Adaptive Polygonization Made Simple

Luiz Velho

IMPA – Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina, 110, Rio de Janeiro, Brazil, 22460

Abstract. This paper describes a simple algorithm for the adaptive polygonization of implicit surfaces. It gives a practical way to construct optimal piecewise linear representations. The method starts with a coarse uniform polygonal approximation of the surface and subdivides each polygon recursively according to local curvature. In that way, the inherent complexity of the problem is tamed by reducing part of the full three dimensional search to two dimensions.

1 Introduction

Implicit models constitute a powerful mathematical description of the geometry of three dimensional objects [10]. Under this framework, a surface is defined as the set of points which satisfy the equation $f(x, y, z) = 0$. Simple primitive implicit shapes can be specified by algebraic functions, such as quadrics, [5]. More complex implicit shapes can be specified by combining primitives using point set or blend operations that are the basis of, respectively, CSG, [13], and Blobby models, [4].

The implicit description is particularly effective to model smooth intricate shapes. For this reason, it has been extensively used in various areas of application. For example: in science for the visualization of molecules [12]; in engineering for the design of industrial products, [14]; and in entertainment for the creation of animated characters, [3].

In order to do computations with implicitly defined objects a graphics system must process the implicit representation either in symbolic or numeric form. The system must incorporate computational methods implementing various classes of operations with these objects. Typical examples are visualization operations, such as ray tracing, [15], and analysis operations, such as interference detection [7]. Although there are algorithms that perform such operations directly on the implicit form, sometimes, a conversion to another representation is necessary.

An important case of conversion between geometric descriptions is the polygonization of implicit surfaces. This operation transforms from implicit to parametric form. It produces a polygonal mesh which gives a piecewise linear approximation of the surface. Polyhedral surfaces are used in many graphical applications because of their simplicity. In particular, most interactive workstations have polygon rendering engines that enable them to display these surfaces in real time.

2 Background

Polygonization methods incorporate two basic operations: sampling and structuring [8]. *Sampling* generates a set of points on the implicit surface. *Structuring* links those points to construct a mesh. The first operation deals with geometry, while the second operation deals with topology. Algorithms can be classified according to how they implement these two operations.

The simplest polygonization algorithms are based on uniform decompositions of the ambient space. They employ a cell complex made of, either cubical, or tetrahedral elements of the same size. (See respectively [18] and [1]). The implicit function is evaluated at node points of the grid underlying the uniform spatial decomposition, and the samples obtained are assembled using adjacency relations from the cell complex. Algorithms of this type are straightforward to implement, but produce polygonal meshes that are not adapted to the implicit surface. Such a solution is only acceptable for shapes with regular features, where the surface curvature is almost constant.

In general, uniform decomposition results in polygonizations that give poor approximations of the implicit surface. The fixed sampling rate causes oversampling of areas with low curvature and undersampling of areas with high curvature. For a given level of accuracy, the algorithm has to sample everywhere with a rate that is high enough to capture the smallest shape feature. This often creates a representation which has too many polygons, and that may be impractical because of memory limitations.

The best solution is to construct an adapted polygonization. This method employs a sampling rate that varies spatially according to local surface complexity. Consequently, it produces the minimum number of polygons required to approximate the surface with the desired precision.

3 Adaptive Polygonization

Adaptive polygonizaton algorithms are more complex, because they must solve two interdependent problems:

- ensure optimal sampling;
- enforce correct topology.

Optimal sampling guarantees a faithful geometric approximation, and depends on the adaptation criteria. Correct topology guarantees the consistency of the polygonal mesh and depends on the structuring mechanism.

The main difficulty is that when changes are made *locally* to the sampling rate, the mesh topology may be affected *globally*. Therefore, it is necessary to devise a mechanism that synchronizes the solution of these two problems. If that is not done correctly, the polygonization may exhibit holes caused by wrong connectivity. For example, different levels of subdivision in two adjacent cells could create a crack along the boundary between them.

