An Interval Refinement Technique for Surface Intersection

Michael Gleicher
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
gleicher@cs.cmu.edu

Michael Kass
Apple Computer
20525 Mariani Ave.
Cupertino, CA 95014
kass@apple.com

Abstract

This paper describes a technique for computing the intersections of two parametric surfaces based on interval arithmetic. The algorithm, which can be stopped and restarted at any point, uses search techniques to refine its description of the intersections progressively. Interval arithmetic provides guaranteed points on the intersection curves to within a user-specified tolerance. These points are connected into polygons and used to triangulate the trimmed surfaces. We provide details of an implementation and give examples of the algorithm's use.

Résumé

Cet article décrit une technique pour calculer les intersections de deux surface paramétriques avec l'intervalle arithmétique. L'algorithme utilise des techniques de recherche pour augmenter la précision des intersections, et peut être interrompu ou redemarrer à n'importe quel moment. L'intervalle arithmétique nous donne des points garantis sur les courbes d'intersections. Nous créons des polygons avec les points et triangulons les regions interieurs. Nous exposon en detail un modèle de mise en oeuvre et nous donnons des exemples de l'utilisation de l'algorithme.

Keywords: Surface Intersection, Interval Arithmetic

1 Introduction

Finding the intersection of two parametric surfaces is an important problem in Computer Aided Geometric Design. It is useful in many applications such as trimming surfaces and performing boolean operations on boundary representation geometric models[17]. The difficulty of this problem forces solutions to trade generality, robustness, and performance.

Here, we present a method for intersecting parametric surfaces based on interval arithmetic. The method is very general, placing few restriction on the class of surfaces it can handle. Nonetheless, the intersection points it finds are are guaranteed to be within a specified tolerance. Since the method continually refines its results, a valid estimate of the intersection is always available during execution. As a consequence, the algorithm can be stopped when tolerance criteria or time bounds are met and restarted if the results are unacceptable. Adjusting the tolerances makes it posible to trade accuracy or sampling rate for computation time.

In sections two and three, we briefly review previous work on the intersection problem and describe the basics of interval arithmetic. Then section four describes the algorithm for finding intersection points, formulating the the task as a search problem. Finally, section five addresses the issue of linking the intersection points together and triangulating the bounded regions of parameter space. Results from our prototype implementation are presented for a variety of shapes.

2 Related Work

Surface intersection problems have been widely studied because of their practical importance (see [16] or [12] for a survey). The general problem is to find the set of points where two surfaces coincide in space. While two surfaces typically intersect at a set of space curves, the intersection may also contain distinct points or surface elements in degenerate cases.

Parametric representations define surfaces by maps from the plane to three-dimensional Euclidean space. These representations are extremely popular because of their convenience for a variety of modeling and rendering purposes. Unfortunately, parametric surfaces are very difficult to intersect[16].

In general, exact analytical solutions for surface inter-

section problems are unavailable or impractical, since even simple surfaces can meet at very complicated curves[11]. As a consequence practical solutions to the intersection problems must resort to approximating the solution. Following Barnhill et. al.[4], we characterize these approximations by tolerances which specify how closely the approximation must match the actual intersection.

The literature on surface intersections contains a wide variety of approaches. The most common are marching and subdivision. Marching methods (e.g. [3, 14, 12, 5, 4, 6]), begin with points known to be at the intersection of the two surfaces and use numerical techniques to compute successive points on the intersection curve. In addition to the numerical challenges of progressing around the curve, these techniques face the additional task of finding the initial points from which to begin marching.

Subdivision is another approach to finding the intersection of two surfaces. The basic idea (e.g. [16, 9, 7]) is to divide the surface intersection problem into smaller pieces until each piece is a solvable problem. For example, Houghten et. al. [9] subdivide surfaces until each piece is nearly planar and use the fact that intersections of planar elements can be calculated directly.

One difficulty with subdivision approaches is that they require a way of deciding whether the subproblems are adequately modeled by the solvable problems, for example deciding if a surface segment is nearly planar [9]. A second difficulty is that the results are often not guaranteed at all, or only guaranteed for a very restricted class of surfaces. For example, some powerful recent results (e.g. [19, 18]) apply only to polynomial or rational functions. Even if the results can be guaranteed, many subdivision techniques have poor performance [19] because they need to search exhaustively for intersections.

