NEW TECHNIQUES FOR RAY TRACING PROCEDURALLY DEFINED OBJECTS

James T. Kajiya
California Institute of Technology
Pasadena, Ca. 91125

ABSTRACT. We present new algorithms for efficient ray tracing of three procedurally defined objects: fractal surfaces, prisms, and surfaces of revolution. The fractal surface algorithm performs recursive subdivision adaptively. Subsurfaces which cannot intersect a given ray are culled from further consideration. The prism algorithm transforms the three dimensional ray-surface intersection problem into a two dimensional ray-curve intersection problem, which is solved by the method of strip trees. The surface of revolution algorithm transforms the three dimensional ray-surface intersection problem into a two dimensional curve-curve intersection problem, which again is solved by strip trees.

KEYWORDS: computer graphics, raster graphics, ray tracing, fractal surfaces, procedural modelling, strip trees, stochastic models, surfaces of revolution.

CR CATEGORIES: I.3.3, I.3.5, I.3.7

§1 Introduction

Of all synthetic images, those rendered by ray tracing stand above the rest in realism (Appel[1], Goldstein and Nagle[10], Whitted[16]). Many have disparaged its use because of its large appetite for floating point computation. However — even though ray tracing is conceded to be the slowest of all methods for rendering computer imagery — no other technique has a performance envelope quite as large. In ray tracing the combined effects of hidden surfaces, shadows, reflection, and refraction are handled with a simplicity and elegance unmatched by its competitors.

This paper is about novel ways of performing the key computational step in rendering procedural objects via ray tracing. We present new ways of computing the intersection between a ray and certain procedurally defined objects. The use of procedural objects is not new to ray tracing — Whitted[16] and Rubin and Whitted[15] have advocated their use. Indeed, one could make the claim that the natural organization for ray tracing programs is one using procedural objects.

In section two, we present a method which efficiently computes the intersection of a ray with a fractal surface. In the course of the development of this algorithm a number of results useful to other rendering techniques of fractals. We also analyze the complexity of the new technqiue.

In section three, we give an algorithm for intersecting a ray with a surface defined by translating a plane curve orthogonally. We call a surface of this type a ***prism***.

In section four, we treat surfaces of revolution. The algorithm works well even though the radius curve may be defined by thousands of points.

Algorithms for efficient rendering the above two objects are of immediate importance to CAD/CAM applications. Combined with well known methods for ray tracing Euler combinations of primitives (Goldstine and Nagel[10], Roth[14]), the above techniques facilitate rendering of models described by constructive solid geometry (CSG). Indeed, most of the objects in CSG are combinations of prisms and surfaces of revolution (Braid[4], Requicha[13]), the so called swept volumes.

In section five we present the results of our algorithms; and in section six, we discuss directions for further work.

It is likely that the locus of ideas exposed here will be

A revised version of this paper will appear in the July 1983 issue of **acm Transactions on Graphics.**

applicable to other types of objects exhibiting either a high degree of symmetry or a hierarchical organization or both. These ideas suggest a rich panoply of techniques applicable to many procedural objects which has heretofore gone unsuspected.

§2 Fractal Surfaces

The so-called method of fractals (Fournier, Fussell and Carpenter[9], Mandelbrot[11]) has generated models of startling realism. One would conjecture that applying a ray tracing rendering algorithm to objects modelled by this technique would yield very interesting images indeed. Unfortuately, practical considerations prevent one from doing this directly.

Though a simple algorithm, the fractal modelling technique generates models of visual interest through their great geometric complexity. A typical fractal surface may consist of polygons whose numbers may reach easily into the six figure range. As we shall see below, naive ray tracing is simply too slow to feasibly render a complex fractal surface.

The method we present here overcomes this problem. It does so by evolving the fractal surface in concert with the rendering of it. We generate only those parts of the surface which are likely to intersect the ray being traced. In this way, each ray need be compared with only a handful of polygons instead of the large collection of polygons making up the entire fractal surface. We compute a certain polytope, viz. an *extent*, which completely encloses the surface ultimately evolved with a high degree of certainty. If a ray intersects this extent we must inspect its contents more closely. If not, the extent is pruned from further consideration thus saving much computation.

We mention that this method is essentially the adaptive subdivision technique mentioned in (Fournier, Fussell, and Carpenter[9]) translated to the ray tracing context.

2.1 The Recursive Subdivision Method

We use the subdivision method for generating fractals (Carpenter[5], Fournier and Fussell[8], Fournier, Fussell, and Carpenter[9]). In what follows we give a brief review of the technique as presented in the references above.

The recursive subdivision technique proceeds as follows.[1] The surface is modelled as a large number of triangles. The x and y coordinates of the triangle vertices are set on an isometric grid (See figure 1), while their z-coordinates are generated recursively as follows:

> Once the displacements z_i, $i = 1,2,3$ for a given level of recursion are determined, generate three new independent random variables ξ_i, $i = 1,2,3$ with mean given by the equation
>
> $$E\xi_i = \frac{z_j + z_k}{2}$$
>
> and variance
>
> $$var(\xi_i) = \left(\frac{l_i}{2}\right)^{2H}$$
>
> where l_i is the length of the side of a triangle and H is the "fractal dimension". That is, z_i is the height of the vertex i of a triangle and the ξ_i is the displacement over the bisector of its opposite edge. These are summed to make z'_i the vertices of triangles at the next level of recursion.

2.2 The Rendering Algorithm

The rendering algorithm is very simple. Instead of instantiating the surface in its entirety, we evolve a piece at a time. There are several advantages to this.

