundary segment; but this can only be checked in a sequential pass after the recursive subdivision has been done. Since an approximation of any reasonable implicit curve will essentially be a (connected) sequence of segments, I decided to investigate only algorithms that track the curve. So as a starting point I took a two-dimensional version of Bloomenthal's tracking algorithm, and called it method 1.

3.2.1 Method 1

Method 1 requires an input square that intersects the curve and is small relative to the curve detail. New squares are 'grown' sharing edges that intersect the curve. Zero-points of the function are calculated on the edges of the squares and are connected to form segments. The zero-points are calculated using the bisection method. An example is shown in figure 3.2.

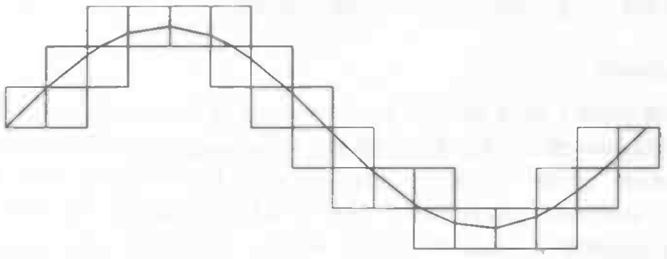


Figure3.2 Part of a sine-curve, approximated by method 1.

This method is easy to implement and always produces a connected sequence of segments. It may however produce arbitrarily short segments, because of the rigid placing of the squares in a grid.

3.2.2 Method 2

To prevent the production of arbitrarily short segments, I decided to place the squares adaptively, as follows: suppose the most recently calculated segment is AB. Then place the square as in figure 3.3.

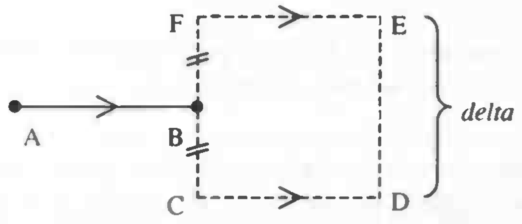


Figure 3.3 Adaptive placement of a square.

Check which of the edges CD, DE or EF intersects the curve; calculate a zero-point of *f* on this edge and connect it to B to form a new segment. This is method 2, illustrated in figure 3.4.

Method 2 (as well as 3 and 4) requires an initial segment instead of an initial square. The zero-points are again calculated by the bisection method.

Table 3.2 (page 9) lists minimum and maximum segment lengths for the four methods described here. As can be seen in this table, method 2 produces no longer the very short segments of method 1.

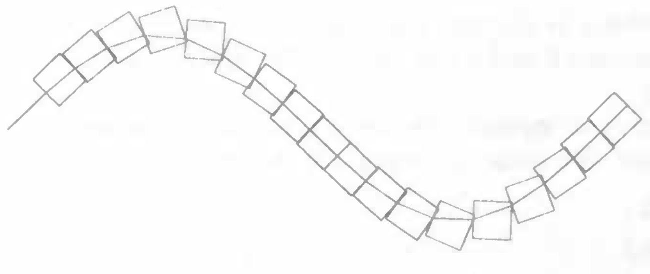


Figure 3.4 A sine-curve approximated by method 2.

3.2.3 Method 3

In both method 1 and 2, the fixed size of the squares implies an upper bound on the length of the segments. So it seems natural now to vary not only the orientation, but also the size of the squares, thus changing them into rectangles of fixed height *delta* and variable length. The rectangle can be stretched as long as the curve lies completely within it, see figure 3.5.

Let $l(t)$ be the line through A and B, such that $l(0)$ = A and $l(1)$ = B. Then let $l_1(t)$ be $l(t)$ shifted orthogonally over a distance of *+delta*, and $l_2(t)$ be $l(t)$ shifted over *–delta*. Now the rectangle, delimited by l_1 and l_2, can be stretched to the point where the curve intersects l_1 or l_2, whichever is closest to the vertical line through B. As before, we will approximate the part of the curve lying inside the rectangle with a straight line BC. This is method 3.

