Chapter 4: Non-Manifold Surfaces

Creative imagination is still of first importance
to the design engineer, and it should be fully developed.
Here, Nature is the great master teacher.
(Heinrich Hertel)

4.1 Introduction

As described in section 3.5.4, implicit surfaces are two-dimensional manifolds. That is, they envelope a volume. A manifold surface is, everywhere, locally homomorphic (that is, of comparable structure) to a two-dimensional disk (manifold surfaces *with boundary* are everywhere homomorphic to a disk or a *half*-disk). For example, a disk may be fully applied to any portion of the torus below, left. But, the disk does not fully apply to all points of a teapot, below, right. In particular, the disk is truncated along the upper boundary of the teapot bowl.

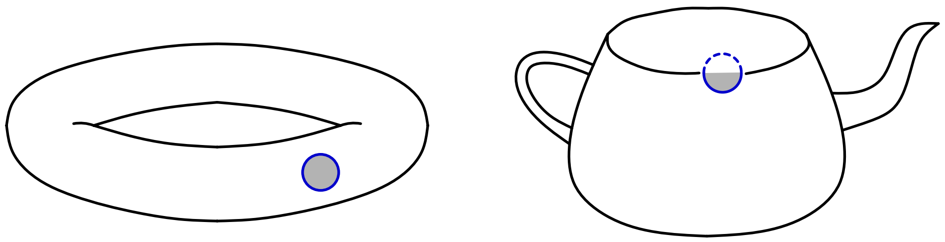


Figure 4.1 Manifold and Manifold-with-Boundary Surfaces

Any tessellation (for example, a polygonization) of a manifold surface will produce edges that are of degree two; that is, all edges are shared by exactly two faces. Tessellations of non-manifold surfaces produce edges of degree 1, 2, 3, or more. A polygonized, bounded implicit surface has edges of degree two, whereas many parametric surfaces, such as patches, have boundary edges of degree one. In the case of parametric surfaces, it requires explicit effort by the designer to define a closed surface, that is, a surface without boundary. The availability of boundary edges is an obvious design asset; with it, the designer may construct objects with openings or flat projections. Representations for edges of degree three or greater exist [Weiler 1986], but are not commonly employed in geometric modeling.

The manifold surface is appropriate for many natural forms; for example, the integument of an animal usually encloses a volume. But certain natural objects, such as hair, feathers, or leaves, are not representable as a volume. For example, we may reasonably consider the leaf to be a combination of volume (the veins) and surface (the leaf blade), as depicted below. Such combinations pose a dilemma as to their geometric representation.

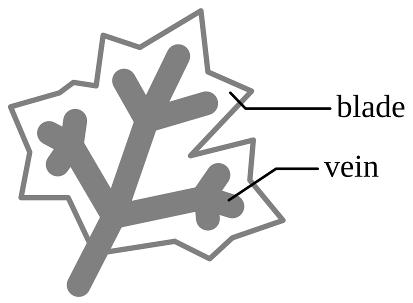


Figure 4.2 A Natural Form Consisting of a Volume and a Surface

As shown in the following illustration, the embedding of vein within blade can be represented as surface only (far left), surface and volume (left), thin volume (right), and volume and trimmed surface (far right). Clearly, the first representation is the least faithful to the intended surface. The second representation is not compact and, for images involving translucent surfaces, will not produce the desired result.

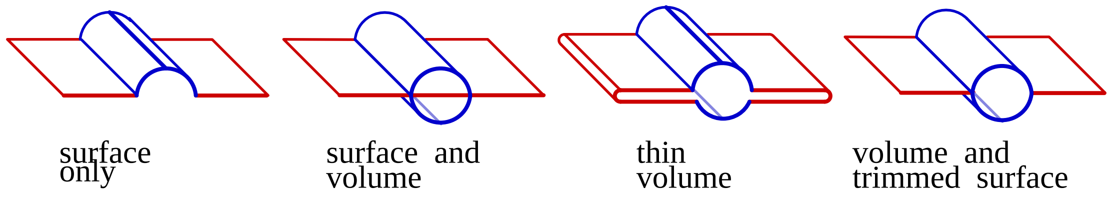


Figure 4.3 Possible Combinations of Vein and Blade

The third representation offers considerable difficulty for the sampling process. Let us consider this issue applied to a simpler geometry. For the remainder of this section, we employ a sphere embedded in a square, rather than a vein embedded in a blade. The sampling rates for a ray-tracer and a polygonizer are compared below for a thin-volume representation and a surface-only representation. A high sampling rate is required to ray-trace or polygonize a thin volume, resulting in an inordinate number of polygons.

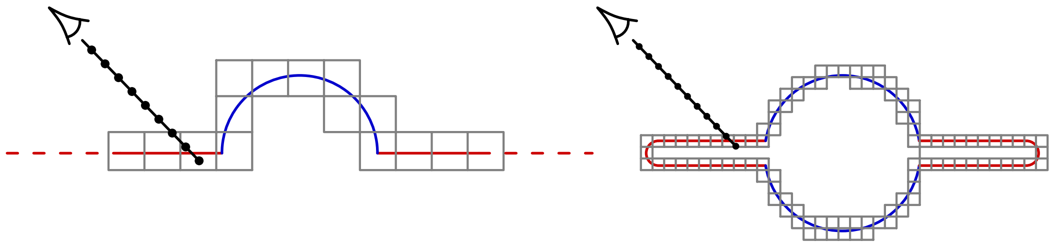


Figure 4.4 Polygonization and Ray-Tracing Sampling Rates

top: coarse sampling is sufficient for surfaces

bottom: fine sampling is required for thin volumes

Therefore, we regard the 'volume and trimmed surface' representation, which contains edges of degrees one, two, and three, as the only accurate, compact, and unified representation in figure 4.3.[1] Although the joins of parametric surfaces [Farin 1988] and the blends of implicit volumes [Rockwood 1989] have received careful study, the representation of a combined surface and volume has received relatively little attention. In this chapter, we consider the extension of our skeletal design representation, which consists of skeletal elements, skeletal primitives, and an implicit surface, to include the combination of surface and volume.

An essential aspect of such a combination is the trimming away of those parts of the surface contained within the volume. Trimming one surface against another or against a volume appears difficult when the surface or volume is compactly expressed in parametric or implicit form [Crocker and Reinke 1987], [Patrikalakis 1993] Alternatively, the surface and volume can first be converted to a concrete representation, such as polygons. Even so, the task of trimming an arbitrarily large set of polygons against another arbitrarily large set is computationally challenging.

We can reduce the computational burden by spatially partitioning the object. Within each cell, the surface can be approximated by a polygon trimmed against the volume, and the boundary of the volume can be approximated by another polygon. In order that a) a single, unified object results, and b) no gap exists between the surface polygon and the volume polygon, the volume polygon is divided at the trimmed surface, as shown below.

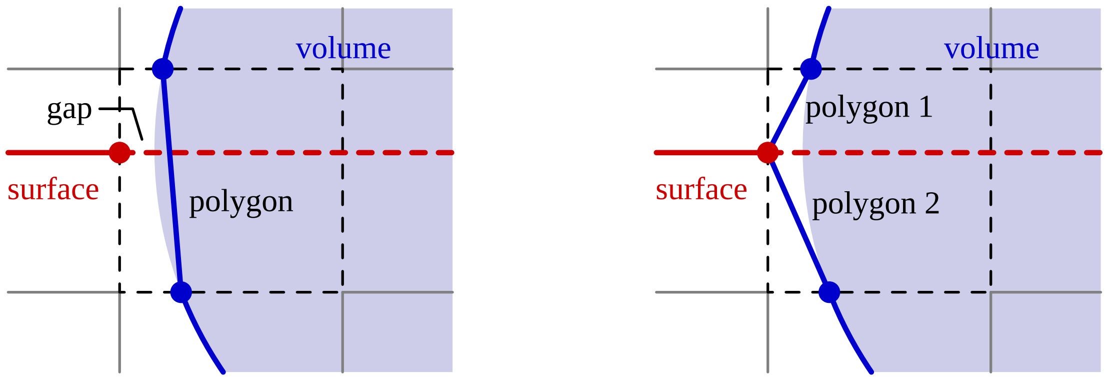


Figure 4.5 Combination of Surface and Volume (side views)

left: surface clipped to cell and volume approximated by polygon

right: gap eliminated by dividing volume polygon

This process may be codified as a set of operations applicable to particular cells. In the example illustration below, there are five cell types:

Cell 1 contains neither surface nor volume, and no operation is performed.

