Adaptive Polygonisation of Non-Manifold Implicit Surfaces

R.J. Balsys,
Faculty of Informatics & Communication,
Central Queensland University,
Rockhampton M.C., Qld. 4702. Australia.

K.G. Suffern,
Faculty of Information Technology,
University Technology, Sydney. PO Box 123, Broadway
NSW. 2007 Australia.
{balsys@cqu.edu.au, kevin@it.uts.edu.au}

Abstract

We discuss the polygonisation and rendering of non-manifold implicit surfaces using adaptive octree subdivision and interval arithmetic for surface exclusion in octree nodes. We present a new algorithm that polygonises some surfaces that self intersect, or have other non-manifold features such as separate sections that meet at points. Gradient information is used to resolve ambiguous polygonisations in plotting nodes. A line-stitching algorithm is discussed that allows for multiple polygons to be in a plotting node. We illustrate the algorithm with a number of surfaces that demonstrate its capabilities and limitations.

Keywords--- implicit surfaces, non-manifold, octrees, intervals, polygonisation, line-stitching.

1. Introduction

An implicit surface $S$ is the zero set of a function $f : \mathbb{R}^3 \to \{\mathbf{p} \in \mathbb{R}^3 \mid f(\mathbf{p}) = 0\}$. The function $f(\mathbf{p}) = f(x, y, z)$, can be defined by polynomial functions in $x$, $y$, and $z$, in which case the surface is algebraic, or defined by arbitrary non-polynomial functions. The surfaces can also be regular or non-manifold. Regular surfaces are locally Euclidean everywhere, and therefore everywhere homomorphic to a 2D disk. Non-manifold surfaces are not Euclidean everywhere, and include surfaces that self intersect, or have cusp points where their local curvature is infinite. The Klein bottle and Steiner’s Roman surface are famous examples of non-manifold surfaces. A general reference on implicit surfaces is Bloomenthal’s book [1].

Algorithms for rendering implicit surfaces must first locate the surfaces before they can be rendered. The robustness of the location process is critical because there may be little or no a priori information about the location of the surfaces, and implicit surfaces can also consist of an arbitrary number of disconnected sections. A number of authors have developed algorithms for rendering regular implicit surfaces using octree based adaptive subdivision techniques, starting with Bloomenthal’s classic paper [2]. This was followed by algorithms by Suffern [3], [4] [5], Suffern and Fackerell [6], Schmidt [7] and Balsys and Suffern [8]. These algorithms start with a user specified search cube that is recursively subdivided in an octree fashion while the surface is present in an octant. The subdivision is stopped when certain flatness criteria for the surface are met, or some maximum recursion depth is reached. The result is a series of cubes (called plotting nodes) that contain the surface, and in which it can be represented for rendering purposes by a polygon.

The early algorithms used point sampling to detect the surfaces, a process that is not robust, and can lead to the algorithms missing parts of the surfaces [3], [4]. Suffern and Fackerell [6] therefore supplemented the point sampling with interval arithmetic techniques [9]-[13], to improve the robustness of the detection process. Since the plotting nodes can be at different depths in the octree, cracks can appear in the surface between adjacent nodes at different depths. Bloomenthal [2] and Balsys and Suffern [8] developed algorithms to fix these cracks.

Non-manifold features can create problems for polygonisation algorithms, where self-intersections complicate the polygonisation process, and cusp points are essentially non-polygonisable. Few papers have been published on the polygonisation of non-manifold implicit surfaces. Schmidt [7] discusses adaptive polygonisation of self-intersecting implicit surfaces. He performs a
tetrathedral volume splitting of any plotting nodes at the maximum subdivision depth that contain the surface but have not been unambiguously resolved into a single polygon. This occurs along the curves of self-intersection.

Bloomenthal and Ferguson [14] provide an extensive discussion of the polygonisation of non-manifold implicit surfaces using uniform subdivision of the search cube to a maximum depth. Again, nodes that contain the surface, but have not been resolved, are subdivided into tetrahedra for polygonisation. They describe a number of problems in doing this, including finding roots on tetrahedral edges and methods to disambiguate the resulting lines on tetrahedral faces.