Our approach has several advantages over typical subdivision approaches. The use of interval arithmetic permits us to make guarantees about finding points on the intersection curve without placing severe restrictions on the class of surfaces the algorithm can handle. The parametric mappings can be expressed in terms of trigonometric functions, for example, with no particular difficulty. In addition, the subdivision strategy we use avoids exhaustive search in most cases. By adjusting the tolerances used for stopping conditions, the tradeoff between time and accuracy can be user controlled.

Interval arithmetic has been used for a variety of purposes in computer graphics[15, 13, 1, 20, 21]. Mudar and Koparkar [15] present the basic idea of using interval arithmetic to identify surface intersections but make no mention of the issues involved in creating efficient reliable algorithms which provide descriptions of intersection curves within user specified tolerances. The work of Von Herzen and Barr[22] is very similar, using Lipschitz conditions to evaluate bounding regions for portions of surfaces. The Lipschitz conditions are derived by hand for each new analytic surface, unlike the automatic interval arithmetic used here. Von Herzen and Barr also do not address the issue of finding the intersections themselves, instead relying on implicit functions for breaking objects into pieces.

3 Interval Arithmetic

Interval arithmetic is a method for providing a bound on the output of a function given bounds on all of its inputs. This section provides a brief introduction to interval arithmetic and its relevance to the surface/surface intersection problem.

Interval arithmetic is based on the idea of extending ordinary scalar operations to intervals on the real line. If S is an interval, we can write it as (S_{min}, S_{max}) to denote a quantity whose value lies somewhere between S_{min} and S_{max}. With every ordinary scalar function, say $f(s,t)$, we associate an interval function $F(S,T)$, which provides a bound on $f(s,t)$ given bounds on s and t. We begin by defining the interval functions corresponding to primitive operations (e.g. basic arithmetic operations and trigonometric functions). For example, if $f(s,t) = s + t$, we can define $F(S,T)$ to be the interval $(S_{min} + T_{min}, S_{max} + T_{max})$. Clearly, if s and t are within their bounds, their sum must lie in the interval $F(S,T)$. Similar rules can be developed for a wide variety of elementary functions[2].

Once we have defined a set of interval functions corresponding to primitive operations, we can create more complicated interval functions by composing them. The interval function corresponding to $f(g(q,r), h(s,t))$, for example, is simply $F(G(Q,R), H(S,T))$. We have automated this operation by defining a set of operations on intervals using the operator overloading capabilities of C++.

Parametric surfaces are defined by mappings from (u,v) to (x,y,z). If we use interval arithmetic to represent the mapping, then we have an interval function which maps from (U,V) to (X,Y,Z). The interval mapping provides an axis-aligned bounding box in world space for every rectangular region of parameter space. The bounding box may not be a tight bound, but we are guaranteed that it contains the piece of the parametric surface defined by the rectangular region of parameter space.

4 An Interval Approach to Surface Intersection

Suppose that we have two parametric surfaces and their corresponding interval functions. If we pick a rectangle in each parameter space, the interval functions provide a pair of bounding boxes, one for each surface. We make use of the bounding boxes as follows. If the bounding boxes do not overlap, we know that the surfaces do not intersect in the corresponding parameter-space rectangles. In that case, we need not examine these regions of parameter space any further. If the bounding boxes do overlap, the surfaces might intersect in the corresponding regions of parameter space, but we cannot be sure that they do. To learn more, we can subdivide the parameter space. Suppose we subdivide until the bounding boxes all have diagonals smaller than $\epsilon/2$. Then if we find a pair of intersecting bounding boxes from the two surfaces, we can conclude that the surfaces approach each other to within a distance of ϵ in the corresponding regions of parameter space. We refer the corresponding parameter-space regions in such a case as a "dot" and its "mate." Each dot gives us a point on the intersection curve to within the dot tolerance ϵ. The problem is to find an appropriate set of dots which can be linked together to form the trimming curve.

4.1 A Simple Interval Intersection Method

One way to find an appropriate set of dots is to divide each parameter space into a uniform grid, but the cost of such a subdivision is prohibitive. Instead, interval algorithms usually divide space hierarchically, only dividing up regions of space which may contain solutions. In addition to time and space efficiency, the hierarchical alogithms have the advantage of progressive refinement approaches: at all times there is a valid approximation of the entire solution and the approximation improves as the algorithm progresses.