Previous adaptive polygonization algorithms are based on the refinement of some three dimensional cell complex. They recursively subdivide space until the adaptation criteria is met and, at that stage, polygons are generated for cells intersecting the implicit surface. Synchronization is achieved either through a hierarchical spatial data structure or exploiting edge coherence. The first approach uses a restricted tree. It maintains the subdivision structure balanced with repeated refinement steps that modifies only one cell at each pass. Whenever a cell is divided its neighbors are constrained to be at levels immediately above or below [6], [11]. The second approach subdivide cells independently constraining polygon edges according to the adaptation criteria. In that way, it is able to make consistent decisions when it splits neighbor cells sharing an edge [17].

All the algorithms described above employ a full 3D adaptive partition of space, and perform sampling a structuring in one single step. Strictly speaking, this three dimensional search is necessary for computing a correct solution in the case of surfaces with arbitrary topology. Unfortunately, such a requirement places an enormous burden in actual implementations, both in terms of algorithm complexity and performance. This is mainly due to the 3D combinatorics of the problem, as well as, the need of space and time resources.

The situation above has motivated the quest for alternative adaptive polygonization methods that avoid the problem intractability at the expense of computing sub-optimal solutions. One attempt is this direction consists in first generating a fine uniform polygonization that is subsequently simplified by merging clusters of co-planar polygons, [16]. This two-step procedure reduces the algorithm complexity, but demands large memory resources and computing time.

4 A Simpler Adaptation Method

In this paper we present an adaptive polygonization method that overcomes the difficulties described in the previous section. The algorithm is easy to implement and it is very efficient both in space and time. For this reason, it provides a practical solution for real-world applications.

The method consists of two steps:

1. initial polygonization;
2. adaptive refinement.

The initial polygonization is created from an uniform space decomposition. The output of this process is a triangle mesh that serves as the basis for adaptive refinement. During adaptation, the elements of the mesh are recursively subdivided according to local curvature of the implicit surface.

The curvature is estimated over each element of the mesh using the deviation of the surface normal from the normal of the element's support plane.

The key to simple adaptation is the separation between structuring and sampling. Structuring is done in the first step, where a coarse sampling is also performed. In the second step, new samples are generated and projected onto the surface using the gradient of the implicit function. The structure of the initial mesh guides the adaptive sampling and, at the same time, provides the connectivity information for maintaining geometrical consistency.

The link between structuring and sampling is based on edge coherence. Whenever the local surface curvature exceeds some predefined threshold a triangle is subdivided by splitting its edges at their midpoints. But a midpoint is projected onto the surface only when the curvature *along* the edge is greater than the threshold. This guarantees that a consistent decision is made for two polygons sharing an edge.

Note that the first step takes place in a three dimensional space (i.e. in the region delimited by the bounding box of the implicit object), while the second step is restricted to a two dimensional space (i.e. the polygonal mesh approximating the implicit surface). This reduction in the dimensionality of the problem makes the computation very efficient.

The performance gains are even more significant because the first step, which is more costly, needs to be executed only once – at a fixed, coarse resolution – while the second step, which is a fast operation, needs to be executed repeatedly until the desired accuracy of the solution is achieved.

In summary, structuring / sampling separation, associated with dimensionality reduction, are the main factors for the simplicity and efficiency of the method.

This new hybrid 3D/2D adaptation method has only one limitation compared with methods that employ full a 3D scheme. If the initial sampling is too coarse in relation to the level of detail of the implicit shape, the polygonization may not capture the correct topology of the surface. This is because structuring is done in 3D during the first pass. At that stage, the method commits to a topology and proceeds on to the second pass, which works in 2D and changes just the geometry. However, this is not a serious problem because there is usually enough information about the surface to determine the correct sampling rate. In practice, a very coarse sampling grid suffices for most shapes of interest.

5 Computing the Initial Mesh

In this section we describe the first step of the algorithm – the computation of the initial mesh. This process corresponds to an uniform polygonization of the implicit surface which is a well understood problem and has various alternative solutions available in the graphics literature. We should remark that since any uniform polygonization method could be used to produce the initial mesh, the description of such a method could be omitted from the paper. Nonetheless, we decided, for completeness sake, to include a polygonization algorithm based on a simplicial space decomposition. This is perhaps the most elegant solution for the problem and it is a perfect match for our mesh adaptation method.