In the ray tracing method, the intersection computation is the most time consuming step. If we were to fully evolve the fractal surface, we would need to determine the intersection point of the current ray with every polygon.[2] As a typical scene requires tracing on the order of a million rays and a fractal surface may contain the same order of magnitude of polygons, the intersection computation under the naive method would require a trillion ray-polygon intersections. Clearly this is impractical.

By evolving only a piece of the fractal surface at a time, we can cut down the number of intersections which must be computed.

The computation must intersect a ray r with a recursively defined surface. Suppose that we are able to enclose each part of the surface with an *extent*, viz. a volume which is guaranteed to contain a segment of

[1] We restrict ourselves to the case of triangularized surfaces. See Carpenter[5] and Fournier and Fussell[8], for other methods.

[2] One of the reviewers has pointed out that a fractal surface can never be *fully* evolved but only approximated to a certain level of detail

A revised version of this paper will appear in the July 1983 issue of acm Transactions on Graphics.

the fully evolved surface. The fractal surface is represented by a tree t of branching ratio four. At each node n of t we associate a pair (p, e) where p is a polygon, called a ***facet***, representing the surface at the level of recursion indexed by the depth of n, and e is an extent which encloses the surface given by the subtree at n. The leaf nodes of the tree correspond to the fully evolved fractal surface. The polygon corresponding to a leaf node shall be called a ***primitive facet***.

An an object A is said to ***shadow*** a node $n(p, e)$ with respect to a ray r if the ray intersects A at a point closer to the ray origin than the intersection with e. A node $n(p, e)$ is said to be ***active*** if its extent e intersects the ray and no primitive facet shadows it.

Note that an inactive node can never contain the closest intersection of the ray with the fully evolved surface.

The algorithm maintains a list of active nodes which are to be traced by the current ray r. With each node is associated a ***distance*** d from the origin of the ray to the closest intersection with its extent.

The algorithm proceeds as follows:

> Choose the closest node n and remove it from the active node list.
>
> Intersect with the four extents e_i, $i = 1, \ldots, 4$ associated with the children n_i, $i = 1, \ldots, 4$ of the node n.
>
> If no e_i intersects r then there are no new active nodes. Start over.
>
> Else, add the nodes whose extents e_i intersect the ray to the active node list.
>
> If the new nodes contain primitive facets p_i and the facets intersect the ray then cull from the active node list the nodes shadowed by the closest p_i. (Simply compare the distance of each node to the distance of the facet intersection point).

It may be conjectured that a node ought to be rendered inactive whenever it is shadowed by any facet (not necessarily primitive). Certainly this holds in the two dimensional case, see figure 2a. But figure 2b shows that this is false in the three dimensional case. The facet for node n shadows its neighboring extent but it contains no primitive facet doing so.

2.3 Computing Extents

The key to the performance of the above algorithm lies with the specification of extents. Beyond the necessary conditions for an extent—that they enclose the actual surface segment corresponding to a node—there are two additional desiderata which extents should satisfy. First, extents should be ***tight***, that is, they should enclose the actual surface snugly. Consider the case when extents do not, say when an extent is the whole of space. Then no pruning will ever take place: the algorithm degenerates to the naive method. As extents become more and more snug, opportunity to prune surfaces arises. Thus tight extents improve the asymptotic complexity of the algorithm. The second desideratum for an extent is that it be easily intersected with a ray. That is, one should not expend undue amounts of computation in determining whether a ray intersects an extent or not. This criterion determines the not insignificant multiplier in the complexity of the algorithm.

By the second criterion, an ideal extent for ray tracing is the sphere. Among all bounding surfaces it is the easiest to intersect with a ray. However, it fails with respect to the first criterion. Spheres do not contain fractal surfaces very snugly.

We have chosen an extent which is shown in figure 3. This extent is formed by taking the convex hull of the facet translated in z by a distance $\pm\eta$. Because of its shape we have called this object a ***cheesecake extent***. Given that η is of minimum value, this extent is relatively tight. Because it is formed from simple polygons, it is easy to determine ray intersections.

How can we determine the value of η? A fractal surface is stochastically defined. Thus it is not possible to predict with complete certainty how it will vary. On the other hand, because the statistical properties of the surface are well known, we can choose η so that there is an overwhelming probability that a cheesecake encloses the fully evolved surface.

There are two approaches we may take. The first uses the Chebyshev inequality (Chow and Teicher[6]):

Let X be a random variable with $E|X| < \infty$ and variance σ_X^2. Then

$$P\{|X - EX| \geq a\} \leq \frac{\sigma_X^2}{a^2}, \qquad a > 0$$

Now, if $S(x, y)$ is a generalized Levy surface, the variance of

$$\Delta S = S(x_1, y_1) - S(x_2, y_2)$$

is simply a function of the distance d between the points (x_1, y_1) and (x_2, y_2). Namely,

$$var(\Delta S) = d^{2H} V_H$$

A revised version of this paper will appear in the July 1983 issue of acm Transactions on Graphics.

where V_H is a constant depending on the fractal dimension H, (see corollary 3.4 of Mandelbrot and van Ness[12]).

The point of maximum distance from all the vertices of a triangle with sides of length l is the center. This distance is $\frac{l\sqrt{3}}{3}$ If we use the expression for unconditional variance, (it is clear that conditioning can only make the variance go down) then the variance is given by:

$$\sigma = \left(\frac{l\sqrt{3}}{3}\right)^{2H} V_H$$

Choosing η to be 10σ guarantees a .99 chance of the cheesecake enclosing the fully evolved surface.

If, as is usual, we know the surface follows a Gaussian distribution then one can do much better than the Chebyshev inequality. The probability that the center point extends beyond the cheesecake is then simply twice the tail of the distribution. For example, choosing η to be 3σ gives $P = .9974$. Carpenter has pointed out that if the random numbers ξ_i are generated via table lookup, then extents may be calculated with total certainty (Whitted[17]).

