Screen-Parallel Calculation of Surface Intersections

David C. Banks

Department of Computer Science

Mississippi State University

ABSTRACT

When surfaces intersect, one may desire to highlight the intersection curve in order to make the shape of the penetrating surfaces more visible. Highlighting the intersection is especially helpful when the surfaces become transparent, because transparency makes the intersections less evident. This paper discusses a technique for locating intersections in screen space using only the information locally available to a pixel. The technique is designed to exploit parallelism at the pixel level and has been implemented on the Pixel-Planes 5 graphics supercomputer.

1 Introduction

Consider the problem of locating intersections of surfaces in order to highlight them in a displayed image. If the geometry of the surfaces in 3-space were static, the intersection curves might be analytically computed [Baraff, Moore] once and for all. When the surfaces change shape dynamically, those intersections must be recomputed during every frame. Finding intersections of n polygons in 3D world space is a significant computational burden, of order $n \log n$ or worse, which can cripple an application that once was interactive. One can instead test for intersections when the surfaces are projected to the 2D screen. The additional cost of the test is small when added to z-buffering, and it lends itself to a parallel implementation in screen-space.

I will describe the simplest version of the algorithm and show it is correct. I will then show two ways to improve the appearance of the intersections. The first modification thickens the intersection curves as they appear on the screen. The second modification makes the thickened curves more uniform in width.

2 The Simple Approach: *ExactMatch*

A simple way to detect intersections in screen space is to modify the usual z-buffer algorithm, where z measures depth perpendicular to the screen. The surface primitives might be spheres or curved patches, but for simplicity I will refer to them as polygons. What characterizes the geometry of a visible intersection? First, there must be at least two polygons that pass through the same point. If two polygons share the same (x, y) coordinates in normalized eye space, they will share the same pixel on the screen. If they share the same z-value, there must be an intersection. Second, the intersection must not be hidden by other polygons. That means that the shared z-value at a visible intersection must be the minimum over all polygons that cover the pixel.

This specification leads to a natural implementation. The z-value of each incoming polygon at a pixel is compared with the contents of the z-buffer. The pixel acquires the polygon's state (color, normal, and depth) if

```
foreach pixel in screen
   pixel.z     = infinity;
   pixel.iFlag = FALSE;

foreach k in [1..numPolygons]
   foreach pixel in rasterize(polygon[k])

      Predicates P1(k), P2(k), P3(k)

      if zval(polygon[k], pixel) < pixel.z then                     (1)
         copy(polygon[k], pixel);                                    (2)
         pixel.iFlag = FALSE;                                        (3)

      elseif zval(polygon[k], pixel) == pixel.z then                 (4)
         pixel.iFlag = TRUE;                                         (5)

Predicate P4
```

Figure 1. Code fragment for finding polygon intersections using *exactMatch*. If the frontmost polygon has the same z-value as the z-buffer, the intersection flag is set.

the polygon's z-value is smaller than the pixel's. But if the polygon's depth matches the z-buffer, an intersection flag is set. Whenever a new frontmost polygon arrives, the intersection flag becomes unset. The result is that all visible intersections will be flagged at the pixel. Call this the *exactMatch* algorithm.

The C-like code fragment (Figure 1) illustrates the naive *exactMatch* algorithm, showing the comparisons performed for each pixel as polygons stream past it. The variables `polygon` and `pixel` are both assumed to be data structures containing color and other relevant geometric state. The Boolean variable `iFlag` tells whether there is a visible intersection at the pixel. The function `zval()` calculates the z-value of a polygon at a particular pixel. The function `rasterize()` returns the set of pixels covered by a polygon. The function `copy()` interpolates the color, depth, and normal of a polygon at a particular pixel, applies lighting, then puts the results into the pixel.

The sample code loops over all polygons in a set and loops over all pixels in a rasterized polygon. The fragment is unrealistic in many regards. For example, the function-call to copy() is an inefficient way to interpolate and illuminate. The code only exposes the essential elements of making comparisons, copying data, and managing the intersection flag.

Each pixel is initialized by setting its depth to some large number (in practice, to the extreme value of the z-buffer's range) and lowering the intersection flag. The if-clause in lines (1-3) implements the traditional z-buffer test-and-copy, augmented by lowering the intersection flag. The if-clause in lines (4-5) raises the flag when an incoming polygon's depth matches the z-buffer and thereby matches the frontmost polygon that has visited the pixel.

Since the flag can be repeatedly set and unset, one might wonder whether order makes any difference when processing the polygons. Might the flag be inadvertently set or unset by some unusual mix of incoming z-values?