The algorithm will be described using a "C-like" pseudo-code notation that is very close to the actual source code of the program.

We define the data structure `Vector`, $\mathbf{w} = (x, y, z)$, that is an element of the three dimensional Euclidean space,

```
typedef struct Vector
  Real x, y, z;
```

and the operators: inner product (also known as dot product); vector addition; and product of a vector by a scalar.

`v_dotprod` : $\mathbf{w} = \mathbf{a} \cdot \mathbf{b} = < \mathbf{a}, \mathbf{b} >$
`v_add` : $\mathbf{w} = \mathbf{a} + \mathbf{b}$
`v_scale` : $\mathbf{w} = s * \mathbf{a}$

The implementation of vector operations is straightforward from their definitions. Therefore, we will not show the pseudo-code.

We also define the data structure `Vertex`, which is the basic unity of information in the program. It is used to represent a node of the 3D cell complex associated with the uniform space decomposition, as well as, a node of the triangle mesh approximating the implicit surface.

```
typedef struct Vertex
  Real d;
  Vector p, n;
```

This structure contains all knowledge about the implicit object that is required at the node: `p`, the position of the node; `d`, the density value of the implicit function f at `p`; and `n`, the normal direction at `p`.

The normal vector to the level surfaces of f is given by the gradient field $\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)$.

The interface with the implicit model is made through the functions: `f_value`, that returns the value of f, and `f_grad`, that returns the gradient of f.

The uniform polygonization algorithm decomposes the bounding box of the implicit shape using a simplicial cell complex. It then identifies the set of cells that are intersected by the implicit surface and for each of these cells, it generates an element of the polygonal mesh approximating the surface.

Our simplicial decomposition uses the classical Coxeter-Freudenthal space subdivision scheme, that is defined as follows. It starts with a rectangular space partition of cubical cells. For a cube in $\mathbb{R}^3$ with vertices $p_0, \ldots, p_7$, it takes the diagonal $p_0 p_7$ and projects it onto each face of the cube. This gives a triangulation of the faces of the cube. The 3D simplicial cells are constructed by adding to each triangle in a face, the vertex of the diagonal $p_0 p_7$ that does not belong to it. We obtain in this way 6 simplices of dimension 3.

$$
\begin{aligned}
\rho_0 &= (p_0, p_1, p_3, p_7) \\
\rho_1 &= (p_0, p_1, p_5, p_7) \\
\rho_2 &= (p_0, p_2, p_3, p_7) \\
\rho_3 &= (p_0, p_2, p_6, p_7) \\
\rho_4 &= (p_0, p_4, p_5, p_7) \\
\rho_5 &= (p_0, p_4, p_6, p_7)
\end{aligned}
$$

The above decomposition scheme is illustrated in Figure 1.

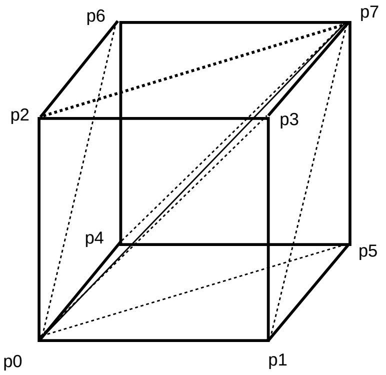


Figure 1: Coxeter-Freudenthal decomposition of the cube in $\mathbb{R}^3$

The implicit surface defined by $f(x, y, z) = 0$ intersects a cell only if the value of implicit function f changes from positive to negative within the cell. When the resolution of the space decomposition is adequate, this fact can be determined from the values of the f at the vertices of the cell, as indicated in Figure 2 for the 2D case.

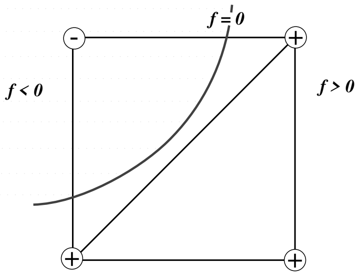


Figure 2: Intersection of a 2D cell with an implicit curve

The procedure `bbox_scan`, shown below, is the top layer of the program. It performs a rectangular subdivision of the implicit object's bounding box and scans all cubical cells testing the sign of the implicit function in order to identify possible intersections with the implicit surface. When a potential hit occurs, the Coxeter-Freudenthal decomposition of the cube is generated and each of its simplices are processed.