Cell 2 contains the surface but no detectable volume. Therefore, that piece of the surface contained within cell 2 is approximated by a single polygon.

Cell 3 contains surface and volume. The surface is trimmed to this cell and connected to the polygons that approximate the volume boundary.

Cell 4 also contains surface and volume, but the surface is fully enveloped by the volume. Thus, polygons approximating the volume boundary are produced but the surface piece is not.

Cell 5 contains only volume. Thus, polygons approximating the volume boundary are produced.

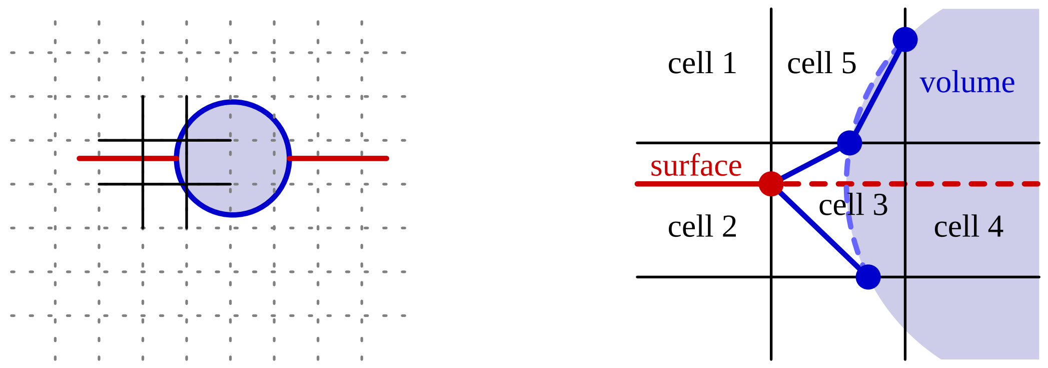


Figure 4.6 Partitioning of Surface and Volume

left: overall geometry, right: different cell types

A spatial partitioning scheme such as this could be employed to produce a concrete representation of a non-manifold surface. We prefer, however, a method within the framework of conventional polygonization, described in the previous chapter. Additionally, we prefer a method of surface definition similar to those based on the skeleton. That is, we seek to broaden the scope of implicit surface modeling.

4.2 Binary and Multiple Regions of Space

As described in section 3.1, an implicit surface separates regions of space for which $f(\mathbf{p}) < 0$ from regions for which $f(\mathbf{p}) > 0$. This binary regionalization of space provides a definition for the 'volume only' shape in figure 4.3, as shown below. It can also define the 'surface only' and 'surface and volume' shapes, if we ignore the unbounded aspect of the horizontal surface. The preferred shape, the 'volume and trimmed surface,’ cannot be defined by a binary regionalization of space, which demonstrates two restrictions associated with binary regionalization. First, it is not possible to combine differently dimensioned, bounded objects. Second, it is not possible to bound a two-dimensional object.

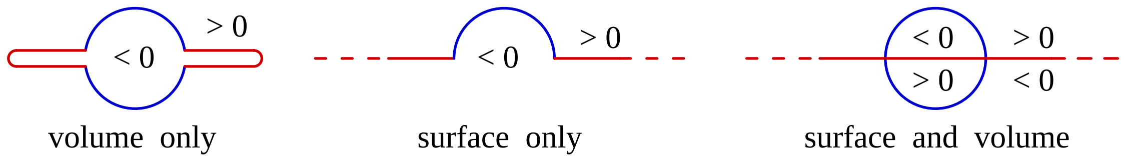


Figure 4.7 Possible Implicit Definitions (side view)

Conventional polygonizers require binary regions of space for their operation because they assume a) that *f* is continuous and, b) that points on opposite sides of the surface have oppositely signed values. Conventional polygonizers approximate an implicit surface by tessellation. Thus, they produce, for a bounded object, a manifold surface. The conventional polygonizer will not produce an edge with only one face, that is, a *boundary edge*. Nor will it produce an edge with three or more faces, that is, an *intersection edge*. In contrast, any tessellation of a volume embedded in a surface, such as the sphere-square shown below, is non-manifold and requires edges of degree one, two and three.

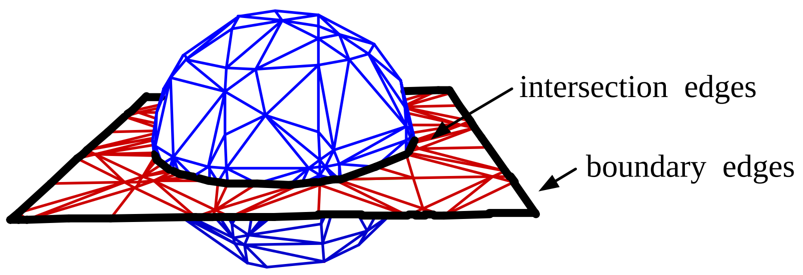


Figure 4.8 Tessellation of a Non-Manifold

sphere/square intersection edges have degree three

square boundary edges have degree one

In this chapter we describe a new method for defining and polygonizing non-manifold implicit surfaces. The method differs from conventional polygonization in that it permits multiple, rather than binary, regions of space.[2] For example, the sphere-square can be defined by four regions, as shown below.

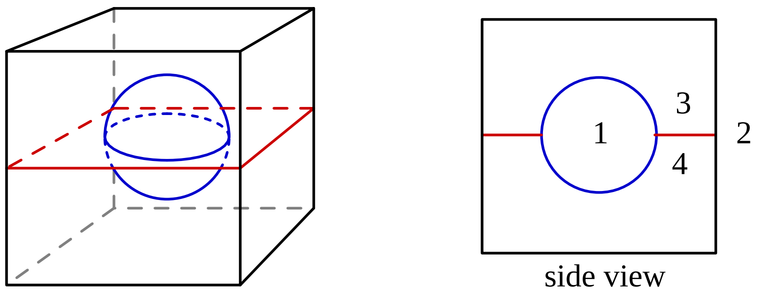


Figure 4.9 Multiple Regions define the Sphere-Square

Here, a cube bounds the plane, creating the square. The cube itself is not of interest; we wish to produce only those *surfaces of interest* that separate regions 1 from 3, 1 from 4, and 3 from 4. Accordingly, our non-manifold implicit surface definition consists of two parts: a) an integer-valued *region function*, f_{reg}, that returns the *region value* of a point, and b) a set of *region-pairs* of interest. For the example above, the region pairs are {(1, 3), (1, 4), (3, 4)}, and $f_{reg}(\boldsymbol{p})$ is:

$1,\ \|\boldsymbol{p}\| < r$

$2,\ max\,(\|\boldsymbol{p}_x\|, \|\boldsymbol{p}_y\|, \|\boldsymbol{p}_z\|) > s$

$3,\ \boldsymbol{p}_z > 0$

$4,$ otherwise,

where r is the radius of the sphere and s is half the length of the square's side.

The use of multiple regions is conceptually simple, and may be implemented following the fundamental steps of conventional polygonization. There are, however, significant differences that complicate the implementation of non-manifold polygonization. Before examining these issues, we consider related work.

4.3 Non-Manifold Representations in Solid Modeling

The literature offers little on the subject of non-manifold surfaces and mixed dimensional modeling. Prior work appears as sculptured surface extensions to boundary representation solid modelers [Miller 1986]. The extension to non-manifolds relaxes the requirement that constructive primitives be solid. [Rossignac and O'Connor 1989] and [Paoluzzi *et al*. 1993] discuss the use of a simplicial complex to represent objects defined in differently dimensioned spaces. These works develop a calculus that permits the application of the standard boolean operations upon the simplicial complexes. At any time, the complexes may be reduced to a boundary representation.

In [Muuss and Butler 1990], operations on non-manifold CSG representations are specifically developed for the purpose of surface tessellation. As with the other CSG approaches, a boundary representation is constructed according to binary evaluations described by a CSG tree. There is, apparently, the additional requirement that the topology of the resulting surface be pre-established.

As observed in section 3.2, the pairwise construction of intermediate CSG structures requires attention to numerical accuracy and significant case analysis of edge/edge and edge/surface intersections. These requirements also apply to non-manifold surfaces in CSG. The complexes developed in [Rossignac and O'Connor 1989] and [Paoluzzi *et al*. 1993] are intended to support this constructive process.

Absent from [Rossignac and O'Connor 1989] and [Muuss and Butler 1990] are concrete examples. The examples given in [Paoluzzi *et al*. 1993] are limited in scope. In this chapter we provide a number of non-manifold definitions and their polygonal approximations.