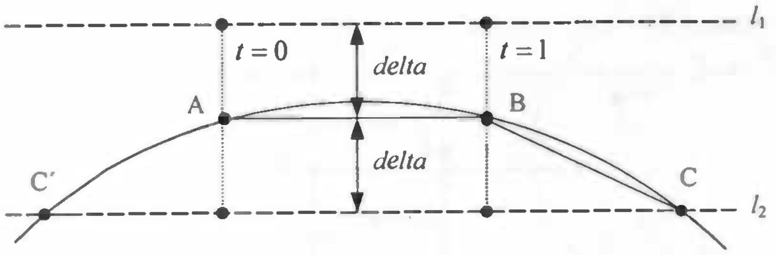


Figure 3.5 Method 3

But how to find the intersection point C? Let's consider only l_1 for the moment. We need to find the smallest zero-point z_1 of $f_1(t) := f(l_1(t))$, with $t>1$. The bisection method does not apply here, since we don't have two points with opposite signs. I have chosen here to use Newton's method [appendix A]. Newton's method requires an initial 'guess' t, for which I have chosen $t = 2$. In general, it converges to a zero z_1 of f_1. In the same way we find a zero z_2 on l_2.

If $1 < z_1 < z_2$ or $z_2 < 1 < z_1$, then let $C = l_1(z_1)$. If $1 < z_2 < z_1$ or $z_1 < 1 < z_2$, then let $C = l_2(z_2)$. If both $z_1 < 1$ and $z_2 < 1$ then we panic.

Here we encounter a major disadvantage of this method: there is no control over the point to which Newton's method converges. It may for instance converge to C′ instead of C. But in practice, using small enough delta, this hardly ever happens.

Another problem is the slow convergence, as can be seen in table 3.3. This is because the functions f_1 and f_2 will be very flat, since l_1 and l_2 lie nearly parallel to the actual curve.

The importance of method 3 is that it produces longer segments in regions of low curvature and vice versa. An example is shown below.

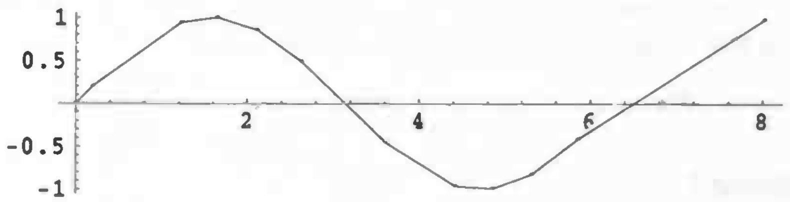


Figure 3.6 A sine-curve approximated by method 3; delta = 0.2. The real curve is printed in grey.

3.2.4 Method 4

Newton's method would converge much faster if we restrict f to a line orthogonal to the curve, since f would be less flat there. This consideration led me to method 4 (see figure 3.7). If the signs of f in D and E differ, move DE to the right and stretch the rectangle. If they don't differ, move DE to the left, shrinking the rectangle. Repeat this until a) stretching would result in equal signs in DE, or b) shrinking results in opposite signs. Next, Newton's method is applied to the line DE, which is expected to be more or less orthogonal to the curve. The zero-point found is connected to B to form the new segment.

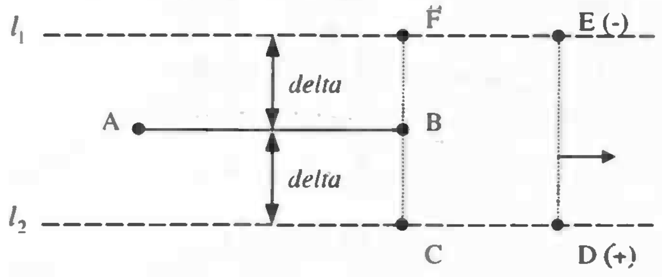


Figure 3.7 Method 4. Because signs in D and E differ, DE can be moved to the right.

As an initial guess, the length of the rectangle CD is taken to be equal to the length of the previous segment AB. Mostly, especially for smaller *delta*, this will be a good guess. For the shrinking and the stretching I chose dividing into halves respectively doubling the length of the rectangle. Other choices are possible here, like increasing or decreasing with a fixed step; or multiplying with a factor other than 2. An example is shown in figure 3.8.