To avoid having to compare every square in one parameter space against every square in the other, each leaf node of the tree maintains a list of the leaf nodes in the other tree which it overlaps. Since the bounding volume of a child must be completely contained within the volume of its parent, when we subdivide, we only need to check the new children against the boxes intersected by their parents. The subdivision step of our algorithm is:

```
Subdivide(node)
  if node's intersect list is not empty
    subdivide node into children
    for each i in node's intersect list
        remove node from i's intersect list
        for each child of node
            if child overlaps i
              add i to child's intersect list
              add child to i's intersect list
```

The basic algorithm for finding intersections is to pick a leaf node from one of the trees and subdivide it. A list of "live" leaf nodes (i.e.: ones which contain overlaps) provides a description of the current model of the intersection curve as well as a "to do" queue. How the next node to be divided is chosen from the list of potential choices provides control over the search algorithm. An obvious choice is searching breadth-first by always choosing the node closest to the root of its tree, which produces an even distribution of sampling (Figure 1). However, we use the ability to control search to create algorithms which fit our needs.

4.2 Search Strategy

Our goal is to compute a set of points on the intersection curves, link them into a polygonal approximation and triangulate the region bounded by the polygon (interior or exterior as appropriate). Doing this requires that we be able to find a set of isolated points on the intersection curves (dots) with a controllable sampling rate. We do this with a two-part search strategy. The first part is a breadth-first search which ensures that each region of parameter space that could possibly contain an intersection is subdivided to a minimum degree. After the breadth-first subdivision, we are left with a set of "live" regions of parameter space which could still contain intersections based on the interval arithmetic tests. Many of these live regions turn out to be false positives – regions which in fact do not contain any intersections despite overlapping bounding boxes. In the second stage of the algorithm, we resolve the false positives using depth-first subdivision. We either find a dot to witness the intersection, or prove that no intersection exists. Figure 1 illustrates the results of the second search phase. The algorithm has proven that many of the live regions of parameter space in figure 2 really do not contain any intersections. In each of the remaining regions, the algorithm provides a dot and its mate in the other parameter space.

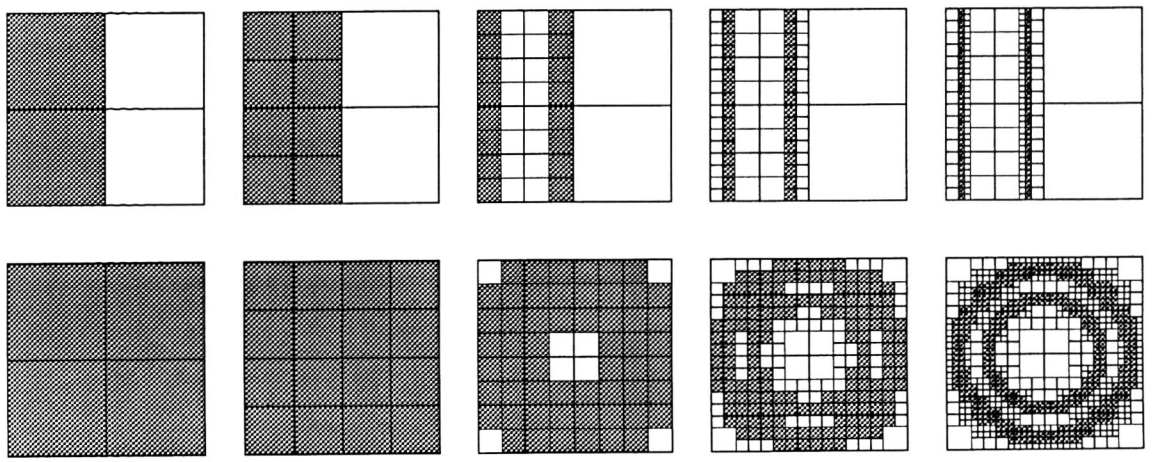

Figure 1: Stages in the breadth-first refinement of the intersection of a torus and a plane. On top is a display of the parameter space of the torus, on the bottom is the plane. All nodes are enclosed by squares. Leaf nodes with non-empty intersect lists are filled with grey.

Beginning with breadth-first subdivision and continuing with depth-first subdivision allows us to control dot spacing and dot accuracy separately. The level of the breadth-first subdivision controls the spacing, and the level of the depth-first subdivision controls the accuracy. This separate control is extremely valuable in practical situations, and is lacking in many algorithms.