$P1(k)$: If $i \in [1, k\text{-}1]$ then `pixel.z` $\leq$ `zval(polygon[`i`], pixel)`

$P2(k)$: If k-1>0 then $\exists i \in [1, k\text{-}1]$ such that `pixel.z` = `zval(polygon[`i`], pixel)`

$P3(k)$: `pixel.iFlag` $\Leftrightarrow$ {$\exists i, j \in [1, k\text{-}1]$ such that $i \neq j$ and
`pixel.z` = `zval(polygon[`i`], pixel)`
= `zval(polygon[`j`], pixel)`}

$P4$: $\forall$ `pixel` $\in$ `screen`,
`pixel.iFlag` $\Leftrightarrow$ {$\exists i, j \in$[1, *numPolygons*] such that $i \neq j$ and, for $k \in$ [1, *numPolygons*],
min(`zval(polygon[`k`], pixel)`) = `zval(polygon[`i`], pixel)`
= `zval(polygon[`j`], pixel)`}

Figure 2. Predicates associated with the *exactMatch* code in Figure 1.

The algorithm looks plausible, but is it actually correct in all cases? Much of computer graphics relies on various approximations or contrivances designed to make an image look right. When an algorithm has a strong geometric or mathematical component, it is often possible to prove its correctness – especially if the algorithm is short. This worthy goal is made unpleasant by the difficulty of applying propositional calculus to convert preconditions into postconditions. I shall take the middle ground between (A) a formal proof full of logical symbols and derivations, and (B) no proof at all. What follows is a one-page prose-proof verifying the code illustrated in Figure 1.

2.1 Proof of Correctness for *ExactMatch*

To prove correctness, one must show that the algorithm terminates with the correct postcondition *P4*. The code clearly terminates, since the outer loop executes only `numPolygons` times and there are only finitely many pixels in the inner loop. The body of the inner loop is annotated (Figures 1 and 2) with three predicates $P1(k)$, $P2(k)$, and $P3(k)$. Each predicate has an argument: the loop variable k. Predicate *P1* asserts that a pixel's z-value is *at least as small* as that of any processed polygon. Predicate *P2* asserts that the pixel's z-value actually *matches* that of some processed polygon. Together these two predicates establish that the pixel's z-value is the minimum of the polygons processed thus far in the loop. Predicate *P3* asserts that the intersection flag is raised exactly when two different polygons match the z-value stored in the pixel. The postcondition P4 claims that the intersection flag is raised precisely when there are two exact matches of the minimum (frontmost) z-values at a pixel. This was the characterization of a visible intersection that the previous section described.

The loop invariants are true upon entry into the loop, which is demonstrated as follows. Predicates $P1(1)$ and $P2(1)$ are vacuously satisfied, since k-1=0 and hence the left-hand side of each implication is false. Predicate $P3(1)$ is also satisfied: `pixel.iFlag` is initialized to `false`, and there can be no i and j in the range [1, 0], which makes the right-hand side of the equivalence $P3(1)$ false as well.

Next, it is required that these predicates be truly invariant and that they imply *P4*. What happens to predicates *P1*-*P3* after an iteration of the loop? Consider each one in turn.

P1: Whenever $P1(k+1)$ is true at the top of the loop, control passes over lines (2-3) to leave $P1(k+1)$ true at the bottom of the loop. If $P1(k+1)$ is false at line (1), line (2) re-establishes `pixel.z` as the minimum and $P1(k+1)$ then becomes true.

P2: The first time through the loop, `pixel.z` is larger than any polygon's *z*-value, so line (2) is executed to make `pixel.z` match a polygon, and *P2(2)* becomes true. On subsequent iterations `pixel.z` stays fixed or else is assigned the *z*-value of `polygon[k]`, so *P2(k+1)* is true at the end of the iteration.

P3: When an incoming polygon has the new minimum, *P1(k+1)* is false, in which case line (2) makes the right-hand side of *P3(k+1)* false by assigning to `pixel.z` the new minimum. Line (3) corrects the intersection flag to re-establish the equivalence in *P3(k+1)*. When the incoming polygon does not contain a new minimum, it may either match the existing minimum stored in `pixel.z` or else exceed the minimum. If it matches, line (5) sets the flag to establish *P3(k+1)*. If it exceeds it, then both sides of the equivalence remain the same and so *P3(k+1)* = *P3(k)*.

The above arguments mean that $Pn(k) \Rightarrow Pn(k+1)$ for $n = 1, 2, 3$. In other words, these predicates are invariant within the inner loop. Consequently, the termination of the loop delivers predicates *Pn(numPolygons+1)*. When the loop completes, the predicates *P1(numPolygons+1)* and *P2(numPolygons+1)* guarantee that `pixel.z` holds the minimum *z*-value of the polygons in the array, and the predicate *P3(numPolygons+1)* guarantees that this minimum corresponds to a pair of polygons exactly when the intersection flag is raised. These predicates are valid at each pixel. Thus the postcondition *P4* is true, proving correctness of the code in Figure 1. That should mean that a visible intersection arises when two different polygons share the frontmost *z*-value at a pixel, which occurs exactly when `iflag` is true. It should, but it doesn't quite.