The current cube of the subdivision is represented by the array, `Vertex v[8]`, that stores the vertices of the cube.

```
bbox_scan(ll, ur, inc)
{
  for (x=ll.x; x<ur.x; x+=inc)
    for (y=ll.y; y<ur.y; y+=inc)
      for (z=ll.z; z<ur.z; z+=inc) {
        for (hit=FALSE, k=0; k<8; k++) {
          v[k].p.x = (k&01) ? x : x+inc;
          v[k].p.y = (k&02) ? y : y+inc;
          v[k].p.z = (k&04) ? z : z+inc;
          v[k].d = f_value(v[k].p);
          v[k].n = f_grad(v[k].p);
          if (k == 0)
            side = sign(v[k].d);
          else if (side != sign(v[k].d))
            hit = TRUE;
        }
        if (hit == TRUE) {
          simplex(v[0],v[1],v[3],v[7]);
          simplex(v[0],v[5],v[1],v[7]);
          simplex(v[0],v[3],v[2],v[7]);
          simplex(v[0],v[2],v[6],v[7]);
          simplex(v[0],v[4],v[5],v[7]);
          simplex(v[0],v[6],v[4],v[7]);
        }
      }
}
```

The procedure `simplex` determines, based on the sign of f, if there is an actual intersection with the cell, and if so, how it is pierced by the implicit surface. Note that, when the parent cube is intersected by the surface only some cells of the simplicial decomposition will be crossed by it.

The implicit surface may intersect the edges of a simplex in either three or four points. This results in two basic configurations: in the first case, a triangle is generated, and in the second case a quadrilateral is generated (see Figure 3).

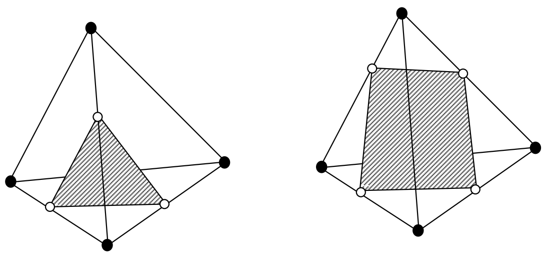


Figure 3: Intersection of the surface with a 3-simplex

```
simplex(v0, v1, v2, v3)
{
  if (v0.d < 0)
    if (v1.d < 0)
      if (v2.d < 0)
        if (v3.d < 0) ; /* no int. */
        else          tri(v3,v2,v1,v0);
      else
        if (v3.d < 0) tri(v2,v0,v1,v3);
        else          quad(v2,v3,v0,v1);
    else
      if (v2.d < 0)
        if (v3.d < 0) tri(v1,v3,v2,v0);
        else          quad(v1,v3,v2,v0);
      else
        if (v3.d < 0) quad(v1,v2,v0,v3);
        else          tri(v0,v3,v2,v1);
  else
    if (v1.d < 0)
      if (v2.d < 0)
        if (v3.d < 0) tri(v0,v1,v2,v3);
        else          quad(v0,v3,v1,v2);
      else
        if (v3.d < 0) quad(v0,v2,v3,v1);
        else          tri(v1,v3,v0,v2);
    else
      if (v2.d < 0)
        if (v3.d < 0) quad(v0,v1,v2,v3);
        else          tri(v2,v3,v1,v0);
      else
        if (v3.d < 0) tri(v3,v0,v1,v2);
        else          ; /* no int. */
}
```

The procedures tri and quad compute the intersection points of the surface with simplex edges, and produce the elements of the triangular mesh that is used as the starting point for the adaptation process. They call tri_adapt to iniciate the second phase of the program.

```
tri(v0 ,v1, v2, v3)
{
  i0 = intersect(v0, v1);
  i1 = intersect(v0, v2);
  i2 = intersect(v0, v3);
  tri_adapt(i0, i1, i2);
}
```