We present here a new algorithm that adaptively polygonises some surfaces that self intersect, or have other non-manifold features that occur on plotting cell faces. Due to the length of the algorithm we do not give it here but it can be found on the authors web site [15]. Our method uses an octree spatial data structure, interval arithmetic as an exclusion test for the adaptive subdivision, and uses the divergence of node vertices and point sampling to drive the subdivision process. Our method requires that at most two approximately planar sections of the surface exist in the plotting node, see Figure 1.

This involves a new root finding technique to handle the case where more than one root lies on a plotting node edge. In this case we use the gradient of the surface at the vertices of the plotting node to help find the roots. Our algorithm correctly handles the case in Figure 1(a) where the surface self intersects, although it doesn’t split the polygons along the line of self-intersection.

After discussing the subdivision and polygon reconstruction aspects of the algorithm in the following two sections, we apply the algorithm to several surfaces that have been difficult to polygonise in the past. These include a horned cycloid, the Klein bottle, and Kummer’s surfaces. We also discuss the limitations of the algorithm.

2. Adaptive subdivision and polygonisation

We start with a brief review of our self-intersection adaptive subdivision algorithm. This algorithm is an adaptation of a similar algorithm in [8] where we specify a search cube that defines the space in which we search for the surface, and then render it. This cube is the root node for the octree based adaptive subdivision algorithm. We also specify minimum and maximum subdivision depths, as the minimum depth can be used to force the subdivision of nodes to a minimum depth, and the maximum depth is used to stop the subdivision in regions of extreme curvature. We use the interval extension of the function to exclude nodes that don’t contain the surface, as in Suffern and Fackerell [6]. If the surface is not present in any of the eight sub-nodes of a parent node, we return; otherwise we check if the minimum subdivision depth has been reached. If not, we subdivide until the minimum depth is reached.

Having reached the minimum subdivision depth, we use a function CreatePolygons which attempts to create polygons in the node. If a valid polygonisation occurs, CreatePolygons returns true and the polygons are returned in a Polygons array. The relative flatness of the polygons are checked in PolygonFlat using criteria as given in [2] and [8]. If the polygons pass the flatness test they are tested for "cracks" and "repaired" if necessary (see [2] and [8]) using function FixHoles. Finally, the polygons are inserted into the polygon display list using an InsertPolygons function.

Figure 1. (a) polygons self intersect in a plotting node and (b) polygons do not intersect in node, show the two ways in which two lines on a face can be used to produce two polygons from vertex points in a node. The gradient function $f$ is used to disambiguate the two cases, as discussed in the text.
Figure 1(a) or the surface sections do not intersect, as in Figure 1(b).

We now discuss the implementation of CreatePolygons. This constructs the polygons in a plotting node, and is different from the techniques used in previous papers [2], [4], [7], [8] and [14]. In CreatePolygons we have the following information: the minimum coordinates \( (x, y, z) \) of the plotting node and its width \( d \), the surface function \( f \), and the gradient function \( g = \nabla f \). The function returns the number of polygons, if any, found in the plotting node, and the arrays of points defining those polygons.

In CreatePolygons we extend the definition of polygonisable cells given in [6] to include cases where a surface segment crosses an edge of the face of the plotting node at most twice and is locally flat as given by planarity and divergence criteria of Bloomenthal [2]. If the surface crosses an edge of a plotting node more than twice, it will fail to polygonise correctly. We come back to this point later.

Each edge of each face of the plotting node must be considered in turn. For a particular \([\text{face}][\text{edge}]\) pair, a sign and gradient test are performed. These tests are:

\[
\text{Sign Test: } (f_a * f_b > 0) \\
\text{Gradient Test: } (g_a * g_b > 0.9)
\]

(1)

Where \( f_a \) and \( f_b \) are the values of \( f \) at the endpoints \( a \) and \( b \), of the edge being tested, and \( g_a \) and \( g_b \) are the corresponding gradient functions. The value 0.9 corresponds to an angle of about 25° between the gradient vectors and we have found this gives satisfactory results.

The following logical expression is then evaluated for each edge of the face:

\[
\text{Subdivision Test: } (f_a * f_b > 0) \text{ AND } (g_a * g_b > 0.9)
\]

(2)

If the sign and gradient test in Equation (2) contradict one another along an edge, we consider the following four possibilities:

1. No intersection along an edge. This occurs when the level surface \( f = 0 \) does not intersect the edge.
2. One intersection along the edge. This occurs when \( f = 0 \) lies on the edge.
3. Two intersections on an edge. This occurs when \( f = 0 \) cuts the edge twice as in Figure 2(a).
4. More than two intersections along the edge. This occurs when there are more than 2 roots along the edge.