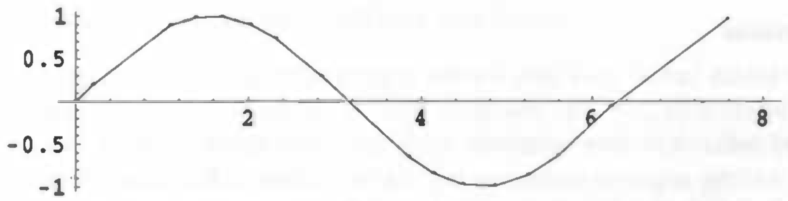


Figure 3.8 Approximation of a sine-curve using method 4; delta = 0.2.

Method 4 has the following advantages over method 3 :

- rapid convergence of Newton's method (table 3.3);
- when Newton's method fails (it should always find a zero-point between D and E), the bisection method can still be used. In practice, this was never necessary;
- it is more robust.

On the other hand, method 4 produces more/shorter segments than method 3. But since we use a factor 2 to shrink or enlarge the rectangle, the segment produced by method 4 cannot be shorter than half the length of the segment that would have been produced by method 3. Thus method 4 will produce no more than twice as many segments (table 3.1).

Figure 3.9 gives another example of a curve approximated using method 4.

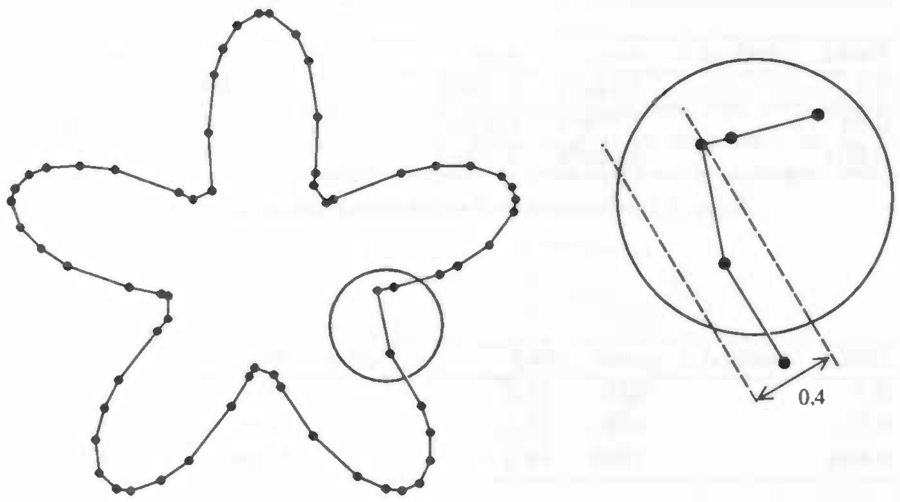


Figure 3.9 An approximation, using method 4, of a 'flower'-function (φ,2+sin(5φ)) in polar co-ordinates; delta = 0.2. The encircled area seems to contain a flaw; but actually the deviation here is just below 0.2.

3.3 Results

The tables below give data for the approximation of $y=sin(x)$ ($f(x,y) = y-sin(x)$) on the interval $[0,2\pi]$. The precision used in calculating the zero-points is $\varepsilon = 10^{-8}$ (see appendix A). This relatively high precision (compared to *delta*) is used to ensure that the segment endpoints will be very close to the actual curve.

In method 1 and 2, *delta* is the size of the squares. In method 3 and 4, *delta* is the height of the rectangles.

delta	method 1	2	3	4
0.1	100	76	15	18
0.01	1026	764	45	57
0.001	10281	7640	141	191

Table 3.1 Number of segments

Delta	method 1	*min*	*Max*	method 2	*min*	*max*
0.1		2.36e-4	1.41e-1		1.00e-1	1.01e-1
0.01		2.29e-7	1.41e-2		1.00e-2	1.00e-2
0.001		1.08e-8	1.41e-3		1.00e-3	1.00e-3

Delta	method 3	*min*	*max*	method 4	*min*	*max*
0.1		3.18e-1	1.13e0		1.80e-1	1.57e0
0.01		1.00e-1	6.87e-1		5.74e-2	9.98e-1
0.001		3.16e-2	3.40e-1		2.27e-2	3.73e-1

Table 3.2 Minimum and maximum segment lengths

Delta	method 3	*total*	*avg*	method 4	*total*	*avg*	*Newton*
0.1		236	15.7		187	10.4	5.6
0.01		690	15.3		549	9.6	5.0
0.001		2008	14.2		1729	9.1	4.8

Table 3.3 Total number of function evaluations and average number of evaluations per segment. Since in method 4 not all f evaluations can be attributed to Newton's method, the Newton column gives the average number of evaluations made when applying Newton's method.