5 Stringing and Triangulating

The interval refinement algorithm presented in the preceding sections provides a bounding region on the curves and points on the curves. In this section, we consider the problem of connecting these points together to build a polygonal representation of the curve and to triangulate only a part of parameter space bounded by these curves, to provide a "trimming" operation of cutting one surface against another.

The output of the interval refinement algorithm could be used to drive a marching method intersection. The points provide starting locations, and the bounding regions could help control the search. However, we are interested in directly applying the found points since we assume the user has specified tolerances which will provide a sufficient number of points to be found.

The first thing to notice about the dot connection problem is that the solution is not uniquely determined by the positions of the dots and their surrounding regions. Figure 3 illustrates the kinds of ambiguity which can arise. In order to string the dots into a chain, we must make further assumptions about the underlying intersection curve.

In stringing the intersection points together, we assume that the intersection curve has low curvature relative to the grid size, and that different curves are always seperated by a grid cell at all points along their length. If we are interested in curves which do not cross, this restriction is acceptable if we pick a sufficiently small grid size. Under these assumption, an intersection curve will almost always pass through a cell exactly once, entering and exiting through different sides. Each grid cell which contains part of the intersection will be adjacent to exactly two others, unless it is at an edge. It is straightforward to connect the dots in this case.

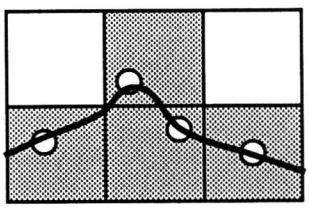

Figure 4: A "bump" is common case where our stringing assumptions fail. It is easy to create a hueristic which handles this special case.

Even if the intersection curve is well-behaved, quantization errors can cause the two neighbor assumption to be violated at any grid resolution. An example of such an error is the "bump" shown in Figure 4. Fortunately, this type of situation is not too difficult to deal with. If we remove the top-most dot in figure 4, the two-neighbor condition is restored and it is easy to connect the dots. Our stringer identifies such situations and removes dots to resolve the ambiguity.

Although we are unable to provide strong guarantees

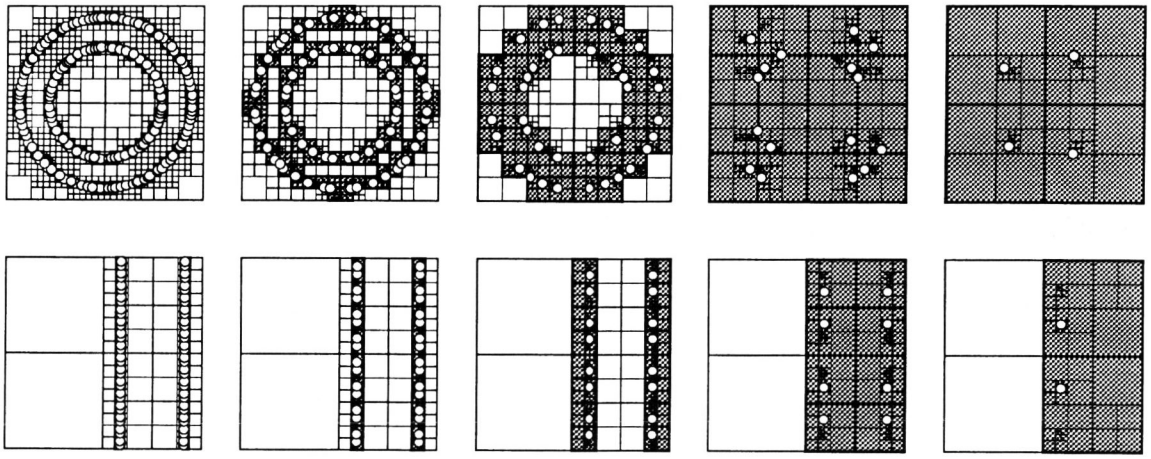

Figure 2: Dot finding is applied to the example of Figure 1. The white circles represent dots. For each gray square in Figure 1, a search for a dot was executed. If no dot was found, the square is rejected, and is not shaded.