The procedure quad splits the quadrilateral in two triangles.

```
quad(v0, v1, v2, v3)
{
  i0 = intersect(v0, v2);
  i1 = intersect(v0, v3);
  i2 = intersect(v1, v3);
  i3 = intersect(v1, v2);
  tri_adapt(i0, i1, i2);
  tri_adapt(i0, i2, i3);
}
```

The intersection of an edge $v_0 v_1$ with the implicit surface is found using a continuation method. The procedure intersect computes the initial guess, p, from a linear approximation of the implicit function. The point p is projected onto the surface by project_s that will be described in the next section.

```
Vertex intersect(v0, v1)
{
  t = v0.d/(v0.d - v1.d);
  p=v_add(v_scale(v0.p,1-t),v_scale(v1.p,t));
  v.p = project_s(1, p, f_value(p));
}
```

6 Adaptive Mesh Refinement Algorithm

The mesh refinement algorithm recursively subdivides triangular cells according to the adaptation criteria. It processes each triangle independently, enforcing geometric consistency through edge coherence.

A triangle, with vertices v_0, v_1, v_2, is subdivided into four triangles by splitting its edges, $e_1 = v_0 v_1$, $e_2 = v_1 v_2$, and $e_3 = v_2 v_0$, at their midpoints — m_1, m_2 and m_3 — and connecting the midpoints of adjacent edges. A diagram illustrating this scheme is depicted in Figure 4.

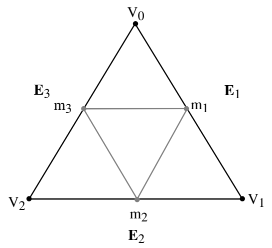


Figure 4: Subdivision scheme of a triangle

The procedure tri_adapt implements the subdivision scheme described above. The decision to subdivide the triangle is based on a classification of the surface curvature along its edges. If all edges are flat, then a piecewise linear approximation is good enough and the program outputs that triangle. If one or more edges are not flat, then the surface is locally curved and that triangle must be subdivided. In this case, the edge midpoints are computed and the procedure is called recursively for each piece.

```
tri_adapt(v0, v1, v2)
{
  e1 = edge_code(v0, v1);
  e2 = edge_code(v1, v2);
  e3 = edge_code(v2, v0);
  if (e1==FLAT && e2==FLAT && e3==FLAT){
    tri_output(v0, v1, v2);
  } else {
    m1 = midpoint(e1, v0, v1);
    m2 = midpoint(e2, v1, v2);
    m3 = midpoint(e3, v2, v0);
    tri_adapt(v0, m1, m3);
    tri_adapt(v1, m2, m1);
    tri_adapt(v2, m3, m2);
    tri_adapt(m1, m2, m3);
  }
}
```

The surface curvature along an edge is measured by the angle α between the surface normals, n_0 and n_1, at the edge endpoints. (See Figure 5.)

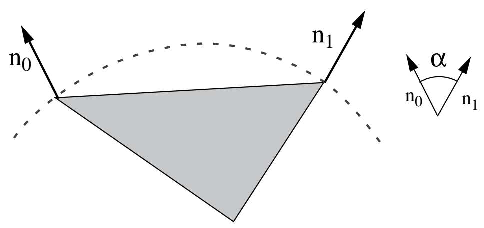


Figure 5: Adaptation Criteria

The procedure `edge_code` estimates the edge curvature from the cosine of α (using the fact that $\cos\alpha = n_0 \cdot n_1$). If the angle is greater than a predefined constant, the edge is classified as *not flat*.

The desired level of accuracy of the polygonal approximation, indicated by the global constant `dot_tol`, can be set by the user when the program gets executed.

```
Int edge_code(v0, v1)
{
  return (v_dotprod(v0.n, v1.n) < dot_tol) ?
          FLAT : NOT_FLAT;
}
```

The procedure `midpoint` returns the midpoint of an edge. In spite of its simplicity, it implements the most important part of the adaptation process — the edge coherence mechanism. Notice that after computing the true midpoint `m` of the linear segment v_0v_1, it projects `m` onto the surface *only* if the edge is not flat. This guarantees that when the edge is flat a hole in the polygonization will not occur, even if one of the triangles sharing the edge is subdivided and the other not.

```
Vertex midpoint(e, v0, v1)
{
  m.p = v_scale(0.5, v_add(v0.p, v1.p));
  if (e == NOT_FLAT)
    m.p = project_s(1, m.p, f_value(m.p));
  return m;
}
```

We employ a physics based method to project a point onto the surface. Under this paradigm, we interpret the modulus of f as a potential function and the gradient of $|f|$ is used to generate a force field that drives particles to the implicit surface [9]. This gives the following equation of motion for a unit mass particle

$$\frac{dx}{dt} + \mathrm{sign}(f)\nabla f = 0$$

.