2.2 Spurious Intersections Between Adjacent Polygons

The predicate *P4* captures the geometric meaning of intersection. But rasterizing actual polygons can lead to problems along shared edges. Two polygons that share an edge will intersect each other along it in a formal, mathematical sense. If one is not careful about determining the extents of spans, a pair of polygons whose edges pass exactly through pixels will "intersect" at those pixels. One could be careful not to redraw pixels along the common edge of adjacent polygons either by maintaining connectivity information or by using strict inequalities when rasterizing, say, the bottom and left edges of a polygon. But the connectivity information is difficult to maintain for datasets spread across multiple processors, especially if the geometry changes dynamically. The strict inequalities will strip off the edges of a rectangle whose corners lie at pixel locations (leaving 1-pixel-wide gaps), and such datasets occur in a wide variety of applications.

There is a simple remedy, at least for Phong-shaded polygons. During Phong-shading the pixel already holds sufficient information to detect spurious intersections of adjacent polygons. That information is the surface normal. The genuine intersections are those of polygons that dive through each other, that is, whose normals are different where they interpenetrate. One can thus modify the *z*-comparison, requiring that the normals be different. When a new *z*-value matches an old one, the new normal vector must differ from the old as well. Otherwise the new polygon probably shares a common edge with the old polygon at this pixel. The condition in line (4) can be enlarged as follows to produce a variation *exactMatch′*.

```
elseif zval(polygon[k], pixel)   == pixelState.z and
       normal(polygon[k], pixel) != pixel.normal then          (4′)
```

The drawback now is that the algorithm fails to locate points where two surfaces osculate, or where a single surface branches. At such points the tangent planes coincide, and thus the normals do as well. That drawback is

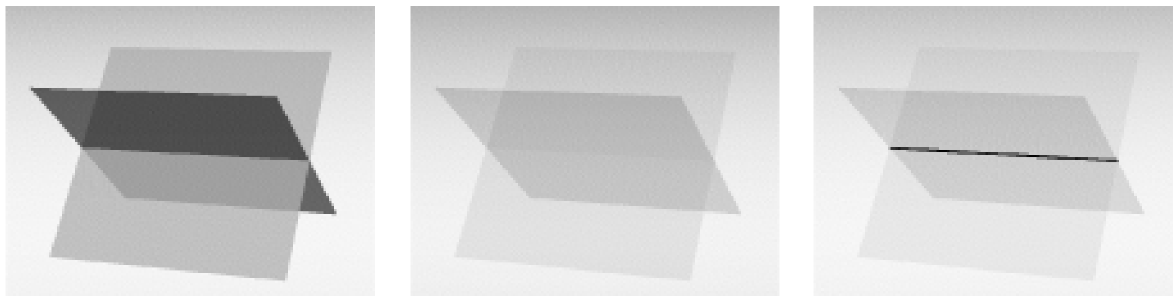

Figure 3. Two intersecting polygons are slightly transparent (left) and very transparent (middle and right). Transparency makes the intersection hard to see, hence the need for highlighting the intersection.

minor for the vast majority of surfaces one actually encounters: first-order contact between curved surfaces is a rare event.

2.3 Combining *ExactMatch* with Transparency

Transparency, in conjunction with highlighted intersections, is an especially helpful method for visualizing surfaces that intersect. In the neighborhood of an intersection, portions of one surface hide portions of another throughout every (or almost every) viewing direction. Transparency makes it possible to see the continuation of the portions that are hidden and hence to perceive the shapes of both surface patches near the intersection. When two opaque surfaces have different colors near their intersection, it is visually obvious that an intersection occurs: there is a discontinuity in the color transverse to the intersection curve. Unfortunately, the colors of two transparent surfaces *A* and *B* blend together. As transparency increases, *A* atop *B* becomes indistinguishable from *B* atop *A* (Figure 3). The resulting color does not change appreciably across the intersection and so the intersection is no longer visually obvious.

Consider how to detect visible intersections of transparent surfaces. Every intersection is visible, whether it is frontmost or not. The order matters when rendering transparent polygons. One might wish to employ a painter's algorithm and traverse the set of polygons from back to front (or, alternatively, render from front to back). But detecting intersections in world space, splitting polygons that intersect so that they can be well-ordered, and sorting them is exactly the computational burden that *exactMatch* was intended to avoid! If *n* is the number of polygons, there is a simple solution whose performance is very good for scenes with maximum depth complexity smaller than $O(\log n)$. That solution is to use multipass transparency [Mammen].