2.4 Analysis of the Algorithm

To analyze this algorithm we note that there are three cases of ray intersection: 1)the ray does not intersect the top level cheesecake; 2)the ray intersects the top level but not the surface; and 3)the ray intersects the surface.

If the ray does not intersect the top level cheesecake then one intersection computation is sufficient to prune the fractal. The number of rays which do not satisfy this criterion is roughly the number of pixels in which the fractal is visible (assuming extensive self shadowing does not occur).

The number of rays of the second kind is small if the extents are at all tight, to be conservative we treat these rays as if they were of the third kind.

The algorithm choses active nodes to process on the basis of distance. In the best case, only those nodes contained in the path from the root to primitive facet are chosen. At each step, new nodes are spawned and added to the active node list.

Each step computes intersections with 4 cheesecakes and may either discard or add the new nodes to the active node list. When the primitive facet is finally encountered the entire rest of the list is culled. Thus the number of intersection computations which must be done is:

$$4 \log_4 n.$$

The number of node generation steps is the same. Generating each new node is relatively cheap compared to the intersection computation.

Thus in the best case, the number of intersection calculations is

$$(\text{number of visible pixels}) \times 4 \log_4 n$$

In the worst case, the extents are able to do no pruning at all. Then we degenerate to the naive case: nm where n is the number of primitive triangles and m is the number of rays for the scene.

It has been our experience that the average case with real data possesses the same asymptotic complexity as the best case. We present some preliminary computations indicating why this should be so. In general, however, evaluating the average case complexity is problematical.

We would like to calculate the probability that a ray striking a triangular facet with a given set of angles will strike neighboring cheesecakes first, thus causing unnecessary intersection computations to be performed. In the following analysis we assume that ray strike points are distributed uniformly across a facet.

Let a ray strike a triangular facet with angle θ to the facet normal. (See figure 4). Then it must be a distance less than l' from the boundary to strike a neighboring cheesecake, where

$$l' = h \tan \theta$$

Now, projecting the ray onto the facet plane gives figure 5.

The ratio of the area of the entire triangular facet to the area of the shaded region gives the probability that a random ray will strike a neighboring cheesecake first.

This ratio may be computed by noting that the following equations hold:

$$c = l' \sin \phi$$
$$a = \frac{c}{\tan \frac{\pi}{3}} = \frac{c}{\sqrt{3}}$$
$$b = l' \cos \phi$$

A revised version of this paper will appear in the July 1983 issue of **acm Transactions on Graphics.**

The ratio of the areas is the ratio of $a+b$ to l or

$$Prob = \frac{l'}{l}(\cos\phi + \frac{\sqrt{3}}{3}\sin\phi)$$
$$= \frac{h\tan\theta}{l}(\cos\phi + \frac{\sqrt{3}}{3}\sin\phi)$$

Where h is determined as the cheesecake height of the previous section, say 4σ. So the probability of at least an extra intersection computation occuring given a certain set of angles for the incoming ray is:

$$P\{\text{Extra Computation}|\theta,\phi\} = 4\left(\frac{\sqrt{3}}{3}\right)^{2H} V_H \tan\theta$$
$$\times(\cos\phi + \frac{\sqrt{3}}{3}\sin\phi)\frac{l_{\text{neighbor}}^{2H}}{l_{\text{triangle}}}$$
$$= K l_{\text{neighbor}}^{2H}$$

The probability given a set of angles goes down exponentially for each level of recursion of the neighbors. Now, each level of recursion halves l_{neighbor}. We sum the geometric series to obtain the expected number of intersection computations required.

$$\text{Number of computations} = K\sum_{n=0}^{\text{max level}}\left(\frac{l_{\text{neighbor}}}{2^n}\right)^{2H}$$
$$\leq K\sum_{n=0}^{\infty}\left(\frac{l_{\text{neighbor}}}{2^n}\right)^{2H}$$
$$= K l_{\text{neighbor}}^{2H}\sum_{n=0}^{\infty}(\tfrac{1}{2})^{2Hn}$$
$$= K\frac{(2l_{\text{neighbor}})^{2H}}{2^{2H}-1}$$

The key observation one should make about the above calculation is that the number of expected superfluous computations for small incidence angles is very small — much less than for the worst case. In fact, the geometric series sums to a small constant making the asymptotic complexity the same as for the best case.

To find the expected number of superfluous computations we use the theorem of total probability. But here the evaluation of the number of computations becomes problematical. The distribution of angles is not a simple uniform distribution. Figure 6 shows that a facet relatively far from the ray source sees a distribution of angles which is highly peaked about the viewing angle. When shadowing and reflection occur, the distribution is far from uniform.

We have observed in practice an average performance whose general character is very good. For example, when the level of recursion was increased from 5 to 6 — quadrupling the number of polygons from 1024 to 4096 — the runtime of the algorithm increased by some 12%.

No effort has been made to cache calculated subtrees. Each ray intersection must evolve the surface anew from the top node. Whitted has suggested that caching the subtree computations can be a very useful optimization. (Whitted[17]).

§3 Prisms

A box is formed by moving a rectangle orthogonally in space, and a cylinder is formed by so moving a circle. In general, we define a ***prism*** to be a volume formed by translating a plane curve along a vector n for a distance d. The curve is defined in a plane P whose normal vector is n.

Many objects can be defined as collections of ***prisms***. Among them include: block letters, machine parts formed by extrusion, simple models of urban architecture, surfaces with "ridges"– small vertical perturbations of a plane which embellish the texture of a surface with sharp edges.