4. Triangulation of smooth implicit surfaces

4.1 Considerations

Methods 3 and 4 in the previous chapter depend crucially upon the fact that the curve is smooth. Because of this smoothness:

- previously calculated roots can be used to find new starting points that are guaranteed to be close to the actual curve;
- Newton's method converges rapidly because it can be applied on a line nearly orthogonal to the curve.

In the following algorithm, these benefits will be exploited.

4.2 Outline of the algorithm

Suppose we have already triangulated a region of the surface, then we can use the vertices on the boundary of this region, which are zeros of f, to compute new starting points for Newton's method, with which we can in turn expand the triangulated region. So the algorithm will have to:

- keep track of the boundaries of triangulated regions, which we shall call 'fronts';
- somehow expand these fronts, thereby calculating new triangles;
- take care of colliding fronts (they may collide with themselves, or with each other).

4.2.1 Fronts

A front is a cyclic list of edges, ordered in clockwise direction with respect to the surface normals in its vertices. Its inner region is supposed to be fully triangulated, i.e. each triangle edge is either a front edge *or* it is shared between two adjacent triangles. See also figure 4.1.

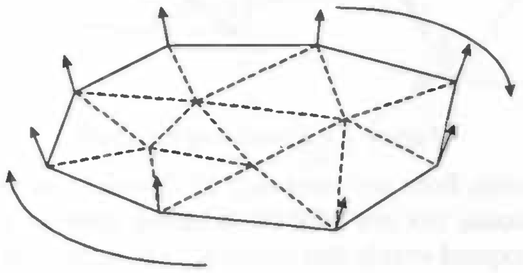


Figure 4.1 Example of a front. Solid edges are front edges; the arrows signify normal vector directions. Front edges are ordered clockwise.

4.2.2 Expanding fronts

A front delimits a region of the surface that is already triangulated. The most important step of the algorithm is to expand the triangulated region, producing additional triangles, and to update the front accordingly. This is called expanding the front. Note that expanding the front will not necessarily increase front length.

For instance, when triangulating a sphere, the front length will increase at first, reach a maximum length (roughly the diameter of the sphere) and then decrease again until the surface is closed (see figure 4.6, page 15).

A front is expanded by repeatedly expanding one of its vertices, see figure 4.2. Expanding consists of the following steps:

- calculate the surface normal N in point p; let V be the tangent plane of the surface in p;
- let v_1 be the previous vertex in the front list, and v_2 be the next. Project the two edges pv_1 and pv_2 onto plane V yielding e_1 and e_2;
- calculate the outer angle θ between e_1 and e_2;
- let $n = \text{round}(\theta/60°)$;
- if $n \leq 1$, simply remove p from the front list, and report the resulting triangle pv_1v_2;
- if not, let $\alpha = \theta/n$; calculate n-1 new points q_i in V (i = 1,..,n-1), at a fixed distance *delta* of p, such that the angle between e_1 and pq_i in plane V is $i\alpha$;
- apply Newton's method to the lines l_i, through q_i parallel to N, resulting in new surface points p_i;
- report the n+1 new triangles $p_i pp_{i+1}$ (i = 0..n), where $p_0 = v_1$ and $p_{n+1} = v_2$;
- update the front to include p_i.

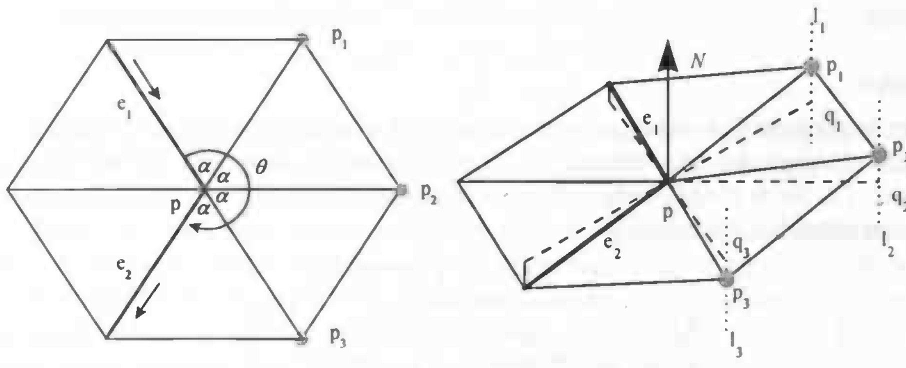


Figure 4.2 Expanding the front.