4.4 Comparison with Conventional Polygonization

As described in section 3.5.1, conventional polygonizers of continuous functions partition space by continuation. The propagation from one cell to the next is controlled by examining the function polarity at each cell corner; a new cell is formed across any face that contains corners of differing polarity. For the non-manifold polygonizer, however, cells are propagated across only those faces that contain a surface of interest, as defined in section 4.2.

The indexing schemes for cubes and cube edges described in section 3.5.3 are also employed in our implementation of a non-manifold polygonizer. Unlike the conventional polygonizer described, we require the decomposition of each cube into tetrahedra. As we soon demonstrate, non-manifold polygonization of an individual cell is considerably more complex than for conventional polygonization. The tetrahedron is the basic three-dimensional polyhedron (*i.e.*, it is a simplex) and its use significantly simplifies the implementation of non-manifold polygonization.

Conventional polygonizers reasonably assume that a surface passes through a polygonizing cell at most once. Thus, only a single vertex occurs along a cell edge that connects differently signed cell corners. This results in zero, three, or four edge vertices within a tetrahedron, as shown in figure 3.7. Also, at most a single polygon edge, or *face line*, is formed across a tetrahedral face. Traversing from one face line to the next, around the cell, produces a single three or four-sided polygon.

The non-manifold polygonizer described in this chapter performs the conventional steps of edge vertex computation, face line formation, and polygon creation, but also accommodates face vertices, multiple edge vertices, and multiple face lines.

4.4.1 Face Vertices

As a consequence of multiple regions of space, more than two regions can occur within a tetrahedral face. This yields more than two edge intersections on the face as well as an intersection internal to the face. For example, consider the three regions of space that meet along the circular intersection of sphere and square, figure 4.9. In the illustration below, left, all three regions are spanned by a single tetrahedron; indeed, all three regions are spanned by a single tetrahedral face, as shown below, middle. This suggests the computation of three edge vertices and a *face vertex*, and their connection by three face lines, as shown below, right.

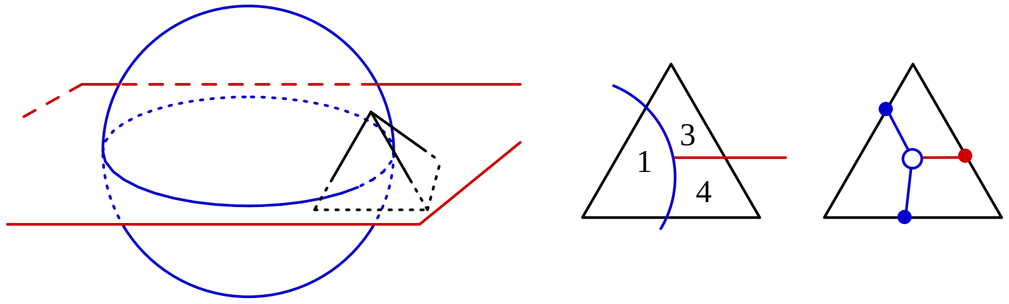


Figure 4.10 Need for Face Vertex

The use of a face vertex introduces significant implementation complications. Therefore, we briefly consider alternative face lines, shown below, that do not require a face vertex. The left method is topologically inconsistent because it leaves an undefined region within the face. The middle approach, in effect, moves the face intersection and two of the edge intersections to a cell corner, yielding a single line within the face. At right, the face intersection is moved to one of the edge vertices. The latter two methods produce, essentially, an inaccurate approximation to the face vertex.

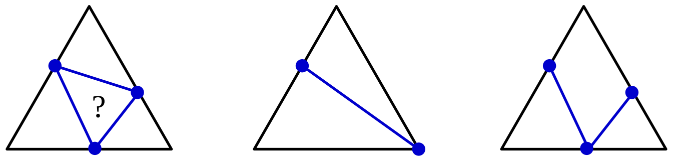


Figure 4.11 Alternatives to the Face Vertex

Accuracy of the face intersection is, however, essential if intersections and boundaries within an object are to be free of undesirable visual artifacts. To achieve this accuracy, the latter two methods above would require relatively small tetrahedra. This can be accomplished by uniformly small tetrahedra, which would produce an excessive number of polygons, or by adaptive subdivision, which is difficult to implement. Therefore, we support a face vertex.

4.4.2 Multiple Edge Vertices

Three regions can intersect a cell face other than as shown in figure 4.10. Two common possibilities, shown below, left, suggest the need for multiple edge vertices along a cell edge, as shown below, right. In both examples, multiple edge intersections occur along an edge connecting equi-valued corners. Thus, unlike conventional polygonizers, our implementation must inspect all edges, not simply those that connect differently valued corners.

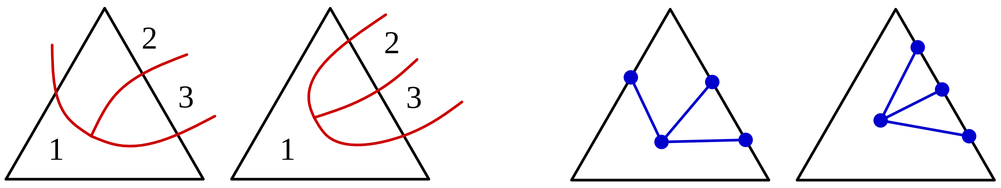


Figure 4.12 Need for Multiple Vertices along an Edge

4.4.3 Disjoint Lines and Surfaces

Let us consider the face shown above, left, its containing tetrahedron, and an adjoining tetrahedron. The face is repeated below, left, and the two tetrahedra are shown below, middle, separated for clarity. The front face of the adjoining tetrahedron is shown at right. The arrangement of polygons produced from these tetrahedra is significantly more complex than the single polygon produced by conventional polygonizers. In particular, the left tetrahedron contains a 3, a 4, and a 5-sided polygon, all sharing a common edge. The right tetrahedron contains *disjoint surfaces*. Consequently, a single face must accommodate multiple disjoint lines, each of which connects an edge vertex separating a pair of regions to another edge vertex separating the same pair of regions. This arrangement of face and edge vertices can occur regardless of cell size.

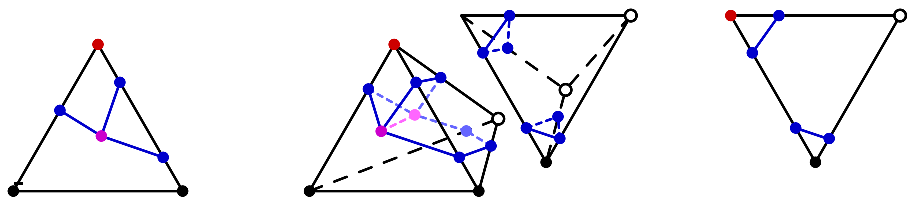


Figure 4.13 Disjoint Lines and Surfaces

adjoining tetrahedra and their front faces

Our implementation accommodates any number of disjoint lines within a face, provided there is no face intersection.

4.4.4 Complex Face Topologies

A sufficiently complex object can require an arbitrary number of face and edge vertices within a single polygonizing cell. For example, a face might contain any arrangement shown below, regardless of cell size. These complex face topologies are distinguished by multiple face intersections or by disconnected components, at least one of which contains a face intersection. A robust polygonizer should handle these cases. We have, however, restricted our implementation to one face vertex per face or a collection of disconnected components, none of which contains a face intersection. Despite these simplifications, our implementation produces satisfactory results for the examples presented in section 4.8.

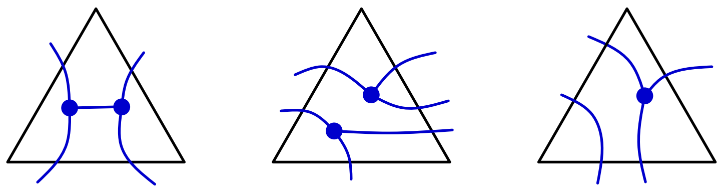


Figure 4.14 Complex Face Topologies

4.4.5 Intersection Curves

If each region of space is closed, then a surface separating two regions is the intersection of the two. Similarly, a one-dimensional curve is the intersection of three or more regions of space. For example, referring to figure 4.9, the circle in the sphere-square is the intersection of regions 1, 3, and 4, and the boundary of the square is the intersection of regions 2, 3, and 4.

These curves intersect the polygonizing cell at a cell face, and are readily approximated simply by connecting face vertices. Thus, the non-manifold polygonizer can approximate intersection curves of two or more surfaces. This approach enforces a singular point of intersection (below, middle), whereas surface/surface intersection methods prone to numerical instability can yield non-intersecting curves (below, right).