All these objects can always be modelled as collections of polygons. We consider the case where the plane curves defining these surfaces are complex, e.g. having thousands of vertices. Ray tracing such prisms defined as collections of polygons would be extremely expensive.

The technique we describe here illustrates a general technique which will be used again later. We take advantage of the essential symmetries characterizing these surfaces to reduce the intersection problem from three to two dimensions. We then solve this two dimensional ray tracing problem by using a representation for plane curves known ***strip trees*** which is popular in computational geometry (Ballard[2]).

A prism is defined as the union of three parts. The first part is the set of ***sides*** of the prism given by the locus of points $tn + \gamma(s)$. Where $\gamma(s)$ is a curve embedded in a plane P known as the ***baseplane***, n is the unit normal of P, and $0 < t < h$ where h is the ***height*** of the prism. The second and third parts of the prism are known as the ***base*** and ***cap***. The base B is set of points interior to the curve $\gamma(s)$ in P, the cap is simply the set $B + hn$.

A revised version of this paper will appear in the July 1983 issue of **acm Transactions on Graphics.**

We now give the algorithm for intersecting a ray with a prism. First find the intersection point of the ray with base and cap planes and project these points down onto the base plane. The ray itself is then projected onto the base plane. We have now reduced the problem to a two dimensional one because of the following fact. (See figure 7.)

Proposition. To intersect a ray $r(t) = o + tv$ with a prism $(B, h, \gamma(s))$, let the ray-base plane strike point be at $t = t_0$. Let the base plane projection of the ray-cap plane strike point be $t = t_1$. Then r strikes the prism iff 1) the base plane projection $r'(t)$ of $r(t)$ intersects the curve $\gamma(s)$ at a point $t = t'$ and

$$0 < |t' - t_0| v \cdot n < h$$

where n is the unit normal of the baseplane B, or 2) $r'(t_0)$ or $r'(t_1)$ is in the interior of $\gamma(s)$.

This proposition then suggests the following algorithm:

> Find the ray strike points $t = t_0$ and $t = t_1$ with the base plane B and with the cap plane $B + hn$, respectively.
>
> Project the ray r down into the base plane to give the two dimensional ray r'.
>
> Find all the intersections of the ray r' with $\gamma(s)$. (We discuss how this will be done later.)
>
> Sort the intersections by distance t from the ray origin o.
>
> Scan the sorted list and perform the actions conditioned by the following cases: 1) Cap plane strike point: if inside $\gamma(s)$ then stop else proceed. 2) Base plane strike point: if inside $\gamma(s)$ then stop else proceed. 3) A $\gamma(s)$ intersection: If the intersection point $t = t'$ is such that $0 < |t' - t_0| v \cdot n(B) < h$ then stop else proceed.

How is the intersection of the two dimensional ray r' with an arbitrary plane curve $\gamma(s)$ to be done? We use the method of ***strip trees***. Strip trees are presented in (Ballard[2]) as generalizations of structures computed in a curve digitization algorithm invented by Duda and Hart[7]. A strip tree is a hierarchical structure which represents the curve at varying resolutions.

The strip tree associated with a curve $\gamma(s)$ is a tree t with nodes $n(e, c)$, where c is a portion of $\gamma(s)$ and e is an extent which completely encloses c. An extent e is shown in figure 8, it is a triple $e = (b, w_1, w_2)$ consisting a ***baseline*** b, two widths w_1, w_2. The baseline is a line segment of arbitrary orientation. Geometrically, the extent is a rectangle whose edges are determined by the baseline and widths. The extent rectangle is chosen in such a way as to enclose the minimum area containing the curve segment c. Thus each edge of e touches at least one point of c. See figure 9. The subtrees of node n subdivide c on a point which touches the edge.

> We now give an algorithm for generating a strip tree associated with any plane curve $\gamma(s)$.
>
> Choose as a baseline the line segment connecting the first and last point of the curve. Now scan the curve for the maximum and minimum signed distance away from the baseline. These set the widths of the root triangle. Divide the curve at one of these points and compute the subtrees recursively.

It is evident that this is an $n \log n$ computation. A linear time algorithm is also available (Ballard[2]).

It is now a simple matter to efficiently intersect a two dimensional ray with a plane curve defined by a strip tree. First intersect the ray with the root extent. If there is an intersection then intersect with each of the subtrees, if not, there is no intersection with the ray. Continue recursively.

Given the above algorithms we are now able to calculate the intersection point of the ray with the prism. Calculating the normal of the surface at the intersection point is simple. If the ray strikes either the base or cap planes then the normal is the plane normal. Otherwise, it strikes the sides. The three space normal is then a simple linear transformation of the two space normal which is given by the strip tree baseline equation. This linear transformation is determined by the base plane normal.

§4 Surfaces of Revolution

Polygon or patch methods may serve as approximations to the surfaces of revolution described in this section. However, this technique can model complex surfaces which would require dozens of patches or hundreds of polygons. The tracing time of such patches or polygon collections would be high. The method described here takes advantange of the high degree of symmetry available in the model. As ray tracing algorithms go, it is relatively fast.

A surface of revolution is defined via a 3-tuple $(b, a, \rho(s))$, where b is a point called the ***base point***, a is the ***axis vector***, and $\rho(s)$ is the ***radius function***. See figure 10.

Figure 11 shows the geometrical interpretations of the following definitions. We define the ***cut plane*** as the plane containing the current ray which is parallel to the axis of revolution. Let the ray be given as an origin

A revised version of this paper will appear in the July 1983 issue of **acm Transactions on Graphics.**

and direction vector:

$$r(t) = o + tv.$$

Then the cut plane normal c is given by

$$c = a \times v.$$

The plane contains the ray origin o. Define d to be the perpendicular distance of the base point b to the cut plane.