The multipass algorithm requires additional pixel-state in the form of variables `zFar`, `alpha`, and `color`. The pixel-variable `zFar` is first initialized to 0. Then, during each pass, a pixel collects the state of the nearest polygon which is farther than `zFar`. At the end of the pass, `zFar` is updated to match that nearest polygon and the polygon's color is blended with the pixel's accumulated `color`, by an amount determined by the opacity `alpha`.

ExactMatch can be incorporated gracefully into the multipass transparency algorithm. In each pass *exactMatch* detects the frontmost intersection, and the multiple passes work their way from front to back among the polygons that cover a pixel. How many passes are required? That depends entirely on the depth complexity of (a region of) the screen. One attractive feature of the multipass algorithm is that the depth complexity does not generally

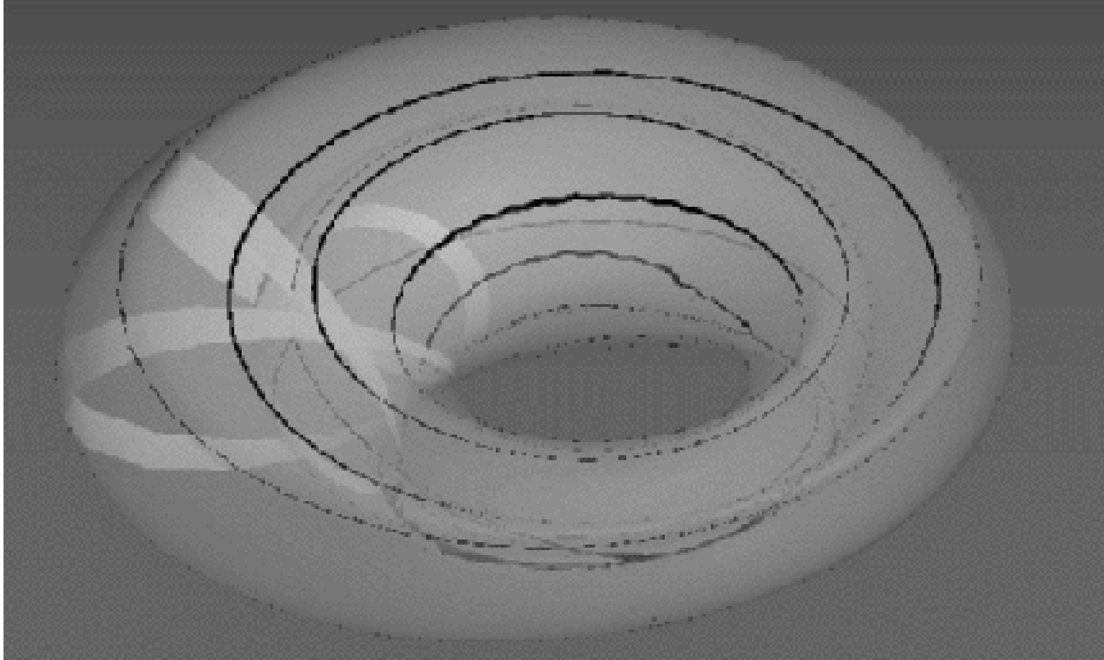

Figure 4. Surface of revolution generated by a trefoil knot (indicated by the opaque ribbon on the left) spun in four-dimensional space. The surface is projected to three-dimensional space, forming a self-intersecting torus.

change when the surfaces are re-meshed at a higher resolution[1]. In other words, once one has chosen a set of surfaces to examine, the depth complexity is essentially $O(1)$ when the surfaces are refined to a higher level of detail.

Figure 4 illustrates a self-intersecting torus. This surface was generated by constructing a trefoil knot in 3-dimensional *xyz*-space, then sweeping out a surface of revolution through the *xw*-plane in 4-dimensional *xyzw*-space. Mathematicians who study knotted surfaces must contend with the fact that any 3-dimensional projection of such a surface must possess self-intersections. Transparency helps reveal the complicated shape of the surface, and highlighted intersections help the eye track the many intersection curves along the surface.

3 Varying-width Intersection Curve: *Threshold*

By requiring an exact match between polygons' z-values one can highlight, at best, a 1-pixel-wide intersection curve (except, of course, for the degenerate case of coincident polygons). At worst one misses most of the curve due to imperfect sampling: identical z-values of different polygons are unlikely to fall on exact pixel locations. One remedy to this problem is to apply a threshold. If the incoming polygon is within ε of the z-buffer, its intersection flag is raised. The condition in line (4) is relaxed as follows to produce the algorithm *threshold*.

```
elseif |zval(polygon[k], pixel) - pixel.z| < ε then                (4″)
```

1. There is a caveat: such surfaces must have bounded curvature almost everywhere. Otherwise the finely-sampled geometry may disclose complicated features that increase the depth complexity. Such features are ubiquitous in fractals, for example.