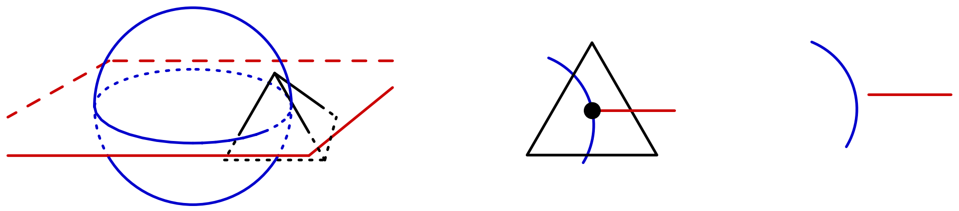


Figure 4.15 Intersection Curves

left: circular intersection of sphere and plane passes through tetrahedral face

middle: polygonization is guaranteed to approximate curve/curve intersection

right: curve following may not find curve/curve intersection

4.5 Implementation

In this section we present details for cell propagation, cell polygonization, and post processes that modify the set of vertices generated by the polygonizer. Throughout, we intend that the polygonizer produce vertices whose locations are independent of the region-pairs of interest to the client. That is, if the client interest changes, the location of remaining vertices should not. Pseudo-code detailing these algorithms is presented in [Bloomenthal and Ferguson 1994].

4.5.1 Cell Propagation and Decomposition

Propagation is the process whereby the polygonizer generates a set of cells that enclose the surface. As with conventional polygonizers that utilize continuation, we begin with a start point, usually supplied by the client. With conventional polygonization, a cell propagates to other cells across faces that contain both positively and negatively valued corners. With the non-manifold polygonizer, however, a face must contain a polygon of interest to the client. This prevents unwanted propagation along uninteresting surfaces, such as the cube in the sphere-square example.

Although the cube is a convenient propagating cell, we prefer the tetrahedron as a polygonizing cell because it simplifies the task of locating face vertices and disjoint surfaces. Therefore, as with some conventional polygonizers, the non-manifold polygonizer decomposes each cube into five tetrahedra, as shown below. The five-decomposition must alternate its orientation from one cube to a neighboring cube so that faces of neighboring tetrahedra coincide (a six-decomposition of the cube, shown in figure 3.6, does not require alternating orientations).

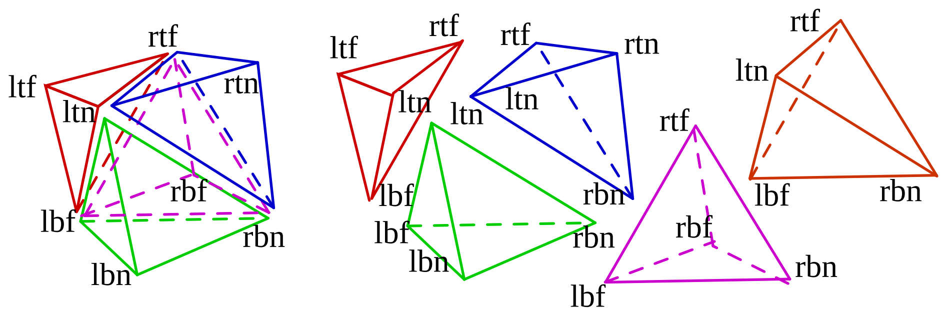


Figure 4.16 The Five-Decomposition of a Cube

l, r, t, b, n, f mean, respectively, left, right, top, bottom, near, far

4.5.2 Cell Polygonization

Given an individual tetrahedron, polygonization creates those polygons that approximate the surfaces within the tetrahedron. This process consists of a) checking each tetrahedral edge for edge vertices, b) checking each face for a face vertex, c) calculating any necessary internal vertex, and d) producing the polygons.

4.5.2.1 Edge Vertex Calculation and Storage

As discussed in section 4.4.2, we place an edge vertex at all edge intersections along a cell edge; also, with each edge vertex we associate the region-pair that it separates. The binary subdivision frequently used in conventional polygonizers does not work well here, as in the first subdivision step one half of the edge is ignored and intervening intersections missed, as shown below.

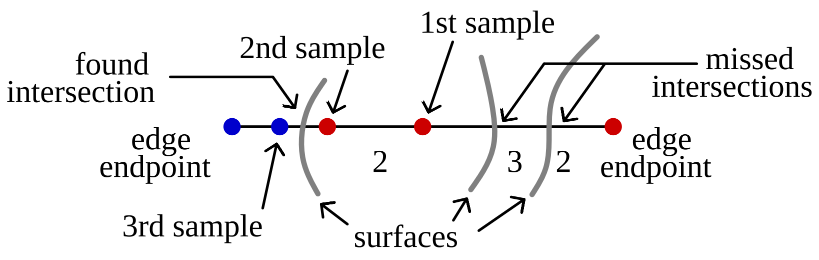


Figure 4.17 Sampled Points by Binary Subdivision

Therefore, we first divide the edge into several equally sized sections and then apply binary subdivision to each section whose endpoints have different region values. If, during this binary subdivision, a 'foreign' region value is encountered (*i.e.*, a region value not equal to either of the section endpoint region values), then the subdivision continues recursively in both halves, detecting two (or more) intersections in the given section. Surface vertices are then placed at the midpoint of each final interval. To guard against a very narrow region crossing a final interval, we test that the region value at the interval midpoint is one of the two endpoint region values of the interval; if not, the subdivision is continued. We choose *n*, the number of initial sections, and *m*, the minimum number of subdivision steps, such that each vertex will be within ε of an actual intersection, *i.e.*, such that $edgeLength/(2^{(m+1)}n) \leq \varepsilon$.

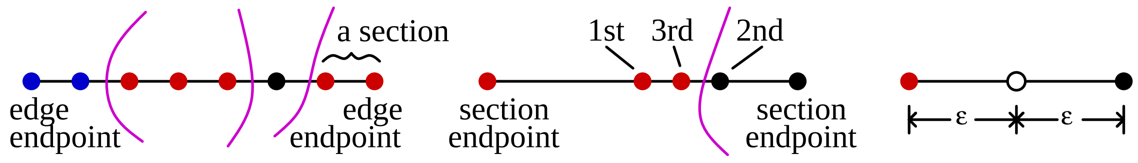


Figure 4.18 Two-Stage Edge Division for a Non-Manifold Polygonizer
left to right: initial division of edge into sections, binary subdivision within one section, vertex placed at middle of final interval

We utilize a hash table, as described in section 3.5.2, to associate surface vertices with tetrahedral edges. As we explain later, it is important to order the edge vertices during their storage; we use the location of the edge end-points to compute an edge direction, and the vertices are ordered accordingly. The region-pair for each vertex is also ordered according to this edge direction.

We store all edge vertices, whether or not they belong to a surface of interest. This allows subsequent parts of the polygonizer to produce geometry that is independent of client interest. Storing all edge vertices also permits computation of the region-set for a face vertex, as described below.

*4.5.2.2 Looped Edge Intersec*tions

A face contour is ‘looped’ if it enters and exits a face along the same edge, as shown below, left. This example results in two adjacent looped edge intersections, on the same edge, having the same but oppositely ordered region-pairs (*e.g.,* (1,2) and (2,1)). As shown below, right, looped intersections may occur on edges of equal or differently valued corners, and may be nested.

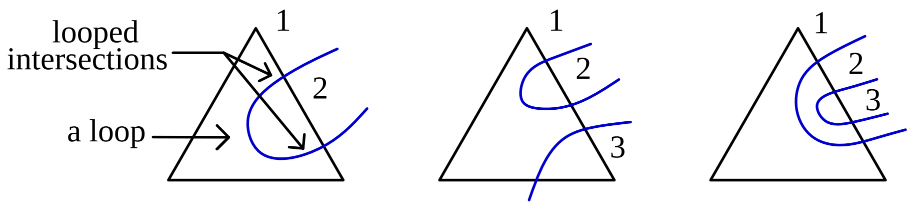


Figure 4.19 Loop Examples

Looped intersections are problematic because there is no face intersection with which they can connect. We briefly consider these options for a pair of looped intersections: a) they may be joined by a face line, b) they may be connected to a point on the face contour, preferably one that is maximally distant from the edge, or c) they may be ignored. The first option is undesirable because the face line would lie along the cell edge; this can produce duplicated polygons or polygons that align with tetrahedral edges or faces, resulting in a staircased tessellation. The second option is an implementation complication we preferred to avoid.