The algorithm first reduces the problem of intersecting a ray r with a surface of revolution to the problem of intersecting a two dimensional ray r' with two plane curves γ^1, γ^2 formed by intersecting the cut plane with the surface of revolution.

Define a coordinate system in the cut plane as follows. The origin of the plane is the projection b' of the base point. The y axis vector is the projection of the axis of revolution a, the x axis unit vector is given by $a \times c$.

The algorithm now attempts to intersect the two dimensional ray with the two curves γ^1, γ^2 representing the surface meeting the cut plane. But how are these two plane curves to be determined? Figure 12 shows how the curves depend on the radius function $\rho(s)$ and the distance d, namely:

$$\gamma_x^i = \pm\sqrt{\rho_x^2 - d^2}$$
$$\gamma_y^i = \rho_y.$$

If the quantitiy under the root is negative then the curve disappears altogether.

Now, the radius curve $\rho(s)$ may contain thousands of points: it would be extremely expensive to calculate the plane curves γ^i for every ray using the above relation. Even if we do, the question of which specific algorithm to intersect the curves with the ray remains unanswered. Of course, we intend to use strip trees. But if we do so directly then we would be forced to recalculate the bounding extents at each intersection computation. This is because the plane curves change radically for each different ray. Such a scheme would be impractically slow. Strip trees only yield advantages if their extents can be precalculated and reused for each ray trace.

The solution to this problem is to trace in a different space. Specifically, we trace not in (x, y)-space but in (x^2, y)-space. In this space the equation defining the two plane curves appears as:

$$\gamma_x = \rho_x - d^2$$
$$\gamma_y = \rho_y$$

The original radius curve ρ is defined as the square of the actual radius curve. The plane curve γ is simply translated by the square of d the distance of the cut plane from the base point.

Figures 13a,b show examples of this transformation. As the distance of the cut plane from the axis grows, more of the curve lies below the axis. Note that we are now tracing curved rays that bounce off the origin. Thus the parts of curves below the axis are inaccessible to the ray. To trace curved rays we use a slightly different ray equation. Instead of tracing rays defined as

$$r = o + tv$$

we now trace rays defined as

$$r_x = (o_x + tv_x)^2$$
$$r_y = o_y + tv_y$$

We are now charged with intersecting a curved ray defined as above with an arbitrary plane curve. Fortunately, this plane curve is translated by d so that now it is a simple matter to use strip trees. We recursively intersect the curved ray with an extent box. Each extent intersection is easily acomplished by straightforward solution of a polynomial with at most quadratic degree in t. Solving for t gives the distance from the ray origin. Substituting into the original vector form for the ray gives the exact intersection point.

The above algorithm then determines intersection point of a ray with a surface of revolution. To complete the ray tracing process we need to compute the surface normal. This is done in two steps.

The first step is to translate the slope of the plane curve in bent (x^2, y)-space back into a plane normal in flat (x, y)-space. This we do by using the following equations. Let i_x be the x-coordinate of the intersection (in flat two space), and l_x, l_y the components of the normal vector in bent two space given by the baseline line equation in the strip tree defintion, then the normal vector (n_x, n_y) expressed in flat two space coordinates is:

$$n_x = \frac{2l_x i_x}{\sqrt{(2l_x i_x)^2 + l_y^2}}$$
$$n_y = \frac{l_y}{\sqrt{(2l_x i_x)^2 + l_y^2}}$$

A revised version of this paper will appear in the July 1983 issue of acm Transactions on Graphics.

The second step is to translate the plane normal into space normal using the geometry shown in figure 14. The space normal is given by:

$$n = n_x i_x c \times a + n_x(-d)c + |x|n_y a.$$

We have now calculated the intersection point and the normal to the surface at that intersection point for a surface of revolution. This is all that is required for rendering the surface.

§5 Results

The above algorithms were coded in FORTRAN on a DECSYSTEM-2060. The actual fractal algorithm performed differs from the presented one in that no selection of closest extents occurs. The facets are expanded and pruned in a strict depth first manner with fixed ordering for the four subnodes. Evidently, the programmed algorithm is slightly simpler at the cost of increased computation time.

Figure 15 shows a sample output of the algorithm. A scene composed of a reflecting sphere placed above a fractal "valley" evolved from a single initial triangle with vertices $(-1, -\frac{\sqrt{3}}{3}, .5), (1, -\frac{\sqrt{3}}{3}, .5), (0, \frac{\sqrt{3}}{2}, 0)$. Note that in the reflection of the sphere the back side of the mountain can be seen to shadow itself.

This image, computed at a resolution of 256 × 256 pixels, consumed 190 minutes of CPU time.

Figure 16 shows an example of the prism algorithm. In this scene, an "E" shaped curve is translated along the z-axis. Two reflecting spheres are placed in proximity with the prism.

An example of the surface of revolution algorithm is shown in figure 17. This scene shows a wine glass resting on a single triangle. The wine glass is defined as a base point b on the surface of the triangle, an axis of revolution coincident with the z-axis, and a radius curve of only 9 points. This points out an interesting feature of the algorithm. Because the radius curve is defined in bent two space, straight lines translate into parabolas. Thus the curvature of the glass is due entirely to the curvature of the coordinate system. This image consumed 20 minutes of CPU time.

§6 Further Work

There are several obvious extensions of the above algorithms which allow for other primitives.

A modification of the fractal rendering algorithm will work for arbitrary ***deterministic*** height fields defined on triangular grids. Many objects in computer graphics may be so defined: human faces, digitized terrain, planar polygons with detailed surface features, injection molded objects, etc. In fact, this algorithm is applicable to scan-line rendering methods as well as ray tracing.