The above equation is solved iteratively using an explicit Euler time integration method. The procedure `project_s` implements one time iteration of the solution. It computes the position of a particle p at the next time-step $t + \Delta t$ from its position at the current time t

$$p^{t+\Delta t} = p^t + \Delta t \;\; \mathrm{sign}(f(p^t)) \; \nabla f(p^t)$$

The iteration process is repeated until the particle is close enough to the surface (i.e. $|f(p^t)| < \epsilon$).

If the time step is too big, it is possible that the numerical simulation causes an overshoot. This would make the particle to oscillate forever from one side of the surface to the other. In order to prevent such a problem, the time step is reduced by 1/2 every time that the particle crosses the surface.

```
Vector project_s(step, p, v)
{
  p=v_add(p,v_scale(f_grad(p),sign(v)*step))
  if (abs(v1 = f_value(p)) < epsilon)
    return p;
  if ((v * v1) < 0)
    step /= 2;
  return project_s(step, p, v1);
}
```

Figure 6 shows schematically how an edge midpoint is converted to a particle and driven onto the surface by simulating the system dynamics.

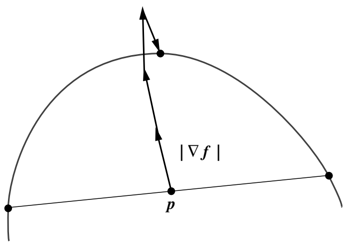


Figure 6: Projecting a new sample onto the implicit surface

7 Results

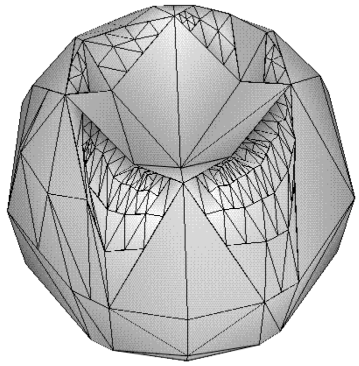
In this section we discuss the results of using our method for the adaptive polygonization of implicit surfaces.

The example is a blobby object, [4]. This type of model defines the implicit function as a density field generated from a point skeleton. The implicit surface is one of the level set of that field. In such a formulation, point sources can be combined either additively or subtractively in order to produce smooth blends.

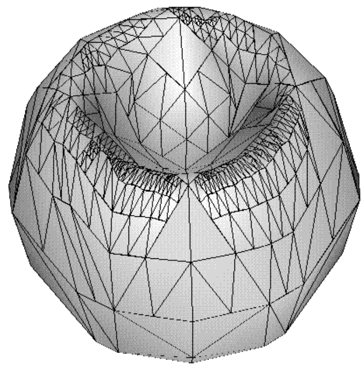
We chose this implicit model because it allows a fairly good control of the local surface curvature through the blending parameters.

The test object is a spherical shape with a cavity on top. It was constructed using a skeleton with two points consisting of one strong positive source and one weaker negative source.

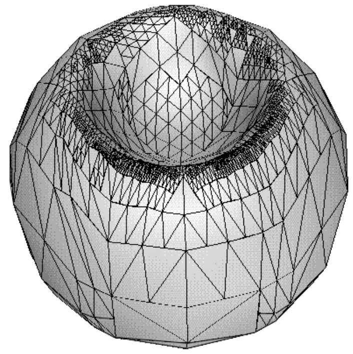
Figure 7 shows a sequence of polygonal approximations of this object. Figures 7(a), (b) and (c) depict a coarse, medium and high resolution versions of the model. They were generated with `dot_tol` set, respectively, to 0.2, 0.4 and 0.8. The corresponding polygonal meshes are composed of 298, 836 and 2351 triangles.