Our algorithm handles cases (1)-(3). When zero intersections are found on the edge there is no contribution to lines on the face for that edge. How close can a surface come to making contact with the edge before we accept that case (2) has occurred? We use a zero tolerance for this to preserve the topology of the situation where two parts of a surface closely approach (but do not touch) a single node face.

In the cases (2) and (3) where we have 1 or 2 roots on an edge, we wish to find a point \( C \) as shown in Figure 2. For the single intersect case we wish to find the point \( C \) which, to within \( \varepsilon \), represents the minimum signed distance between the intersects in Figure 2(a) before we consider that the two roots can be replaced by one as in Figure 2(b).

In case (2), we find the single root on an edge with regula falsi but in case (3) we can’t initially use this technique to find the two roots because the function values at the ends of the edge, \( A \) and \( B \), have the same signs. Instead, we use binary interval subdivision, combined with a constraint based on the magnitude of the function. The approach is as follows. The function value \( f \) is calculated at \( a = A, b = B \) and \( c = (A + B) / 2 \) resulting in values \( f(a), f(b) \), and \( f(c) \) on the edge. The products \( f(a)*f(c) \) and \( f(b)*f(c) \) are calculated and if either is negative we return the point \( C = c \), and 2 as the number of roots. In this case \( C \) is a point in the interval for which regula falsi will find roots in intervals \( AC \) and \( CB \). If however, \( |f(c)| < \varepsilon \), we have found a double root of \( f \) on the edge at \( c \), so we return \( C = c \) and one root of the function, even if the products indicate otherwise.

If no solution is found, the method iterates a set number of times. If \( abs(f(a) * f(c)) < abs(f(b) * f(c)) \) the method iterates with \( B = c \); otherwise it iterates with \( A = c \). If no root is found within the required number of iterations, the method returns with \( C \) undefined, and the number of roots set to zero. We used a maximum of 25 iterations with \( \varepsilon = 1e^{-9} \) with satisfactory results.

A number of combinations of lines on a face can occur when there are two intersections on two edges of a face. DoubleIntersectsOnEdges is the algorithm that we use to form lines out of the roots on edges in these cases, and an example is given in Figure 3 (a). We can determine which combinations of lines to use with the dot products of the gradients at the two intersections, and the other two roots.

A further possibility can occur when the distance between both roots on both edges becomes less than some small value \( \varepsilon \). In this case the line of self-intersection of the surface lies on the face of the plotting node, as shown in Figure 3 (b). Such lines belong to two polygons in the plotting node and so two corresponding lines are put into an array for the polygon stitching routine.

We also have to deal with a number of other special cases. The possibility exists that the surface passes exactly through one or more plotting node vertices. This...
situation complicates the correct polygonisation of the node. In this case $f(A) * f(B) = 0.0$ for an edge, and we count the number of zero vertices on a face. If there is more than one such vertex, we call a function $\text{LineOnFaceAndEdge}$, as shown in Figure 3 (c). Otherwise, if we have a face that has two edge roots found with the sign and gradient test, we call $\text{DoubleIntersectsOnEdge}$. Finally we have the possibility of a single line on the face, as in Figure 3 (d), and we call $\text{LineOnFace}$. In these functions the points found on the edges of the face are used to form lines on the face, which in turn become the segments of the polygon in the plotting node.

![Figure 3](image)

Figure 3. Combinations of lines on faces. (a) 2 lines with intersects on two edges, (b) 2 lines within of each other on face, (c) line on face with endpoints on vertex and (d) single line on face.

In functions $\text{LineOnFace}$, $\text{LineOnFaceAndEdge}$, and $\text{DoubleIntersectsOnEdge}$, edges that are shared by two polygons are placed in a $\text{nonManLine}$ array, whereas those that only occur in only a single polygon are placed in a $\text{line}$ array. The arrays $\text{nonManPlane}$ and $\text{plane}$ are used to store the face numbers of the lines in the $\text{nonManLine}$ and $\text{line}$ arrays. The total numbers of lines in the $\text{nonManLine}$ and $\text{line}$ arrays are stored in $\text{numNonManLines}$ and $\text{numLines}$ variables respectively.