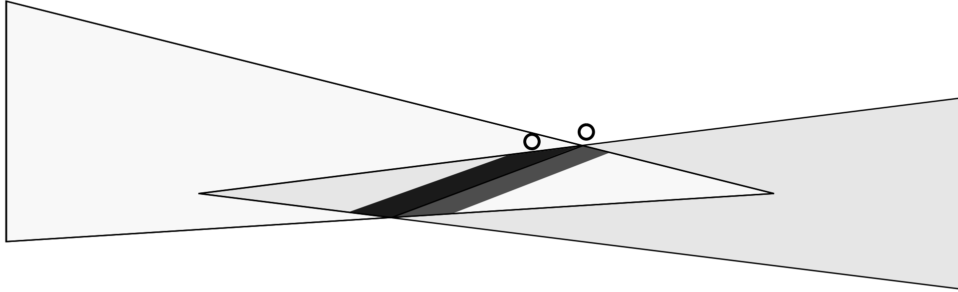

Figure 5. At their common intersection, two polygons share *z*-values. The *z*-values are within some threshold of each other along a thickened intersection curve. The two circles indicate pixels that lies very close to the intersection but fails to lie in the thick curve. The circle on the left is supported by only one polygon. The circle on the right is supported by no polygons.

Threshold produces a thickened intersection curve (Figures 3 and 4). The curve typically varies in width as it winds across the screen. If two polygons intersect each other at a shallow angle, their *z*-separation remains small over a large area of the screen and the curve that satisfies line (4″) is many pixels wide. If they intersect each other at a steep angle, a short excursion to neighboring pixels finds them separated far apart in the *z*-direction (Figure 7) and therefore the curve is thin. Notice also that *threshold*, like *exactMatch*, merges gracefully into the multipass algorithm for transparency.

There is an implementation detail to consider. In order to correctly *z*-buffer, one generally interpolates reciprocal-*z* (not *z* itself) across the polygon because perspective projection does not preserve linear functions. For locating intersections across large ranges it is necessary to compare distances using *z* itself, but over small values of ε the distinction is not so important and thus one need not bother recomputing *z* from its reciprocal.

3.1 Artifacts of *Threshold*

One artifact of *threshold* is that the thickened intersection curve gets trimmed by surface boundaries, since the depth-comparison is performed only in the *z*-direction (rather than, say, in the normal directions of the participating polygons). In other words, a pixel can only be construed to lie near an intersection if two polygons lie atop the pixel. In the case of *exactMatch*, $\varepsilon = 0$ which implies that there truly are at least two polygons at the pixel. But with *threshold*, the implication no longer holds. A pixel may lie very close to the intersection without being included in the thick curve, because it fails to be supported by two or more polygons.

Figure 5 illustrates the situation for two interpenetrating triangles. The thick intersection curve (dark gray bands) can only be calculated at pixels covered by both triangles. As a result the thick curve is trimmed at the ends to produce bevels. Such bevels can result from surface edges or from silhouettes that lie near the intersection. Bevelling may be desirable or not; in any case it is hard to overcome without using pixel-to-pixel communication (which would let a pixel know it is near an intersection, even though is has received no geometric state that would inform it of its proximity to the intersection).

A second artifact that arises is that *threshold* may detect nonexistent intersections. As long as one polygon lies within ε of another (in the *z*-direction), a portion of it will be deemed to lie within a thick intersection curve. A near-miss is highlighted as though it were the neighbor of a direct hit between two polygons.

A special case of the false intersections occurs along silhouettes. A silhouette edge separates two polygons, one frontfacing and the other backfacing, that cover some of the same area on the screen. The *z*-distance between the

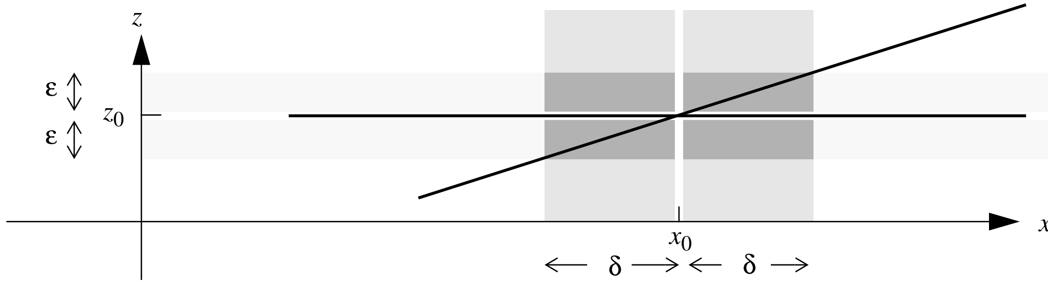

Figure 6. To produce a fixed-width intersection in 2 dimensions, the goal is to determine whether a point on a segment lies within δ of the intersection. Depending on the slopes of the two segments, the point must lie within some ε (in the z-direction) of the other surface.