A ***subdivision tree*** is constructed by recursively subdividing the height field. Each node of the tree stores an extent enclosing a section of the field. Once such a tree has been constructed, we use the fractal rendering algorithm. The algorithm operates in exactly the same manner as above except that the tree is fully instantiated before rendering. Generating the subdivision tree is similar to the strip tree algorithm.

> Recursively execute the following step:
>
> Scan the surface defined by the domain triangle, searching for the point of maximum and minimum distance η_1, η_2 to the triangle containing the vertices. Store the extent cheesecake with distances η_1, η_2. Subdivide the domain triangle into four subtriangles.

Note that the complexity of constructing this tree is $n \log_4 n$ where n is the number of points in the height field. A linear time algorithm which computes somewhat larger cheesecake extents is also possible by constructing the tree in a bottom up fashion. Both these algorithms construct bivariate analogs of strip trees and essentially mimic the strip tree algorithms (Ballard[2]).

A further modification of the height field algorithm follows from the observation that objects other than polygons can serve as leaf nodes in the tree. Such a tree, with extents given by procedural definitions, is easily seen to be an elaboration of the well known method of Rubin and Whitted[15].

It may well be that extents other than cheesecakes would be more effective for the fractal rendering algorithm. For example, it is easy to argue that ellipsoids would be tighter extents than cheesecakes. The conditional variance of a fractal segment is maximum in the center of the fractal and diminishes as it approaches the vertex of a facet. Ellipsoids also exhibit this behavior but cheesecakes do not.

To construct an ellipsoid extent, one would choose two of the three ellipsoid principal axes to be in the plane of the triangle. The principal lengths would be set to cover the vertices. The third principal axis should be vertical with principal length sufficient to cover $\pm\eta$ in the center of the ellipsoid.

A revised version of this paper will appear in the July 1983 issue of **acm Transactions on Graphics.**

It should be easy to calculate intersection points between a ray and an extent. And, certainly, intersecting a ray with an ellipsoid is almost as easy as intersecting one with a sphere. Let the matrix A be formed using the principal axes as rows. Intersecting a ray r with an ellipsoid is equivalent to intersecting a ray $A^{-1}r$ with a sphere.

The prism and surface of revolution algorithms may be extended in a number of ways. One extension is to allow curves other than simple line segments to reside in the leaves of the strip tree. There are two requirements that such new leaf curves should satisfy. First, they should be completely enclosed by a suitable rectangular extent. Second, it should be relatively easy to intersect a ray with these primitive curves. Useful primitives are: circular arcs, parabolas, and algebraic curves.

Both algorithms may also be generalized by linearly transforming the incoming ray. In this way surfaces of revolution may have elliptical instead of circular cross sections. In addition, this technique can model skewed surfaces of revolution and skewed prisms.

Note also that the methods presented here may be applied in a mutually recursive manner to efficiently render objects of potentially high complexity. We can best explain what we mean by an example. Suppose the task is to model an office building. One way to do this would be to model it as a prism where each face of the prism is the base plane of a collection of smaller prisms each of whose faces is, say, a deterministic height field. The rendering of this building would recursively invoke the algorithms presented above.

Of course, the normal computation of the surface is subject to the now standard techniques of Phong smoothing, and Blinn perturbation mapping. The prism and surface of revolution algorithms also allow a cheaper version of these techniques to be applied to the two dimensional normals as well. Thus, it is possible to generate, say, smoothly reflecting and refracting prisms even though the boundary curves may be coarsly polygonal.

ACKNOWLEDGMENTS. I would like to thank Howard Derby for many discussions and much help during the course of this investigation. I also thank the SIGGRAPH and TOGS reviewers for suggesting many useful changes.

§7 References

1. APPEL, A. Some Techniques for Shading Machine Renderings of Solids. SJCC (1968) 37-45.

2. BALLARD, D.H. Strip trees: a hierarchical representation for curves. *Comm. ACM*, 24 (May 1981) 310-321.

3. BLINN, J.F. Simulation of wrinkled surfaces. *Computer Graphics* 12 (August 1978) 286-292.

4. BRAID, I.C. Designing with Volumes, Ph.D. Dissertation. Univ. Cambridge, England (1973).

5. CARPENTER, L.C. Computer rendering of fractal curves and surfaces. *SIGGRAPH80 Conference Proceedings Supplement* (August 1980).

6. CHOW Y.S., TEICHER, H. *Probability Theory: Independence, Interchangeability, Martingales.* Springer Verlag, Heidelberg (1978).

7. DUDA, R.O. AND HART,P.E. *Pattern Classification and Scene Analysis.* Wiley-Interscience, New York (1973).

8. FOURNIER, A., FUSSELL, D. Stochastic modelling in computer graphics. *SIGGRAPH80 Conference Proceedings Supplement* (August 1980).

9. FOURNIER, A., FUSSELL, D. AND CARPENTER, L. Computer rendering of stochastic models. *Comm. ACM*, 25 (June 1982) 371-384.

10. GOLDSTEIN, E. AND NAGLE, R. 3D visual simulation *Simulation* 16 (Jan 1971) 25-31.

11. MANDELBROT, B. *Fractals: Form, Chance, and Dimension.* W.H. Freeman, San Francisco(1977).

12. MANDELBROT, B. Fractional brownian motions, fractional noises and applications. *SIAM Review* 10,4 (October 1968) 422-437.

13. REQUICHA, A.A.G. Representations for rigid solids: theory, methods, and systems. *ACM Computing Surveys* 12 (December 1980) 437-464.