Therefore, we ignore looped intersections, effectively truncating the surface, as illustrated below. Conventional polygonizers also truncate loops if they occur between equi-valued cell corners. As shown, truncation can be arbitrarily large. If significant, these errors are best remedied with a smaller cell size. We regard face vertex accuracy (section 4.4.1) as more important than accommodating loops because the occurrence of non-manifold edges is more likely (within a non-manifold context) than the occurrence of elongated protuberances.

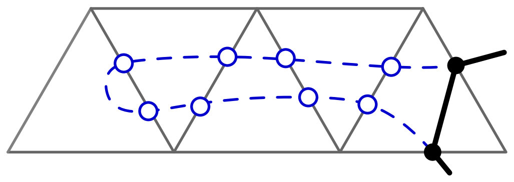


Figure 4.20 Truncation of Large Loop

4.5.2.3 Face Vertex Calculation

Our implementation permits a single face intersection per face. Because face vertices indicate the intersection of three or more regions, the limitation of one face vertex per face requires the designer to separate region intersections by the size of the polygonizing tetrahedron.

As with edge vertices, face vertices are shared between adjoining tetrahedra. Therefore, we first test if a face has been previously processed. If not, we then determine if it contains an intersection; a face contains at least one intersection if at least one of its face lines is not disjoint. To determine if a face contains only disjoint lines, and no face intersection, we traverse its edge vertices in order, adding and removing vertex region-value pairs from a stack. If there are only disjoint lines in the face, then, beginning with a vertex *startV*, the stack will become empty when the partner of *startV* is reached, then begin to fill again, and will empty a second time upon the return to *startV*.

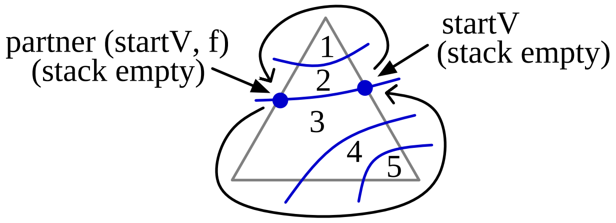


Figure 4.21 Determining if a Face Contains only Disjoint Lines

If an intersection exists, a face vertex is created and stored. Because it is computationally expensive to locate the position of a face intersection, we set its location only if it represents an intersection with a surface of interest.

To compute a face intersection, we follow the face contour beginning at an edge vertex *v* and continue until we find a foreign region (*i.e.*, a region other than the two separated by *v*). This is similar to other local methods [Bajaj *et al.* 1988], [Mortensen 1985]. As shown below, the face contour is surrounded by small triangles until a region other than 1 or 2 (in this example) is encountered. The face vertex location is then set to the center of the final triangle.

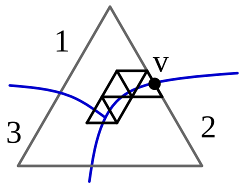


Figure 4.22 Following a Face Contour

(the small triangles are enlarged for illustration)

For simple face topologies, such as above, the choice of start vertex is immaterial. For complex topologies, some vertices, such as *v1* and *v2* below, do not yield a face intersection. The contour follower must recognize this, and begin again at another edge vertex. Also, some vertices, such as *v3* and *v4*, yield different intersections; therefore we begin searches first from edge vertices separating regions of interest. This compromises our goal of a geometry independent of client interest, but is reasonable given our limitation of a single face vertex.

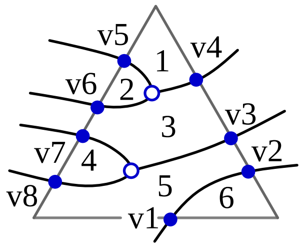


Figure 4.23 A Complex Face Topology

The small triangles enclosing the face contour are each defined by a directed edge crossing the contour, as shown below; an initial directed edge spans the start vertex. Each directed edge implies a new triangle apex, whose region value determines which triangle side becomes the next directed edge. The final triangle is one whose apex belongs to a foreign region; the center of this triangle is used as the face vertex location. To ensure this is within ε of the actual intersection, the triangle side length must not exceed (2√3)ε, assuming the actual intersection is contained within the final triangle. A recursive contour follower that contains the contour within increasingly smaller triangles may be more efficient and accurate.

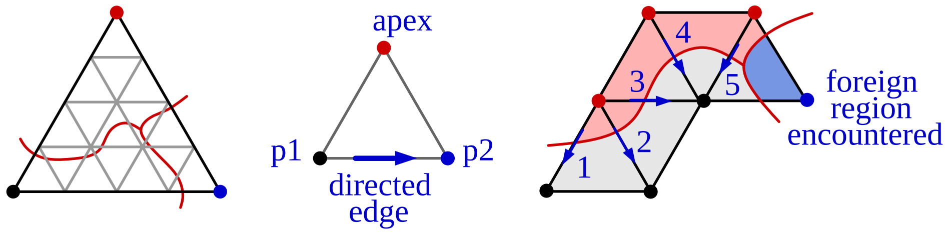


Figure 4.24 Face Contour Following using Directed Edges

4.5.2.4 Polygon Formation and Internal Vertices

When no face vertices are needed for regions of interest, those edge vertices needed for regions of interest are connected to form one or more disjoint polygons, as shown below, left. All such polygons will have 3 or 4 sides (this claim is established in [Bloomenthal and Ferguson 1994]).

Polygon formation begins with an edge vertex separating regions of interest and a tetrahedral face containing the vertex. This first vertex is stored as a member of the polygon. We now proceed to the partner of this vertex, with respect to the given face (see figure 4.21). The 'current' vertex becomes the partner, and the 'current' face becomes the face on the other side of the edge containing the new current vertex. This process, similar to ones described in [Bloomenthal 1988] and [Wyvill and Jevans 1993], is iterated until the current vertex arrives at the start vertex. An optimized procedure could separately process common cases, such as those typical of conventional polygonization (see section 3.5.3).

So that the client can associate one side of a polygon with a particular region value (in order, for example, to determine polygon color or whether the polygon is front-facing), the polygons must be constructed with a consistent orientation. We order each polygon so that, when viewed from the lesser of the two regions it separates, its vertices appear in counter-clockwise order.

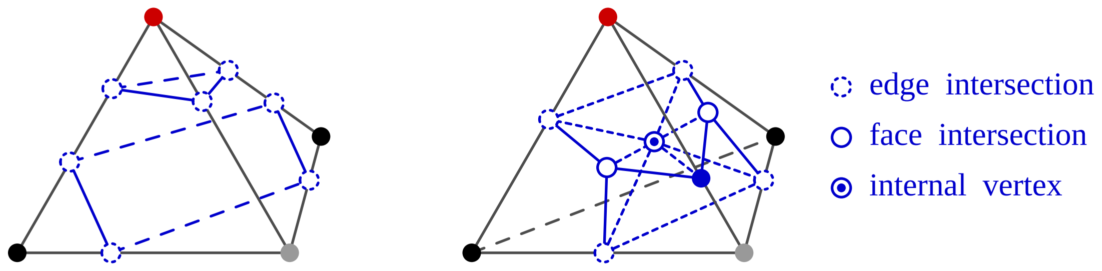


Figure 4.25 Polygon Formation

left: disjoint, right: non-disjoint

In the case of non-disjoint surfaces, there must be at least two face vertices that separate regions of interest (this claim is substantiated in [Bloomenthal and Ferguson 1994]. We create a vertex *internal* to the tetrahedron whose location is the average location of those face vertices that separate regions of interest. Our implementation is restricted to a single internal vertex. Each face line, together with this internal vertex, creates one triangle, as shown above, right. Each triangle is ordered counter-clockwise when viewed from the lesser-valued of the two regions it separates. The triangles produced are all adjoining.

Triangles are returned to the client of the polygonizer via a 'callback' procedure. Arguments to the procedure include a pointer to the (ever-growing) array of vertices, three integers that index the vertex array and represent the current triangle, and a pair of integers that represents the region-pair separated by the current triangle.

4.5.3 Post Processes

There are two minor but significant problems whose remedy we postpone until all cells have been polygonized. These are the problem of undesirable triangles, which can occur in conventional polygonization, and the need for duplicated vertices, which is specific to non-manifold polygonization.

4.5.3.1 Thin or Small Triangles

Conventional polygonizers produce *thin* triangles (*i.e.*, triangles with appreciable height but minimal width) if an edge of the polygonizing cell is nearly tangential to the implicit surface. Polygonizers also produce *small* triangles (*i.e.*, triangles with minimal height and width) if a corner of the polygonizing cell is close to the surface. These cases occur with the non-manifold polygonizer as well; additionally, thin or small triangles can occur when an internal vertex is close to a face vertex, or when a face vertex is close to an edge vertex.