two polygons falls to zero at the silhouette, so the algorithm detects intersections in the vicinity of the silhouette. A simple remedy for this behavior, when each individual surface is closed, non-self-intersecting, and opaque, is to cull backfacing polygons. A more elaborate solution would be to interpolate higher-order curvature information across each polygon in order to determine whether two polygons that share a pixel are likely to be adjacent. This is analogous to the modified line (4″) which uses the interpolated normal to detect a shared edge. An effective, but ad-hoc, remedy is to ignore the intersection flag if the frontmost surface has a normal nearly perpendicular to the eye-vector. In practice the false intersection highlight is unobtrusive at the silhouette, since the polygons on either side approach the silhouette at steep angles. Over a short distance on the screen, their z-values separate by a large amount and the false intersection curve is quite thin.

These artifacts rule out *threshold* as an exact rendering technique, especially for datasets where it is important to distinguish between true intersections and near-misses. But when interactive speed is the driving concern, *threshold* permits an implementation at the pixel-parallel level to yield high performance.

4 Fixed-width Intersection Curves: *epsilon*

In order to draw a fixed-width intersection curve one is required to determine, at any point on the surface, the distance from the point to the nearest intersection curve (where distance is measured on the screen). Given two intersecting polygons and a point $\mathbf{p} = (x_p, y_p)$ on the screen, how far is $\mathbf{p}$ from the intersection?

The situation is easy to see in two dimensions (Figure 6). Instead of polygons in (x, y, z)-space, suppose two line segments in (x, z)-space intersect one another at (x_0, z_0) on a one-dimensional screen. At a horizontal distance δ from the intersection, corresponding points on the two segments will be separated by a vertical distance ε. Whenever the vertical distance $\Delta_z(\mathbf{p})$ is less than ε, the screenwise distance $\Delta_x(\mathbf{p})$ is less than δ, so $\mathbf{p}$ is included in the fixed-width intersection curve. The figure shows the geometry for a one-dimensional screen viewed edge-on, with z now pointing upwards.

Let the planes of the two polygons in (x, y, z)-space be represented by the following equations:

$$z_1(x, y) = a_1 x + b_1 y + c$$

$$z_2(x, y) = a_2 x + b_2 y + c$$

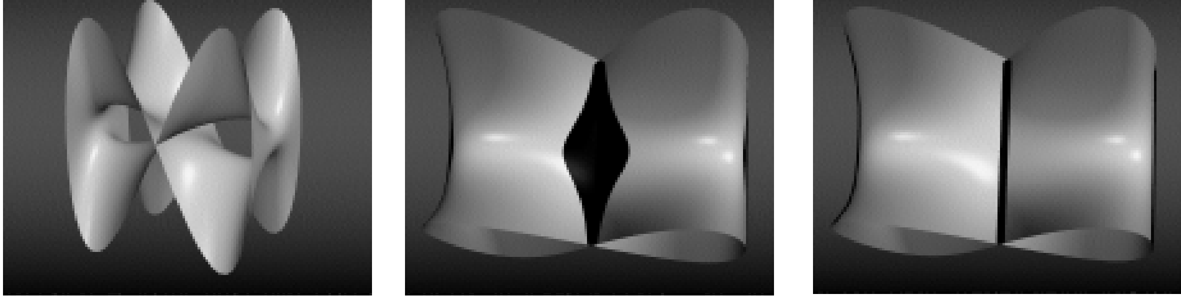

Figure 7. This surface has cross-sections that form figure-eights. The figure-eights self-intersect steeply at the front and at the rear of the surface, but self-intersect at a shallow angle in the middle of the surface (left). Simple thresholding of *z*-distance locates a thick intersection curve of varying width (center). By compensating for the slopes one can make the curves fixed-width.