Although any vertex from any front may be chosen to be expanded, the present implementation chooses the one with the smallest external angle θ. This choice causes the front to expand evenly and minimises the number of collisions (see next section).

4.2.3 Colliding fronts

When a vertex v that is to be expanded is too close to another front, the above scheme will produce overlapping triangles. In this case a 'bridge' edge is introduced, connecting v to the nearest vertex v' of the opposing front, and both fronts are rearranged as in figure 4.3.

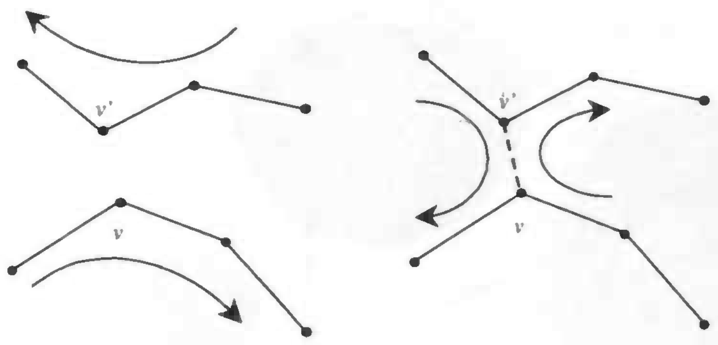


Figure 4.3 Meeting fronts. Left: before introduction of bridge edge. Right: after introduction of bridge edge v-v'. The arrows indicate the clockwise direction of the fronts.

A front may very well collide with itself. In that case, it is effectively split in two. Conversely, if two different fronts meet, they will be merged into one. Both cases can be observed in figure 4.7, page 16.

4.2.4 Getting started

The algorithm will need at least one initial front to start with. Given a point *p* on the surface, an initial 'honeycomb' front may be created as in figure 4.4.

Alternatively, one may want to triangulate only part of the surface. In those cases, one or more pre-calculated fronts may serve as input to the algorithm. Intersecting the surface with a plane yields a 2-dimensional curve. Using the methods described in chapter 3, this curve can be approximated to form an input front. The truncated sphere in figure 4.5 was created this way.

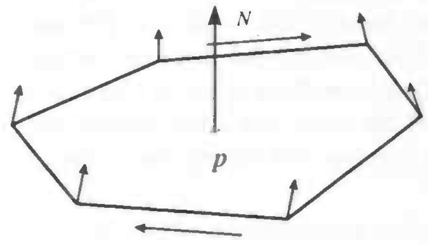


Figure 4.4 Initial front around p.

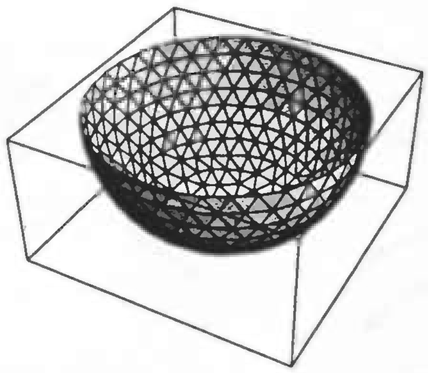


Figure 4.5 The unit sphere truncated by the plane z=0.

When a front has eventually reached the shape of a triangle, this triangle is reported and the front is deleted. When no fronts remain, the surface is closed and the algorithm terminates.

4.3 Results

The figures below illustrate the triangulation process for two surfaces:

- the unit sphere, $f(x,y,z) = x^2 + y^2 + z^2 - 1$, and
- a torus with outer radius R=0.7 and inner radius r=0.3, $f(x,y,z) = (x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2)$.

Figure 4.6 shows how the front proceeds 'evenly', keeping roughly a circular shape. This is because the front vertex to be expanded next is always chosen to be the one with the least external angle.