Such triangles can cause visualization artifacts. They are also prone to large orientation errors, (*i.e.*, the normal of the triangle can vary significantly from the actual surface normal) if the length of the shortest edge of the triangle is comparable to, or less than, ε, the accuracy to which edge and face intersections are computed.

We explored the possibility of eliminating, during polygonization, those vertices located within *close-factor* $\times$ ε of another vertex (*close-factor* is defaulted to 1 or provided by the client). This allowed a complex cascade of identification (*e.g.*, an internal vertex could be moved onto a face vertex, a face vertex could be moved onto an edge vertex, and an edge vertex could be moved onto a cell corner). An alternate approach suggested in [Moore and Warren 1991] perturbs cell corners so that thin or small triangles do not result.

To simplify our implementation, we chose to separate the process of removing thin or small triangles from the process of polygonization. Thus, we include a post processing stage in which each pair of close vertices, connected by a triangle edge, is replaced by the average of the two vertices. Also, any triangle vertex close to the opposite triangle edge is replaced by the midpoint of the edge. Any degenerate triangles that result are removed. The vertex modifications are illustrated below.

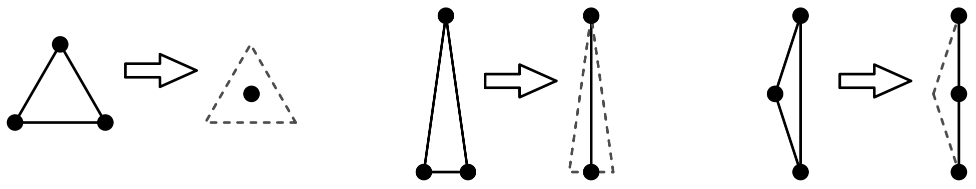


Figure 4.26 Small and Thin Triangles

4.5.3.2 Duplicated Vertices

Consider three triangles, t_1, t_2, and t_3, that share an edge along the intersection circle of the sphere-square, as shown below, left. For smooth shading, each vertex requires a surface normal, but, in this example, it is not possible to compute a single normal for ***v***, a vertex shared by all three triangles. This is because, as shown below, right, t_1 and t_2 require a right-facing normal, whereas t_3 requires an upward-facing normal. To accommodate these competing requirements, we produce coincident vertices located at ***v***, (and ***vv***) one for each different region-pair of the polygons that share ***v***. Consequently, each vertex located at ***v*** may have an independent surface normal, color, and any other property associated with the surface that separates the region-pair of the vertex.

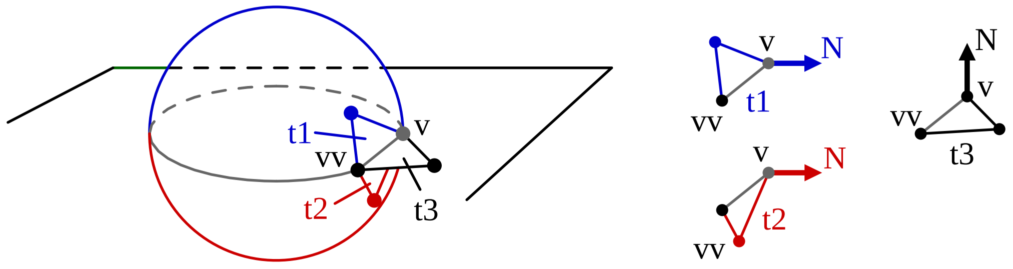


Figure 4.27 Need for Duplicated Vertices

4.6 Calculation of Surface Normals

Although a surface normal for a vertex can be approximated by the averaged normals of its surrounding polygons, conventional polygonizers usually provide a more accurate calculation by approximating the gradient, ∇f, as discussed in section 3.5.3. Our region function, f_{reg}, is, however, integer valued and cannot yield a gradient. Thus, in addition to f_{reg}, we allow the client to provide a real-valued function g from which the gradient can be calculated by the polygonizer. As discussed in section 4.5.3.2, a vertex normal depends not only on its location but also on the region-pair it separates. Therefore, g requires a point in space and a region-pair. For a fixed region-pair, g should be a continuous function in the neighborhood of the surface separating that region-pair. As demonstrated in section 4.8.1, g usually can be defined in terms of those functions that underlie f_{reg}.

4.7 Performance

We first compare the performance of our polygonizer with that of a conventional polygonizer; we then present several polygonized, non-manifold objects.

For non-trivial objects, both conventional and non-manifold polygonizers devote the majority of their time to evaluation of the implicit surface function (f or f_{reg}). Thus, we compare the two polygonizers according to number of their function evaluations. The non-manifold polygonizer performs many more evaluations along a surface border, where it must locate a face intersection. Because a conventional polygonizer does not attempt to locate such intersection, we compare the frequency of evaluations for solid models only. This limits our consideration to the typical cases of three and four edge intersections per tetrahedron.

We assume the non-manifold polygonizer uses n (the number of sections on an edge, see section 4.5.2.1) = 16 and m (the number of steps in the binary subdivision) = 4, and that the conventional polygonizer uses m = 8, so that their surface vertex accuracies are equal. We ignore function evaluations for corners of the tetrahedra. which should be the same for both polygonizers. For three edge intersections, the conventional polygonizer requires (3×8) = 24 evaluations of f, whereas the non-manifold polygonizer requires $3\times16+3(16+4)$ = 108 evaluations of f_{reg}. For the four edge intersections case, the conventional polygonizer requires (4×8) = 32 evaluations, and the non-manifold polygonizer requires $2\times16+4(16+4)$ = 112 evaluations. So, as a rough estimate, the non-manifold polygonizer requires four times as many function evaluations as does the conventional polygonizer.

4.8 Examples

For each example given in this section, the designer provided the following parameters and callback functions, although some were defaulted. The symbol Z indicates the set of integers.

Region ($\boldsymbol{p} \in \Re^3$) returns *regionValue* $\in Z$

This is f_{reg}, which returns a single value for a given point $\boldsymbol{p}$ (despite our description in section 4.4.5 of a surface as an intersection, implying that $\boldsymbol{p}$ can belong to two regions).

Interest (*region-pair* $\in (Z, Z)$) returns Boolean

This determines whether a client is interested in a particular surface; it could also return miscellaneous polygon properties, such as color.

Gradient (*region-pair* $\in (Z, Z)$, $\boldsymbol{p} \in \Re^3$) returns $\Re$

This procedure is required to assign an accurate normal to a surface vertex.

propagating cell size: $\Re$

The client provides a size small enough to obtain an acceptable level of detail yet large enough to avoid an excessive number of polygons.

start point: $\Re^3$

The distance from *startpoint* to the surface should be less than one-half the size of the propagating cell. If not provided, the polygonizer can perform a random search for a start point.

ε: $\Re$

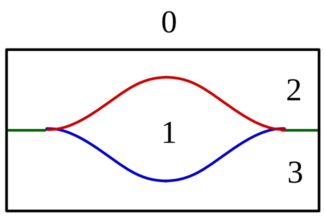
This is the accuracy to which edge and face intersections are located; we have found 1/256 of the propagating cell size to be acceptable.

close-factor: $\Re$

When collapsing thin or small triangles, two vertices are considered close if they are within *close-factor* $\times\, \varepsilon$. We have used a *close-factor* of 1.

4.8.1 A Saucer and a Square

This first example is topologically equivalent to the sphere-square discussed in section 4.2. To produce a more interesting object, however, the sphere function was scaled by distance to the plane, producing a saucer-shaped blend to the plane. For this example, we provide pseudo-code for the region value, f_{reg}, surface interest, and the gradient.