The *z*-separation $\Delta_z(\mathbf{p})$ between the two planes is simply the difference between their plane equations:

$$\Delta_z(\mathbf{p}) = z_2 - z_1 = (a_2 - a_1)x_p + (b_2 - b_1)y_p + (c_2 - c_1)$$

The intersection occurs where $\Delta_z(\mathbf{p}) = 0$. The gradient vector $\mathbf{v} = (a_2 - a_1, b_2 - b_1)$ specifies the direction of maximum slope of $\Delta_z(\mathbf{p})$. In that direction the equation for a line is

$$\begin{aligned}\Delta_z(t\mathbf{v}) &= (a_2 - a_1)\, t\, (a_2 - a_1) + (b_2 - b_1)\, t\, (b_2 - b_1) + (c_2 - c_1) \\ &= [(a_2 - a_1)^2 + (b_2 - b_1)^2]\, t + (c_2 - c_1)\end{aligned}$$

and so the slope m satisfies

$$m^2 = (a_2 - a_1)^2 + (b_2 - b_1)^2$$

Dividing $\Delta_z(\mathbf{p})$ by the magnitude of the gradient yields a normalized plane equation with the same zero-set, but with a slope of 1 in the gradient direction. In such a normalized plane, the height $\Delta_z(\mathbf{p})/m$ is precisely the same as the screenwise distance $\Delta_{xy}(\mathbf{p})$ between the point $\mathbf{p}$ and the zero-set along the intersection. Therefore, given some $\delta>0$ (the half-width of the curve), choose

$$\varepsilon = \delta m$$

which ensures that

$$\Delta_z(\mathbf{p}) < \varepsilon \;\Rightarrow\; \Delta_{xy}(\mathbf{p}) < \delta$$

That is, the point $\mathbf{p}$ on the screen lies within δ of the intersection if the difference in polygon depths is no larger than ε in magnitude. Note that when the two planes are parallel or coincident, ε is exactly zero and the left-hand inequality is nowhere satisfied. The computation is limited in practice by the precision of the *z*-buffer and of the plane coefficients, so planes separated by small angles are effectively parallel due to numerical representation.

```
foreach pixel in screen
   pixel.first.z = pixel.second.z = infinity;
   pixel.iFlag   = FALSE;

foreach k in [1..numPolygons]
   foreach pixel in rasterize(polygon[k])
      if zval(polygon[k], pixel) < pixel.second.z                   ( 1)
         if zval(polygon[k], pixel) < pixel.first.z                 ( 2)
            The incoming polygon lies in front of the 2 frontmost values at the pixel.
            pixel.second = pixel.first;                             ( 3)
            copy(polygon[k], pixel.first);                          ( 4)
         else                                                       ( 5)
            The incoming polygon lies between the frontmost two values at the pixel.
            copy(polygon[k], pixel.second);                         ( 6)

foreach pixel in screen
   if (pixel.second.z != infinity)
      There are 2 polygons. See if they intersect nearby, based on their plane coefficients.
      slopeSquared = (pixel.second.a - pixel.first.a)^2 +           ( 7)
                     (pixel.second.b - pixel.first.b)^2;            ( 8)
      epsilon = delta*sqrt(slopeSquared);                           ( 9)
      if |pixel.first.z - pixel.second.z| < epsilon                 (10)
         pixel.iFlag = TRUE;                                        (11)
```

Figure 8. Code fragment for finding polygon intersections using *epsilon*. The frontmost pair of polygons are retained at each pixel. After all the polygons have been rasterized, each pixel uses the plane coefficients of the frontmost samples in order to calculate `epsilon`. If the difference in z-depths is smaller than `epsilon`, the pixel lies within a fixed-width intersection curve.

One has the choice of either storing the plane coefficients in the pixel or recovering them from the Phong-interpolated normal in the pixel. The first way is faster, but the second way uses less memory. To compute plane coefficients a and b from a normal $\mathbf{n} = (x_n, y_n, z_n)$ with $z_n \neq 0$, let

$$a = -\frac{x_n}{z_n}, \qquad b = -\frac{y_n}{z_n}$$

It is easy to see that $\mathbf{n}$ is perpendicular to the plane $(x, y, ax + by)$. Notice that only the a and b coefficients are needed in order to compute m.

The calculation of ε uses only the linear information from the participating surfaces. But that is sufficient to dramatically improve the uniformity of intersection highlights even for curved surfaces (Figure 7).

How should the calculation of ε enter into the algorithm for finding intersections? Is it enough to just modify line (4) yet again? The answer this time is no. Consider a polygon P, parallel to the plane of the screen, with several other polygons situated behind it. If the backmost polygon B is nearly perpendicular to the screen, the slope m (of the difference between the plane equations) is very large. As a result, every pixel over P and over B might satisfy

the condition $\Delta_z(\mathbf{p}) < \varepsilon$. Even though P and B are completely separated by intervening polygons, the inequality suggests that an intersection between them is likely to occur nearby. As a result, pixels would highlight false intersections. The simplest solution is to retain the state of the frontmost two polygons at each pixel. Only these two set of plane coefficients are used in order to compute the screenwise distance to an intersection. This computation is performed in a second pass over the pixels.