Figure 4.7 illustrates the front collision process. Expanding from a single source vertex, the front quickly folds around the torus (a). When it collides with itself, a bridge edge is introduced, and the front splits in two separate fronts (b). One front proceeds clockwise around the torus, the other counter-clockwise (c). Finally they meet again, merging back into one and closing the surface (d).

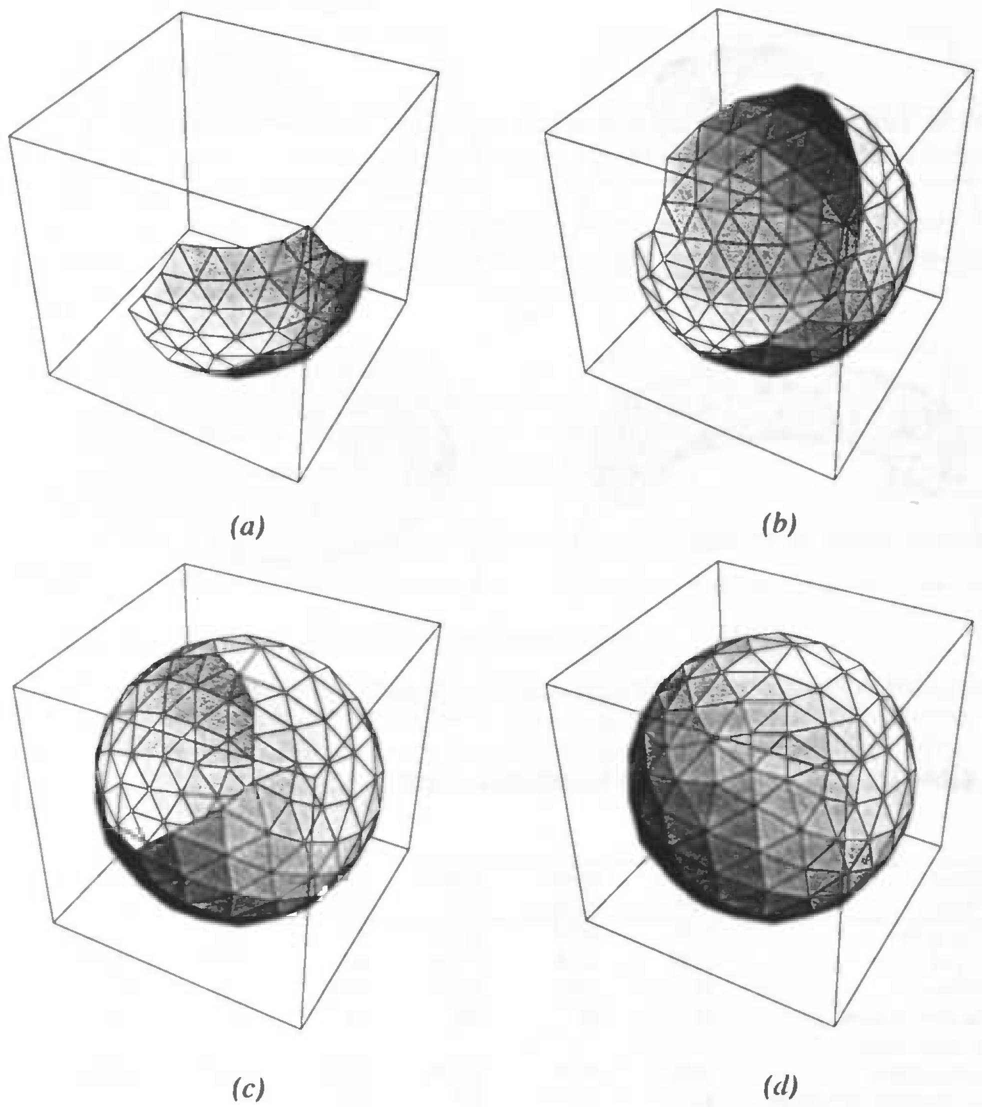


Figure 4.6 Four stages in the triangulation of the unit sphere.