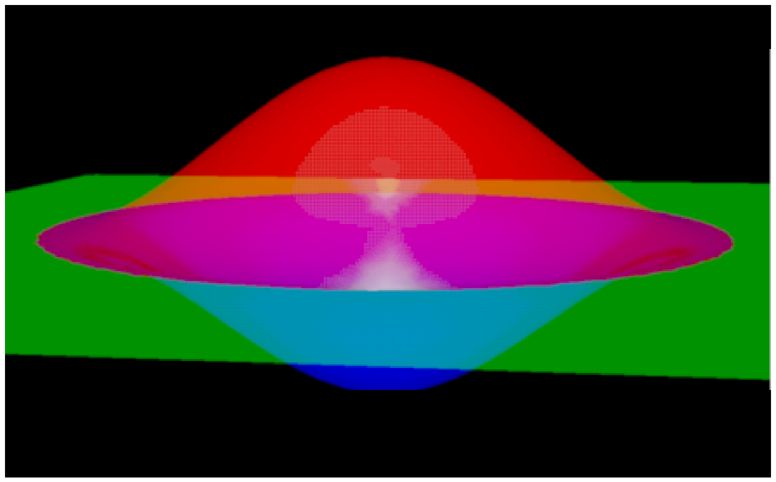


Figure 4.28 Saucer Shape

left: cross-section of regions

right: rendered with transparency

```
Sigmoid (d) {
    (a blend function from [Wyvill et al. 1986])
    if abs (d) > 1
                then return 0
                else return $1-(4d^6-17d^4+22d^2)/9$
}
```

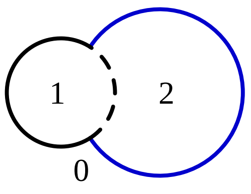
```
Saucer (p) { return .35 Sigmoid ($1.4\sqrt{\mathbf{p}_x^2+\mathbf{p}_y^2}$) }
```

```
Region (p) {
    if not InsideCube (p) then return 0
    if Saucer (p) > abs ($p_Z$) then return 1
    if $p_Z$ > 0 then return 2 else return 3
    }

Interest (region-pair) {
    switch region-pair
            (1, 2) => return (true, red)
            (1, 3) => return (true, blue)
            (2, 3) => return (true, green)
            return (false)
    }

Gradient (p, region-pair) {
    switch region-pair
            (1, 2) => return $-p_Z-$Saucer (p)
            (1, 3) => return $p_Z-$Saucer (p)
            (2, 3) => return $-p_Z$
    }
```

4.8.2 Sphere with Hole

This simple example defines a sphere with a disk removed, as shown below. The simple Boolean set operation 2–1 (2 being the large sphere and 1 being the small sphere) produces a solid sphere with a ‘scoop’ removed. This is easily reproduced by our polygonizer by adding (1, 2) to the surface interest list, but this is not our desired surface. In particular, the sphere-hole has less surface area than the large sphere. The CSG surface will have an *increased* area (because the smaller sphere has higher curvature than the large sphere). CSG, a *solid* modeling system, cannot reproduce the sphere-hole surface.

Although this surface is easily constructed with parametric techniques, a simple extension, given in the next section, would be substantially more difficult to accommodate parametrically.



```
*Region* (***p***)
    if ***p*** inside small sphere then 1
    if ***p*** inside large sphere then 2
    else 0
```

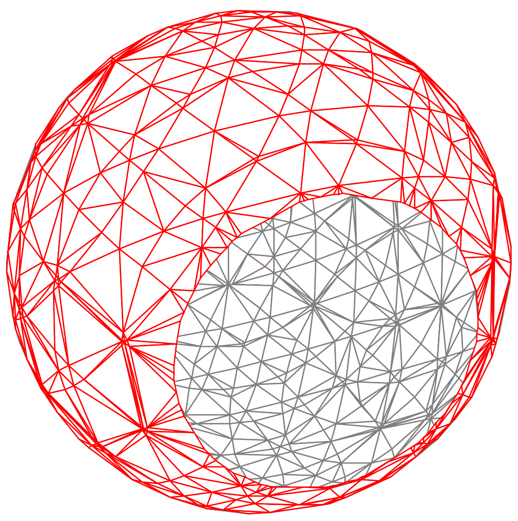
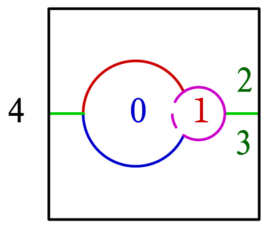


Figure 4.29 Sphere/Hole

left: region definition, right: surface (0, 2) (grey inside, red outside)

4.8.3 Sphere and Plane with Hole

This example combines the ideas of the previous two examples; it consists of a sphere and a square with a sphere removed in the vicinity of their intersection. Here we indicate interest in some intersection curves, which can be seen as thin lines (blue, yellow, cyan, and magenta) in the polygonized results below.



$$\begin{aligned}\text{surfaces of interest} &= \{(0, 2), (0, 3), (2, 3)\}\\ \text{curves of interest} &= \{(0, 1, 2), (0, 1, 3),\\ &\qquad (1, 2, 3), (0, 2, 3), (2, 3, 4)\}\end{aligned}$$

Figure 4.30 Definition for Sphere/Plane/Hole

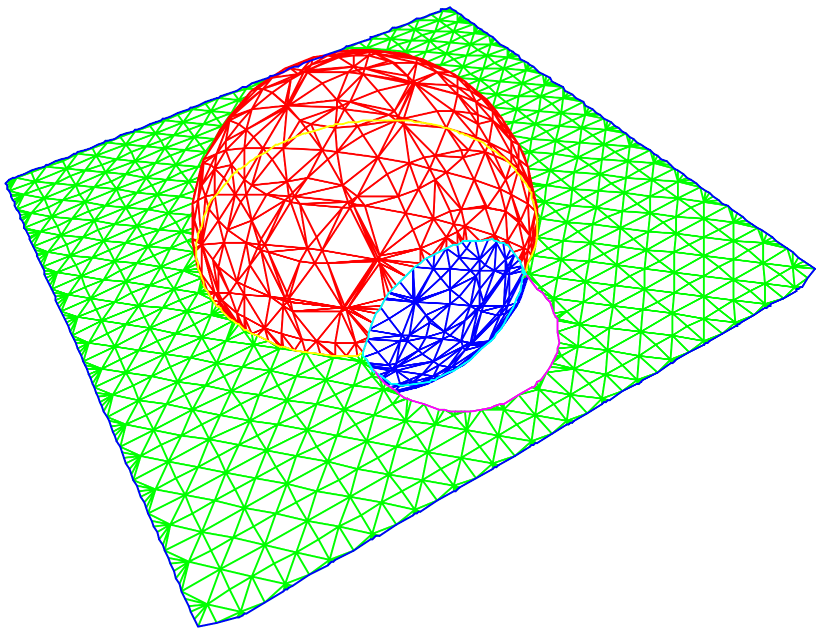


Figure 4.31 Surfaces and Intersection Curves for Sphere/Plane/Hole

4.8.4 Blend to a Patch

A variation on the saucer is a blend between sphere and bicubic (tensor product) patch. The definition utilizes distance from ***p*** to the nearest point ***n*** on patch ***P***, with ***n*** = ***P*** (*s*, *t*). An initial *s* and *t* are determined from an approximating triangle mesh (discussed in section 7.2) and are refined by projecting the vector from ***n*** to ***p*** onto the tangent plane at ***n***, as shown below. For this example, the mesh contained 98 triangles, and few iterations were required for Δ*s* and Δ*t* to fall below .001.

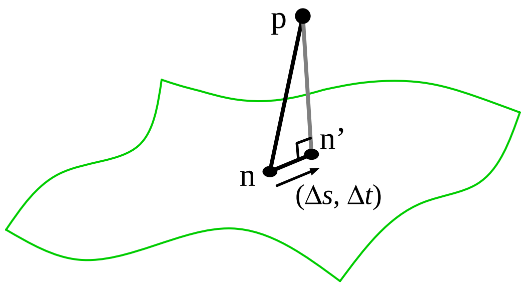


Figure 4.32 Determining a New s and t

The region definition for the Sphere/Patch given by:

```
if Beyond (p, patch) then return 0
if Sigmoid (p) > abs (DistanceToPatch (p, patch)) then return 1
if DistanceToPatch (point, patch) > 0 then return 2 else return 3,
```

where *DistanceToPatch* is signed distance from ***n*** to ***p***, and ***p*** is considered beyond the patch if either *s* or *t* were less than .001 or greater than .999. This definition is illustrated below, followed by surface results.