14. ROTH, S.D. Ray casting for modeling solids. *Computer Graphics and Image Processing* 18 (1982) 109-144.

15. RUBIN, S. AND WHITTED, T. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics* 14 (1980) 110-116.

16. WHITTED, T. An improved illumination model for shaded display. *Comm. ACM*, 23 (June 1980) 343-349.

17. WHITTED, T. *private communication* (1983).

A revised version of this paper will appear in the July 1983 issue of acm Transactions on Graphics.

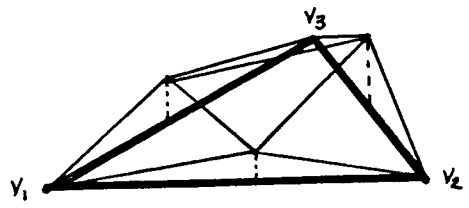

Figure 1. **Fractal subdivision.** The midpoint of each edge of the triangle is displaced in z by a random variable.

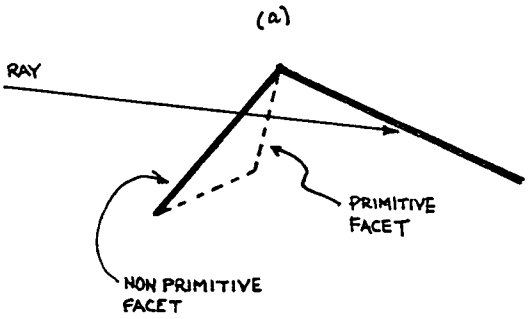

Figure 2. **Shadowing by nonprimitive facets.** (a) In the two dimensional case, non-primitive facets shadowing other facets is a sufficient condition to cull them from further consideration. (b) In the three dimensional case a non-primitive facet may shadow another at a given stage of evolution but later stages (shown as dotted lines) may uncover the shadowed facet.

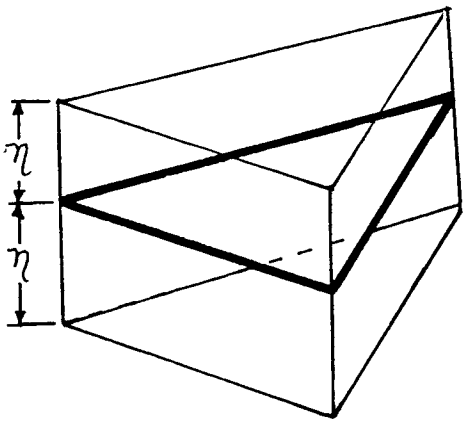

Figure 3. **A cheesecake extent.** This extent is formed by the volume swept out by translated a facet in z by $\pm\eta$.

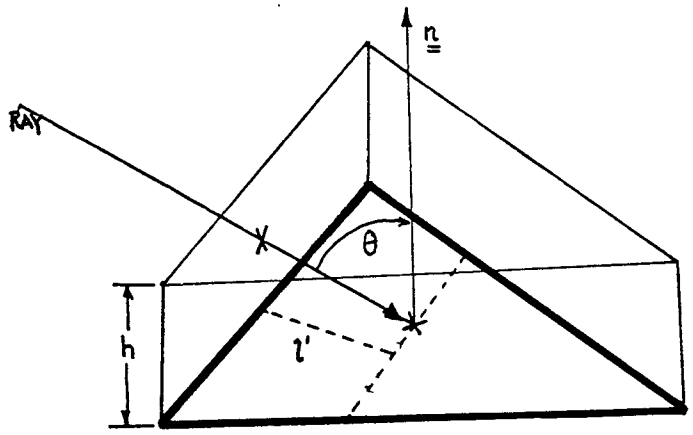

Figure 4. **Estimating wasted computations.** If an incoming ray strikes a facet at a small enough angle θ, it will miss the sides.

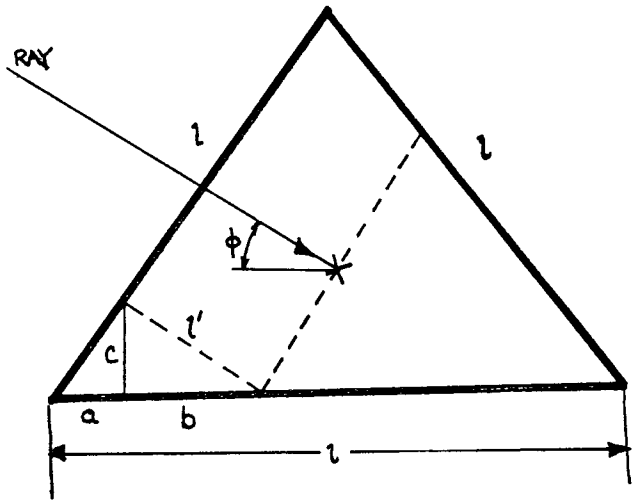

Figure 5. **Estimating wasted computations.** This figure shows a top view of figure 4. We use it to calculate the area of the shaded region.

A revised version of this paper will appear in the July 1983 issue of **acm Transactions on Graphics.**

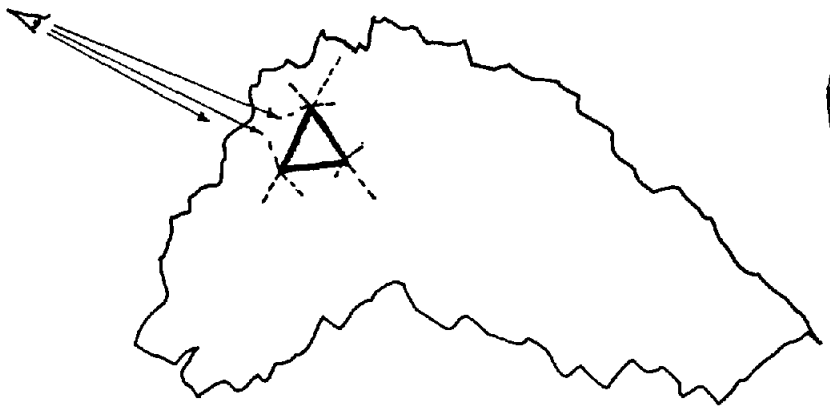

Figure 6. **Estimating wasted computations.** Under typical viewing conditions, the angle of ray-facet intersections is hardly uniformly distributed.