When all faces of the plotting node have been treated in this manner we end up with two arrays, $\text{line}$ and $\text{nonManLine}$, that hold coordinates for the lines on the faces, and corresponding arrays that hold the face numbers. The face number is used in the $\text{FixHoles}$ function and polygon outline functions as discussed in [8]. The following section details how we use the information in these variables to create polygons in the plotting node.

The above criteria, although extensive in the number of special cases, produce good results when used to polygonise self-intersecting surfaces and surfaces with cusp points and infinitely thin sections, such as Kummer’s surface (Figure 5), the cyclide surface (Figure 6), the the Klein Bottle (Figure 7), and Steiner’s surface (Figure 8).

3. Polygonisation by Line Stitching

The procedure used here is similar to that described in [9], but in this work two arrays $\text{line}$ and $\text{nonManLine}$ hold line values. The $\text{StitchPolygon}$ function creates a polygon at a time by joining lines when they have exact matching end coordinates. A polygon is complete when we add a line to the polygon line list that has the same start and end coordinates as the start and end coordinates of the polygon list being constructed.

The most common case that $\text{StitchPolygon}$ has to deal with is a single polygon in the plotting node. The next most common case is where there are two polygons, as in Figure 1. In this case, $\text{StitchPolygon}$ returns the two polygons, whether they intersect, as in part (a), or don’t intersect, as in part (b). Specifically, we currently don’t find the line of intersection and split the polygons along this line, because this would further complicate the already complex $\text{StitchPolygon}$ code. As it turns out, a lot of the interesting self-intersecting surfaces have curves of self-intersection in the coordinate planes, and can therefore be handled by the situation illustrated in Figure 3.

![Figure 4](image)

Figure 4. Two polygons intersect along the black line on the top face of plotting node. The + and – are the signs of $f=0$ at plotting node vertices.

When we have two polygons in the plotting node we use a line in the $\text{line}$ array as the starting point, and then iteratively process the rest of the lines in the array. In each iteration we test both end coordinates of the line to see if they join to the start or the end of the polygon point values. In some cases this requires that we swap the end coordinates of the line so that a match can be made. The tests need to be defined to within some $> 0$. We use an that’s within $1/1000$ of the length of the plotting node size to maintain a constant precision with respect to the size of the node being polygonised. When a match is found, the line’s coordinates are removed from the $\text{line}$ array. If a match can’t be found, the line is left in the $\text{line}$ array, and all the other lines are checked in turn.

After a match is found from the $\text{line}$ array, we check the lines in the $\text{nonManLine}$ array, provided it’s not empty, to see if a match that closes the polygon list can be found. See Figure 4. Since the black line on the face is
shared by two polygons in the plotting node, two versions of the line are kept in the nonManLine array. Polyonisation starts with any of the non-shared edges in the plotting node. Successive lines are joined by matching end coordinates until the point is reached when the black line from the nonManLine array joins the start to the end of the polygon. At the end of this sample process the nonManLine array has one less line in it and the line array has two or more fewer lines in it. In Figure 4 a new polygon defined by a sequence of four points is created in the process.

4. Surface Examples

In this section we use a number of example surfaces to illustrate our algorithm’s capabilities and limitations for polygonising surfaces with non-manifold features. All polygon rendering is performed with the painter’s algorithm.

The first example is Kummer’s large family of quartic surfaces [16]. These are given by the algebraic equation

\[
\left( x^2 + y^2 + z^2 + 2u^2 \right)^2 - pqrs = 0
\]

where \( p = \frac{3a^2 + 1}{3} \), \( q, r, \) and \( s \) are the tetrahedral coordinates: \( p = z \sqrt{2x} \), \( q = \frac{\sqrt{2y} + \sqrt{2y} + \sqrt{2y}}{3} \), \( r = \frac{\sqrt{2y} + \sqrt{2y} + \sqrt{2y}}{3} \), \( s = \frac{\sqrt{2y} + \sqrt{2y} + \sqrt{2y}}{3} \). In addition, \( z \) is a scaling parameter which we take to be unity. The parameter \( u > 0 \) controls the number of double points for the surface. These are singular points where separate parts of the surfaces meet. Figure 5 shows two Kummer surfaces with different numbers of singular points. These surfaces have been difficult to render with previous polygonisation algorithms, but as Figure 5 illustrates, our current algorithm is able to successfully render the surfaces near the singular points.