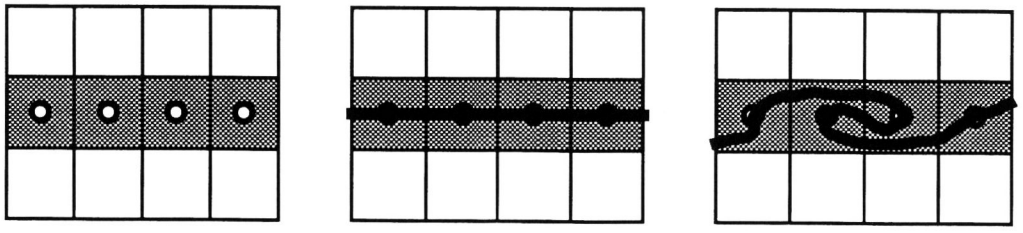

Figure 3: Although the stringing order for a set of dots may seem obvious, the curve may actually do something else. Without making further assumptions, points and bounding regions cannot uniquely determine a stringing order.

about the robustness of our stringer, it has performed well in our limited tests. Obviously, if the intersection is not a curve, but rather some degenerate case such as a surface region, stringing will not succeed in building a curve. Curves which intersect, or meet with tangency, violate the two neighbor criteria. By delaying decisions about ambiguous cases, other parts of the curves can be built correctly, typically providing enough information to make the stringing decisions, or at least satisfy the user. In cases that remain ambiguous, our prototype implementation uses further depth-first subdivision to verify the dot positions. We continue to explore other ways to resolve ambiguous situations with additional subdivision

Once the dots on each surface are connected, corresponding chains on each surface can be merged. This is important since we want to have one description of the curve which has the property that it includes a dot in each grid cell of both parameter spaces through which it passes.

5.1 Triangulating Trimmed Surfaces

One of our motivations for performing intersection calculations is to create trimmed surfaces which can be sewn together. In such cases it is crucial that the pieces can be assembled and mate together without "cracking" at the seams. In order for surfaces to fit together without cracks, their edges (as space curves) must be identical.

Our intersection method provides us with chains of dots in the parameter space of each surface which form piecewise linear approximations to the edges of the trimmed surface. If we triangulate two regions of different parameter spaces bounded by the same chains, the triangles will match without cracking, as shown in figure 5.

In our prototype implementation, we use a flood fill to place triangles in each grid square bounded by the intersection curves and then march around the intersection curve to fill in small triangles around the edge. This simple strategy has the disadvantage that it is not adaptive. To get sufficient detail around complicated intersections, we must also create large numbers of triangles in other areas of the surfaces. Techniques for doing adaptive subdivision could easily be added to our triangulation scheme, and in fact can employ information provided by the interval evaluations.

6 Status and Directions

We have implemented interval arithmetic as part of our mathematical toolkit in C++[8]. The algorithm described in this paper has been implemented and incorporated in an interactive scene composition program (used to create figures 5, 6, and 7) and in a mathematical modeling system [10].

Figures 5, 6, and 7 show results of our algorithm. In figure 5, the intersection curve is used to cut the sphere and cone to create a boundary representation of the boolean subtraction. The front row shows the results for three different levels of initial breadth-first search. Note that since dot size is controlled independently of spacing, even coarse tesselations with few points on the intersection curve still yield objects without cracks. The boolean union is rendered in the rear center. On each side, the separate objects are rendered with textures showing their parameterizations. The red curve on the texture is the trimming curve found by the algorithm. We have rotated the sphere so that its trimming curve is visible. The checkerboard pattern shows the cells that would be created by a uniform subdivision of parameter space to the level used in the breadth-first search of the front-center subtraction. Note that the actual breadth-first subdivision is not uniform. The blue squares are breadth-first subdivisions through which the trimming curve passes.

In figure 6, a sphere whose radius is modulated with a sinusoid is intersected with a torus. Figure 7 shows the intersection of a $sin(r)/(r+\epsilon)$ height field with a sphere. In both cases, the objects have been texture mapped in the same manner as figure 5. Figure 7 shows the union from top and bottom.

Although the prototype has been useful for creating a wide variety of interesting objects, there are several extensions to the basic algorithm which would be interesting to explore. For example, the search and triangulation techniques could be made to be non-uniform and adaptive to improve performance on surfaces with varying levels of detail. Stringing could be improved by using further subdivision to disambiguate difficult cases.

The surface intersection technique presented here can handle a very wide variety of surfaces because of its reliance on interval arithmetic. The algorithm progressively refines its model of the curve until user specified tolerances are met allowing separate control over the spacing and accuracy of the intersection points. Stringing the points together and triangulating the resulting parameter-space regions makes it possible to construct crack-free boundary representations of objects which are difficult to create with other methods.