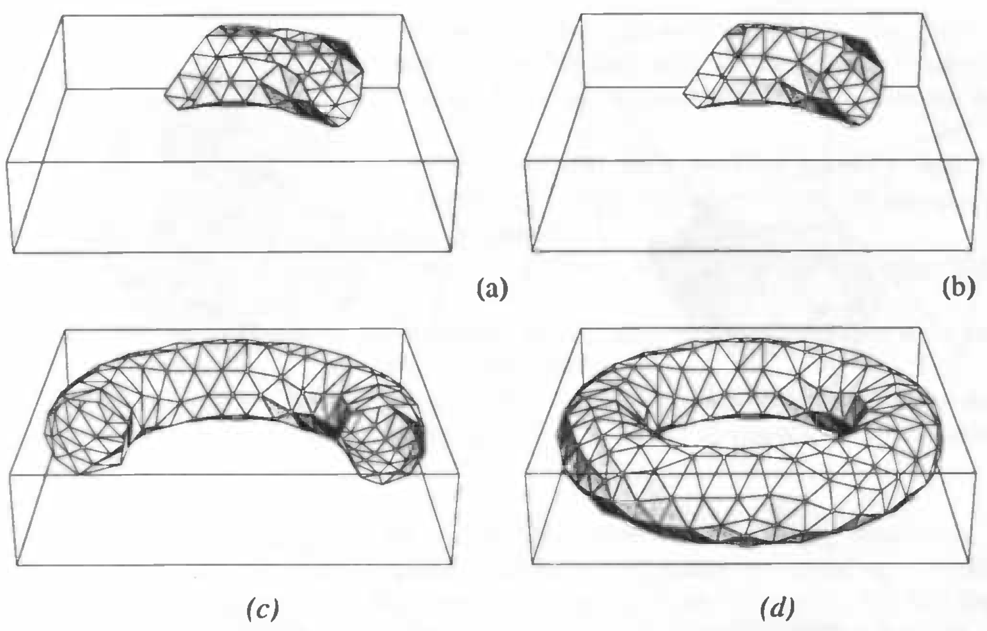


figure 4.7 Four stages in the triangulation of a torus.

Table 4.1 gives numerical data for different runs of the algorithm.

Surface *Delta*	sphere 0.20	Sphere 0.10	sphere 0.05	torus 0.20	torus 0.10	torus 0.05
Triangles	768	3034	11830	628	2650	10448
Edges	1152	4551	17745	942	3975	15672
Vertices	385	1518	5916	314	1325	5224
maximum number of front edges	41	81	158	29	67	113
f evaluations	1540	4554	17748	1197	4355	15533
f evaluations per vertex	4.0	3.0	3.0	3.8	3.3	3.0
∇f evaluations	1926	6073	23665	1515	5688	20763
∇f evaluations per vertex	5.0	4.0	4.0	4.8	4.3	4.0
$max\,\vert f(vertex)\vert$	2.37e-16	2.57e-16	2.81e-16	2.04e-15	2.46^{e}-15	2.47^{e}-15
minimal triangle angle	29.1	23.8	31.7	31.6	16.5	30.7
maximal triangle angle	101.0	122.0	51.5	99.4	139.9	105.9
average minimal angle	51.2	51.4	93.9	50.7	50.7	50.6
average maximal angle	70.6	70.6	70.1	70.0	70.2	70.1
total area	12.461	12.540	12.559	11.630	11.796	11.831
average triangle area	0.0162	0.0041	0.0011	0.0185	0.0045	0.0011
minimal triangle area	0.0084	0.0025	0.0006	0.0110	0.0019	0.0006
maximal triangle area	0.0338	0.0097	0.0020	0.0438	0.0099	0.0022

Table 4.1

4.4 Implementation details

4.4.1 Data structures

It is not necessary to store calculated triangles; it suffices to keep track of the fronts. In the present implementation, triangles are identified by their three corners and are written to an output to file.

Fronts are stored as cyclic lists of vertices. All front vertices are also stored in one linear list: the *overall* front vertex list. For each front vertex, the following information is stored:

- its location (x,y,z);
- the normal vector in (x,y,z);
- pointers to the previous and next vertex in the front;
- pointers to the previous and next vertex in the overall front vertex list (when this list is empty, the algorithm terminates);
- its exterior angle θ.

Each time a front vertex is expanded, the exterior angles of its two neighbors change and must be recalculated.