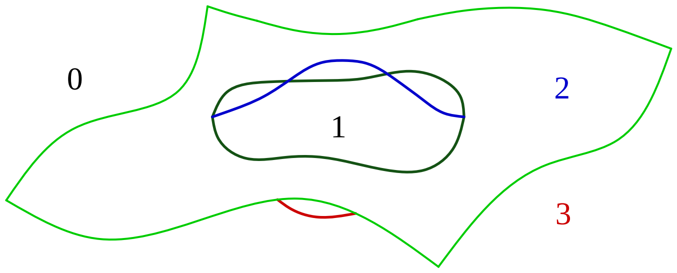


Figure 4.33 Region Definition for Sphere/Patch

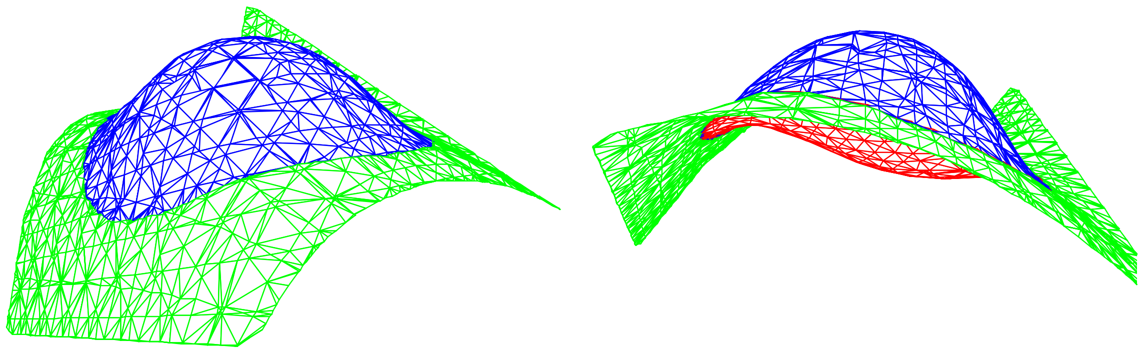


Figure 4.34 Line Drawings

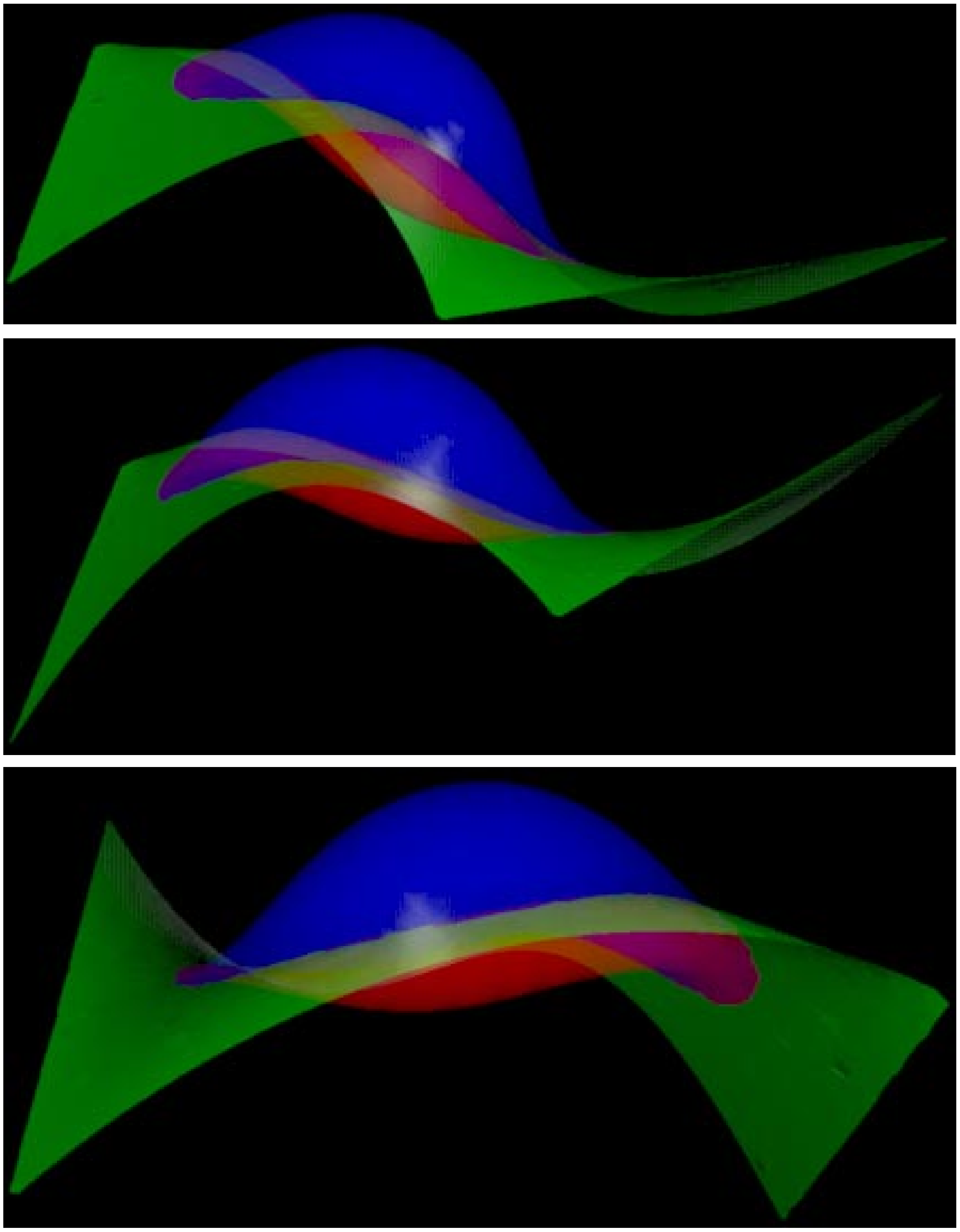


Figure 4.35 Shaded Images (with transparency)

4.9 Conclusions

In this chapter a new, simple, and effective method was presented to express and polygonize non-manifold implicit surfaces. Although the current implementation has limitations, it expands the range of surfaces normally employed in computer graphics. For example, it permits a simple expression and evaluation for parametric surfaces combined with and trimmed against implicit volumes. These non-manifold implicit surfaces are defined by multiple regions of space, unlike the binary regions that typically define implicit surfaces. When polygonized, non-manifold surfaces may have borders (*i.e.*, edges of degree 1) and intersections (*i.e.*, edges of degree 3 or more). The object is defined by a list of region pairs whose separating surfaces are of interest.

This simple method significantly complicates the polygonizer. In particular, the accurate polygonization of surface borders requires vertices internal to faces of the polygonizing cell, which in turn require multiple intersections per cell edge and the examination of all edges. This in turn leads to the need for multiple, disjoint surfaces. Moreover, the accurate polygonization of surface intersections requires vertices fully internal to the polygonizing cell. This additional complexity, however, is exercised only in those relatively few tetrahedra that contain borders or intersections of surfaces.

The present polygonizer does not accommodate an arbitrarily complex intersection of surface with polygonizing cell. It does, however, accommodate an arbitrary number of disjoint surfaces (limited, however, by the maximum number of edge vertices, as described in section 4.5.2.1). Non-disjoint surfaces are limited to a single intersection per face and a single internal vertex per polygonizing cell. The accommodation, within a single polygonizing cell, of disjoint surfaces and multiple face and multiple internal intersections could be the subject of future work.[3] Other areas of future study include improved boundaries and reduced truncation. In particular, although the accuracy of our face contour follower produces good approximations to smooth boundary edges, it performs poorly for sharp bends in the boundary; a corner, for example, becomes beveled. This can be mitigated by improving the accuracy of the internal vertex location and allowing multiple face intersections per face. The non-manifold polygonizer is far from ideal, and we have discussed aspects of its limited accuracy. Nonetheless, the approach has substantial potential, which we hope is demonstrated by the examples.

Ray-tracing is eminently able to render non-manifold, implicitly defined shapes. Most ray-tracers return an identifier for a particular object or region of space, usually as a means to accelerate performance or invoke anti-aliasing. It would be instructive to compare polygon rendered images (at both moderate and high polygon resolutions) with line drawings and ray-traced images of non-manifolds.

Within the context of design, however, we prefer the ability to produce a concrete instantiation of our object. The power of this method of shape specification and the extent to which an editor allows ready modification of shapes remain to be determined.

4.10 Notes

1. In keeping with the theme of skeletal design, we may regard the non-trimmed part of the parametric surface as an *exo-skeleton*.

2. The insight that a multi-regionalization of space was a conceptually elegant solution to the problems associated with binary regions is due to Keith Ferguson.

3. Perhaps this can be accomplished by first decomposing the surfaces within the cell into disconnected components, and then polygonizing each component separately. Correctly accommodating arbitrary complexity will eliminate the compromises we have made concerning a vertex geometry that is independent of client interest. Alternatively, rather than accommodate arbitrary complexity within each tetrahedra, our present restriction of a single vertex per face could be

combined with adaptive subdivision techniques. For example, for surfaces with high curvature within a tetrahedron or for a tetrahedron with more than four impinging regions, a tetrahedron can be subdivided into four similar tetrahedra [Moore 1991].

The clearest way to the Universe
is through a forest.
(John Muir)