Acknowledgements

This research was conducted at Apple Computer and at Carnegie Mellon. At Carnegie Mellon, this research is supported in part by a grant from Apple Computer

and by an equipment grant from Silicon Graphics Incorporated. The first author is supported in part by a fellowship from the Schlumberger Foundation. Photorealistic RenderMan software, which was used to create images in this paper, was provided to Carnegie Mellon through Pixar's Renderman Education Program.

References

[1] Jarmo Alander. On interval arithmetic range approximation methods of polynomials and rational functions. *Computers and Graphics*, 9(4):365–372, 1985.

[2] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, 1983.

[3] C. Asteasu and A. Orbegozo. Parametric piecewise surfaces intersection. *Computers and Graphics*, 15(1):9–13, 1991.

[4] R. E. Barnhill, G. Farin, M. Jordan, and B. R. Piper. Surface / surface intersection. *Computer Aided Geometric Design*, 4:3–16, 1987.

[5] John J. Chen and Tulga M. Ozsoy. An intersection algorithm for C2 parametric surface. In Alison Smith, editor, *CAD86: Seventh International Conference on the Computer as a Design Tool*, pages 69–77. Butterworths, September 1986.

[6] Koun-Ping Cheng. Using plane vector fields to obtain all the intersections curves of two general surfaces. In *Theory and Practice of Geometric Modelling*, pages 187–204. Springer-Verlag, 1989.

[7] Daniel Filip, Robert Magedson, and Robert Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3:295–311, 1986.

[8] Michael Gleicher and Andrew Witkin. Snap together mathematics. In Edwin Blake and Peter Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*. Springer Verlag, 1991. Also appears as CMU School of Computer Science Technical Report CMU-CS-90-164.

[9] Elizabeth G. Houghton, Robert F. Emnett, James D. Factor, and Chaman L. Sabharwal. Implementation of a divide-and-conquer method for intersection of parametric surfaces. *Computer Aided Geometric Design*, 2:173–183, 1985.

[10] Michael Kass. CONDOR: constraint-based data flow. *Computer Graphics*, 26, July 1992. to appear.

[11] Sheldon Katz and Thomas Sederberg. Genus of the intersection curve of two rational surface patches. *Computer Aided Geometric Design*, 5:253–258, 1988.

[12] Gabor Lukacs. The generalized inverse matrix and the surface-surface intersection problem. In *Theory and Practice of Geometric Modelling*, pages 167–185. Springer-Verlag, 1989.

[13] Don P. Mitchell. Robust ray intersection with interval arithmetic. In *Graphics Interface*, pages 68–72, 1990.

[14] Michael Mortenson. *Geometric Modelling*, chapter 7: Intersections, pages 319–344. John Wiley & Sons, 1985.

[15] S. P. Mudur and P. A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics and Applications*, pages 7–17, February 1984.

[16] M. J. Pratt and A. D. Geisow. Surface/surface intersection problems. In J. A. Gregory, editor, *The Mathematics of Surfaces*, pages 117–142. Clarendon Press, 1986.

[17] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *Computing Surveys*, 12(4):437–464, December 1980.

[18] T. W. Sederberg and R. J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric design*, 5:161–171, 1988.

[19] Thomas Sederberg and Tomoyuki Nishita. Geometric Hermite approximation of surface patch intersection curves. *Computer Aided Geometric Design*, 8:97–114, 1991.

[20] Kevin G. Suffern. Interval methods in computer graphics. *Computers and Graphics*, 15(3):331–340, 1991.

[21] Daniel Toth. On ray tracing parametric surfaces. *Computer Graphics*, 19(3):171–179, July 1985.

[22] Brian Von Herzen and Alan Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics*, 21(4):103–108, July 1987. Proceedings SigGraph '87.