(a)



(b)



(c)

Figure 7: Sequence of polygonal approximations

Note that the density of the mesh increases mostly around the rim of the crater, where the subtracted material blend occurs. This area contains the sharpest variations in surface curvature.

Figure 8 shows a close up view of a portion of the triangle mesh with high polygon density.

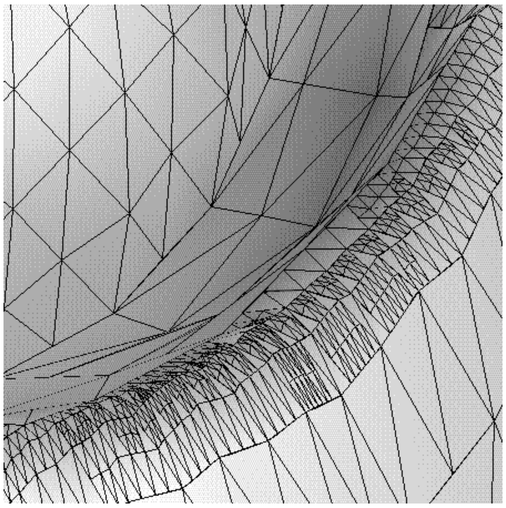


Figure 8: Detail of an area with high surface curvature

8 Conclusions

The adaptive polygonization algorithm presented in this paper is a simplification of the method introduced by the author in [17]. The main difference is that the original method employs a full 3D adaptation, making the implementation more complex and less efficient. In the new method we decompose the problem in two subproblems, such that most of the computation can be done in 2D. This makes the implementation simpler and more efficient.

A similar method was developed independently by Thad Beier of Pacific Data Images – PDI, [2].

This adaptation approach was incorporated by the author into Globograph's 3D modeling system. Since then, it has been used in many successful animation projects.

References

[1] E. L. Allgower and S. Gnutzmann. An algorithm for piecewise linear approximation of implicitly defined two-dimensional surfaces. *SIAM Journal of Numerical Analysis*, 24(2):2452–2469, April 1987.

[2] T. Beier, 1990. (personal communication).

[3] T. Beier. Practical uses for implicit surfaces in animation. ACM Siggraph Course Notes, 1990. Modeling and Animating with Implicit Surfaces.

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

[5] J. F. Blinn. The algebraic properties of homogeneous second order surfaces, 1984.

[6] J. Bloomenthal. Polygonization of implicit surfaces. *Comp. Aid. Geom. Des.*, 5(4):341–355, 1988.

[7] S. A. Cameron. Collision detection by four-dimensional intersection testing. *IEEE Trans. Robotics and Automation*, 6(3):291–302, 1990.

[8] L. H. de Figueiredo. *Computational Morphology of Implicit Curves.* PhD thesis, IMPA - Instituto de Matematica Pura e Aplicada, 1992.

[9] L. H. de Figueiredo, J. de M. Gomes, D. Terzopoulos, and L. Velho. Physically-based methods for polygonization of implicit surfaces. In *Proceedings of Graphics Interface 92*, 1992.

[10] J. Gomes and L. Velho. *Implicit Objects in Computer Graphics.* IMPA, 1993.

[11] M. Hall and J. Warren. Adaptive polygonization of implicitly defined surfaces. *IEEE Computer Graphics and Applications*, 10(6):33–43, 1990.

[12] Arthur J. Olson and David S. Goodsell. A functional view of proteins. *IEEE Computer Graphics and Applications*, 11(1):15–17, January 1991.

[13] A. Requicha. Representation for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4), 1980.

[14] A. P. Rockwood and J. Owen. Using implicit surfaces to blend arbitrary solid models. In G. Farin, editor, *Geometric Modeling: Algorithms and Trends.* SIAM, 1987.

[15] S. Roth. Ray casting as a method for solid modeling. *Computer Graphics and Image Processing*, 18(2):109–144, 1982.

[16] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):65–70, July 1992.

[17] L. Velho. Adaptive polygonization of implicit surfaces using simplicial decomposition and boundary constraints. In *Proceedings of Eurographics 90.* Elsevier Science Publisher, 1990.

[18] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.