4.4.2 Collision detection and bridge construction

When vertex v is to be expanded, the overall front vertex list is searched to determine if there exists a front vertex w, not being a direct neighbor of v, having a distance v-w that is smaller than v's distance to any of its neighbors. If it exists, a collision is detected. A bridge edge is then introduced between v and the nearest 'intruding' front vertex w.

5. Conclusions

The algorithms, as presented in chapter 3 and 4, make it possible to triangulate a great range of implicit surfaces. By varying the 'step size' *delta*, the faithfulness of the approximation can be controlled. Parts of surfaces can be triangulated by specifying boundary fronts.

The triangulation algorithm is fast and uses little working memory because only the actual fronts need to be stored in memory. Surface vertices are calculated fast and accurately by using Newton's method.

The data structure for fronts is simple, which makes the algorithm straightforward to implement.

The algorithm produces good-quality triangles, in terms of triangle area and min/max triangle angles: no small or skinny triangles.

Further research could investigate the possibility of making the algorithm adaptive to surface curvature, producing larger triangles in areas of low curvature and vice versa.

Note. Recently, a paper by Erich Hartmann [5] appeared, describing a "marching method for the triangulation of surfaces", which is indeed *very* similar to the algorithm presented in this paper. As much as I am pleased by the fact that our common method is worthy of publication, I am disappointed by the fact that he "beat me to it".

6. References

[1] L.P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 274-280, 1993.

[2] J. Bloomenthal. Polygonization of implicit surfaces. In *CAGD* 5 (1988), pages 341-355.

[3] L.P. Chew. Constrained Delaunay triangulations. In *Algorithmica* 4 (1989), pages 97-108.

[4] K.E. Atkinson. An introduction to numerical analysis. *Wiley*, 1989

[5] E. Hartmann. A marching method for the triangulation of surfaces. In *The Visual Computer (1998) 14*, pages 95-108

[6] T.M. Apostol. Calculus, volume II, second edition. *Wiley,* 1967

Appendix A. Rootfinding methods

Two rootfinding methods are mentioned in this paper. They will be described here shortly. A more elaborate description can be found in [4].

Both methods are used for finding the zero-point of a single-valued continuous function *f*. In this paper they are used to find zeros of 2-valued and 3-valued functions *F*. In that case they are applied on a line $l(t)$, such that $f(t) := F(l(t))$ is again a single-valued function. When $f'(t)$ is needed, it can be computed from ∇F.

A.1 The bisection method

If $f(x)$ is continuous on an interval $[a,b]$ and it also satisfies $f(a)f(b) < 0$, then *f* must have at least one root in $[a,b]$. Let $c = (a+b)/2$. If $f(b)f(c) \leq 0$ then $a := c$ otherwise $b := c$. Now the interval $[a,b]$ is divided in halves, and still $f(a)f(b) < 0$. Repeating this process, $[a,b]$ converges arbitrarily close to a root of *f*. The iteration is terminated when $|a-b| < \varepsilon$.

The bisection method is a very simple and robust method. It is said to converge *linearly* with a rate of ½, which means that the average error in each iteration step decreases with a factor ½. Compared to the following method, this is a rather slow convergence.

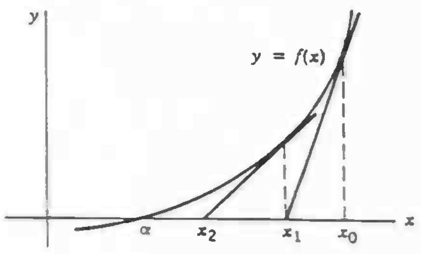


figure A.1 Newton's method

A.2 Newton's method

Assume that an initial estimate x_0 is known for the desired solution α of $f(x) = 0$. Since x_0 is supposed to be close to α, we can approximate *f* by the tangent line in $(x_0, f(x_0))$ to the graph of *f*. We then take the intersection of this tangent line and the *x*-axis as a new (and hopefully better) approximation of α.

This leads to the following iteration formula (see also figure A.1):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The iteration stops when $|x_n - x_{n-1}| < \varepsilon$.

Newton's method converges *quadratically*, i.e. $|\alpha - x_{n+1}| \leq c|\alpha - x_n|^2$ for some $c > 0$. Nevertheless, it may not converge at all. If f has more than one root, it may also converge to the 'wrong' root. In the examples in this paper, this happens very rarely, because the initial 'guess' is always sufficiently close to α.