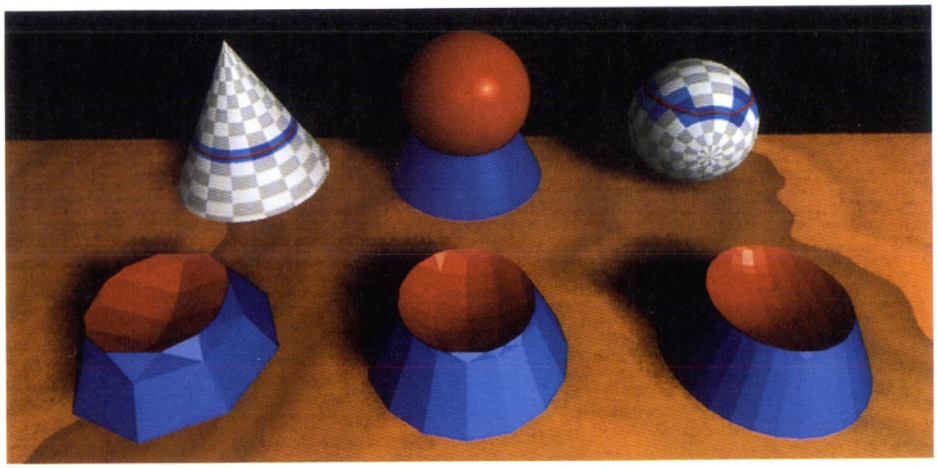

Figure 5: Trimming a cone against a sphere. Note that even coarse tesselations are crack-free.

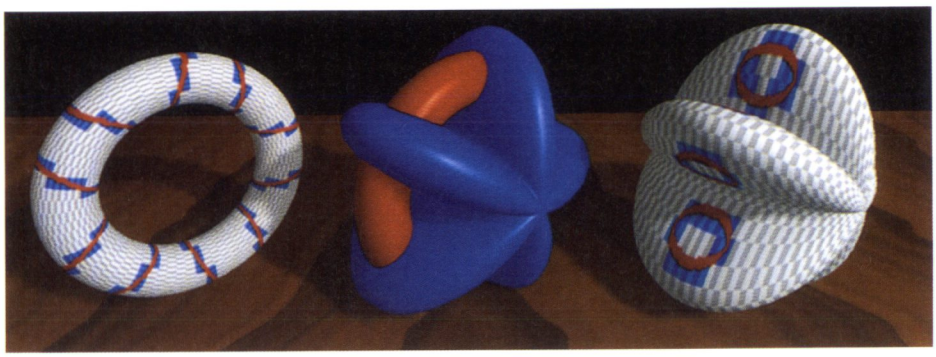

Figure 6: Intersection of a five-lobed surface and a torus.

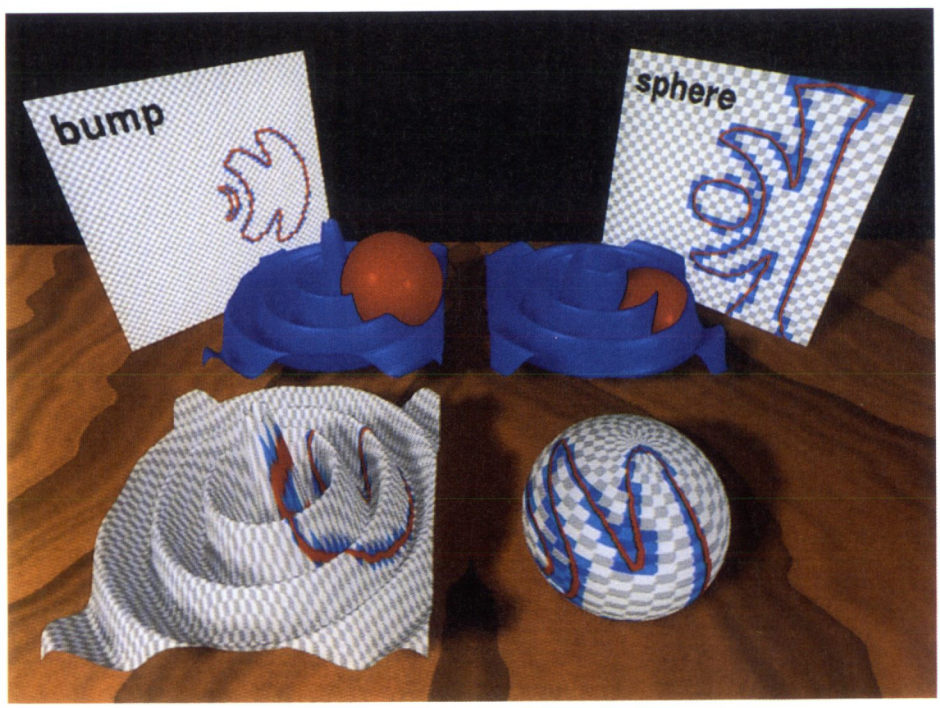

Figure 7: Intersecting a sphere with a $sin(r)/r$ bump. The middle right is a bottom view of the middle left.