Figure 8 shows the improved fixed-width calculation. The pixels (now storing data for the first- and second-frontmost samples) are all initialized as before. When a polygon is rasterized into pixels, the contents of each pixel are updated to maintain the geometry of the frontmost polygonal samples. This occurs in lines (1-6). In the last pass over the pixels, the slopes and z-values of the frontmost two samples determine whether an intersection is nearby, based on some threshold `delta`. This occurs in lines (7-11).

5 Implementation

I implemented the varying-width and fixed-width intersection algorithms on Pixel-Planes 5 graphics supercomputer [Fuchs]. Its architecture assigns a SIMD arrays of processors (each with 208 bits of local memory) to pixels in rectangular regions of the screen. Application development on Pixel-Planes 5 is supported by a PHIGS-like graphics library. This library allocates the pixel memory into fixed areas, storing color, normal, depth, and texture information. The allocation scheme permitted me to implement the varying-width algorithm in a straight-forward fashion [Banks]. But maintaining an additional normal-vector (48 bits), a second z-value (24 bits), and scratch space for calculating epsilon (64 bits) simply consumes too many bits to implement the fixed-width algorithm in Figure 8. As a compromise, I eliminated the last pass by folding the comparison of lines (7-11) into the rendering pass of lines (1-6). As noted in the previous section, this strategy can be sensitive to the order in which polygons are processed. Even so, for a broad collection of intersecting surfaces the thickened curves are satisfactory. The results are illustrated in Figures 4 and 7.

Pixel-Planes 5 can transform and Phong-shade at a sustained rate of well over 1 million polygons/second using hand-coded assembly instructions for the i860 graphics processors, or about 100,000 polygons/second using compiled C code. The difference in speed is due primarily to two factors: the compiler does not make use of the processor's dual instruction mode, and the hand-coded assembly fits into the instruction cache. It was the C code that I modified in order to locate intersection curves. The best sustained performance for variable-width curves was 90,000 polygons/second, and for fixed-width curves was 66,000 polygons/second. Finding variable-width intersections in the context of multipass transparency (for a surface with up to 8 layers) yielded 17,000 polygons/second. These rates were measured using the torus in Figure 4, composed of 8192 triangles, at a screen resolution of 1280x1024 pixels.

6 Conclusions

This paper describes a technique for locating, in screen space, intersections between surfaces. This makes it possible to enhance the appearance of an intersection either by changing its color, its opacity, or its width. Transparency helps reveal the interior shape of intersecting surfaces but blends away the intersection curve; changing the curve's color or opacity compensates for the effect of transparency.

The algorithm relies solely on the local information available to a pixel. In the simplest form the algorithm looks for exact matches between coordinates of polygons and can be proven correct. Unfortunately, the exact-matching scheme misses most points on an intersection curve due to discrete per-pixel sampling of z-values. One can relax the requirement for an exact match by comparing z-differences against a more generous threshold. This admits

many more pixels into the intersection curve but is no longer correct: the thickened curve has varying width, the curve is trimmed whenever it is not supported by at least two polygons, and false intersections are detected due to near-misses, silhouettes, or distant polygons that are steeply sloped. The first drawback can be remedied by normalizing the threshold according to the slopes of the polygons and testing only the frontmost pair of polygons. This produces fixed-width intersection curves. The other drawbacks require higher-order approximations, pixel-to-pixel communication, or ad-hoc measures in order to be overcome. The advantage of the technique is that it can be parallelized at the pixel level.

This method is helpful for interactive visualization of dynamically changing surfaces that self-intersect. The depth-complexity of a surface's refinement is typically constant, so the time complexity of the algorithm is linear in the number of polygons when the user changes to higher-resolution views. This compares well to world-space algorithms that calculate intersections analytically but generally exhibit super-linear complexity in the number of polygons.

References

[Banks] Banks, David. *Interacting With Surfaces in Four Dimensions Using Computer Graphics*. TR93-011, Department of Computer Science, University of North Carolina at Chapel Hill. 1993. 97-107.

[Baraff] Baraff, David. "Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation," *SIGGRAPH 90 Proceedings*, 19-28.

[Fuchs] Fuchs, Henry, *et al*. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *SIGGRAPH 89 Proceedings*, 79-88.

[Mammen] Mammen, Abraham. "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique," *CG&A*, July 1989, 43-55.

[Moore] Moore, Matthew, and Jane Wilhelms. "Collision Detection and Response for Computer Animation," *SIGGRAPH 88 Proceedings*, 289-298.