Another family of surfaces that can have non-manifold features are the cycloids [17], given by

\[
\left( x^2 + y^2 + z^2 \right)^2 - 2\left( x^2 + r^2 \right) f^2 = 0
\]

\[
2\left( y^2 + z^2 \right) a^2 f^2 + 2afrz + \left( a^2 + f^2 \right) = 0.
\]

When \( a = 10 \), \( r = 2 \), and \( f = 2 \), we have a horned cycloid that becomes infinitely thin at a point. As Figure 6 (a) shows, our previous algorithm [8] had difficulty polygonising the surface near this point. But as the results in Figure 6 (b) show, our current algorithm can correctly render this part of the surface, at least to the pixel resolution of the image. There are two reasons why the polygonisation works in this case. The first is that the horns meet at a plotting node vertex, and second is that the thin sections lie along plotting node edges. See the discussion below on Steiner’s Roman surface.

The Klein bottle surface, which can be expressed implicitly as

\[
f(x,y,z) = \begin{vmatrix} x^2 + y^2 + z^2 + 2y & 0 \\ x^2 + y^2 + z^2 & 2y \end{vmatrix} - 8z^2 = 0
\]

self intersects. Figure 7 (a) shows the outside of this surface and Figure 7 (b) shows a cross section to reveal its inner structure. Our algorithm correctly renders this surface along the curve of self-intersection because it lies in the \((x, y)\) plane. This results in the two polygons in the plotting nodes intersecting in the node faces, as in Figure 4.

When we rendered Steiner’s Roman surface,

\[
f(x,y,z) = x^2 + y^2 + z^2 + x^2 z^2 + yz = 0,
\]

in Figure 7, we found that in some plotting nodes one line remains in the nonManLine or line arrays after the final complete polygon was found. These nodes had a coordinate axis along an edge, and the extra lines were caused by the double lines along \( x = 0, y = 0, z = 0 \). As a result, we allow for the creation of degenerate polygons in our algorithm. Such polygons have one edge and no area but are still rendered.

Figure 8. Steiner’s Roman surface (6) showing the 3 double lines (clipped to the view volume) along the Cartesian axes.

This approach allows us to render the three double lines of Steiner’s surface. However, we note that these lines are only detected because they lie along edges of plotting nodes. Although these lines go to infinity, they are clipped to the view volume in Figure 8. Sederburg and Zundel’s [18] scan-line algorithm also found and rendered these double lines.

Steiner’s surface also self intersects along the coordinate axes, which allows us to correctly render this surface along these lines. The polygons again intersect along plotting node edges.

Because of the self-intersections along the coordinate axes, the curved outer section becomes cusp-like as it crosses the axes. This creates the jagged appearance at the top of the surface in Figure 8. Although this isn’t particularly accurate or attractive, our algorithm has still rendered Steiner’s surface as a closed surface.
5. Summary and Future Work

In this work we make the following contributions to the polygonisation and rendering of non-manifold implicit surfaces with adaptive subdivision.

- A new subdivision criteria for adaptive polygonisation based on a combination of function values at node vertices and the divergence of the gradients of the surface at node vertices.
- A polygon stitching algorithm that can be used when up to two surface sections are in a plotting node.
- A method for calculating function roots along edges of a plotting node when two function zeroes occur along the edge.

The above algorithm can also handle cases where function zeroes lie on one or more plotting node vertices.

Using the algorithm, we have been able to successfully polygonise and render a number of non-manifold surfaces that have been difficult to render with previous polygonisation algorithms.

The algorithm needs to be extended to split the intersecting polygons in Figure 1 (a) into four polygons. This would allow the painter’s algorithm to correctly render surfaces that self intersect along arbitrary curves.

It would also be useful to extend the algorithm so that it could correctly polygonise the more general case of rendering self-intersections where the lines of self-intersection lie inside the plotting node itself. (Isn’t this the same as the previous paragraph?) Finally it would also be useful to extend the algorithm so that it could correctly polygonise $C^0$ surfaces where ridges lie inside the plotting nodes. It can currently only do this when a ridge coincides with plotting node faces or edges.

Acknowledgements

The authors will like to acknowledge the support given by their respective universities of this work.

References

Figure 5. Kummer surfaces (a) $a = 2/3, \quad b = 1$ depth = 8, cutaway by plotting node to reveal inner detail, 4 double points, 
(b) $a = \sqrt{2}, \quad b = 1$, maximum plot depth = 9, 15 double points.

Figure 6. The cyclide surface (4) with $a = 10, r = 2$ and $f = 2$. (a) Previous work [8]. (b) Current work.

Figure 7. (a) Klein bottle (5) surface, (b) cross-section.
# Table of Contents

**Preface** ........................................................................................................................................................................... xi
**Organization** ......................................................................................................................................................................... xii
**Reviewers List** ....................................................................................................................................................................... xvi

## Imaging

**Combinations of Range Data and Panoramic Images - New Opportunities in 3D Scene Modeling** .......................................................... 3  
*R. Klette and K. Scheibe*

**Imaging Retrieval**

**A Method to Index and Retrieve Images to Improve Reasoning during the Design Process** ............................................................ 13  
*S. Kacher, J.-C. Bignon, G. Halin, and P. Humbert*

**Texture Feature Fusion for High Resolution Satellite Image Classification** ................................................................. 19  
*Y. Zhao, L. Zhang, and P. Li*

**Sketch Retrieval Based on Spatial Relations** ...................................................................................................................... 24  
*S. Liang, Z. Sun, and B. Li*

## Imaging - Recognition

**Corresponding Points Matching Based on Position Similarity** ................................................................................................. 33  
*J.-J. Pan and Y.-N. Zhang*

**Concealment of Damaged Block Coded Images Using Intelligent Two-Step Best Neighborhood Matching Algorithm** ................. 38  
*L. Xiao, C. Huang, H. Liang, and H. Wu*

**An Edge Based Segmentation Algorithm for Rock Fracture Tracing** ....................................................................................... 43  
*W. Wang*

**Image Segmentation of Irregular Shape Grains on Ceramic Material Surfaces** ................................................................. 49  
*W. Wang*

**Pattern Recognition Based Color Transfer** .......................................................................................................................... 55  
*J. Ying and L. Ji*

**Head Nod and Shake Recognition Based on Multi-view Model and Hidden Markov Model** ............................................ 61  
*P. Lu, M. Zhang, X. Zhu, and Y. Wang*
Medical Visualisation

A Practical Rebinning-Based Method for Patient Motion Compensation in SPECT imaging ................................................................. 209
L. Ma, S. Gu, S. Nadella, P. Bruyant, M. King, and M. Gennert

Visualisation within a Multisensorial Surgical Planner ................................................. 215
G. Clapworthy, M. Krokos, R. Mayoral, R. Liang, and D. Podgorelec

A Novel Medical Image Registration Method Based on Mutual Information and Genetic Algorithm ......................................................... 221
H. Zhang, X. Zhou, J. Sun, and J. Zhang

Real-Time Predefined Shape Cutaway with Parametric Boundaries ................................. 227
R. Liang, G. Clapworthy, M. Krokos, and R. Mayoral

Surface Hatching for Medical Volume Data .................................................................. 232
Y. Cai and F. Dong

Computer Graphics

Adaptive Smooth Principal Curves Design................................................................. 241
L. Zhang and Z. Luo

Perspective Correct Normal Vectors for Phong Shading ................................................. 245
H. Zhang, C. Zhu, Q. Zhao, and H. Shen

PDE-Driven Implicit Reconstruction of 3D Object ........................................................ 251
H. Zeng, Z. Liu, and Z. Lin

Adaptive Polygonisation of Non-manifold Implicit Surfaces .......................................... 257
R. Balsys and K. Suffern

Solid Blends of High Order Continuity .................................................................. 264
J. Zhang and L. You

Texture Driven Pose Estimation ........................................................................ 271
B. Rosenhahn, H. Ho, and R. Klette

Fast Individual Facial Animation ........................................................................ 278
J. Yao, M. Zhang, X. Zeng, and Y. Wang

Real Time Interactive

Real-Time Dynamic Cloud Modeling and Rendering ................................................... 285
B. Qin and L. Tao

3D Scene Transmission for Web-Based Shiphandling Training .................................. 291
C. Xie, X. Liu, and Y. Jin