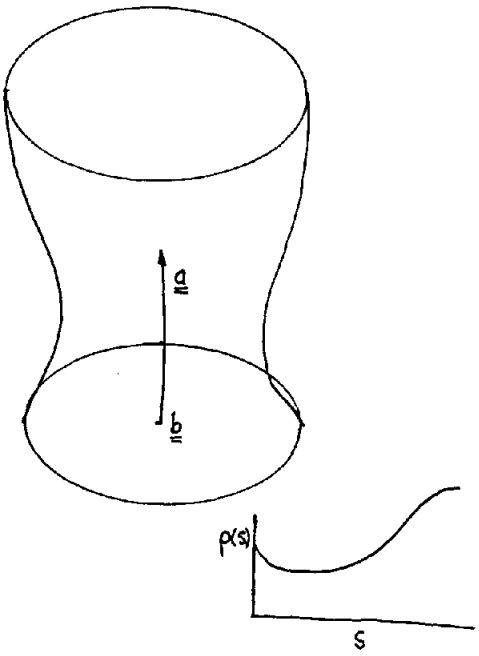

Figure 10. **A surface of revolution.** The surface is defined by a base point b, an axis vector a, and a radius function $\rho(s)$.

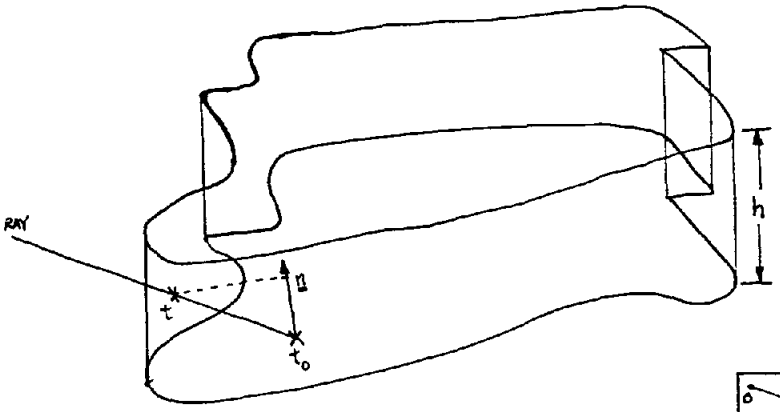

Figure 7. **A prism.** The ray is shown striking the side of a prism of height h with base plane normal n.

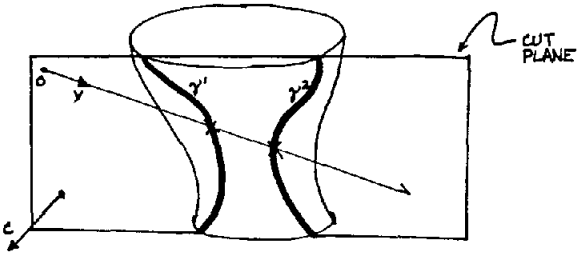

Figure 11. **Computing the intersection.** The cut plane contains the ray and is parallel to the axis of revolution. It cuts the surface in two curves.

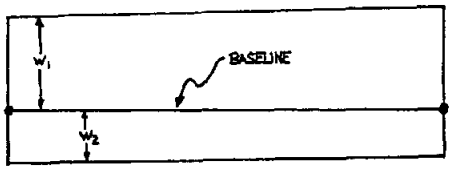

Figure 8. **A strip tree extent.**

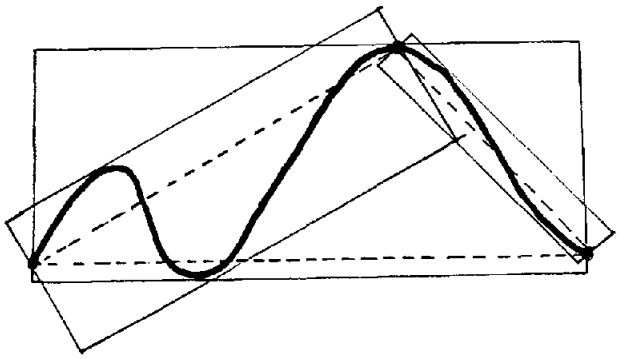

Figure 9. **A strip tree.** Rectangular extents completely enclose the bold curve.

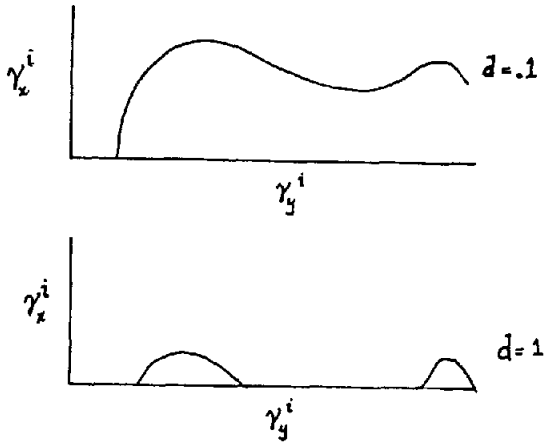

Figure 12. **Intersection curves.** The cut plane intersection curves vary dramatically in shape with the cut plane to base point distance.

A revised version of this paper will appear in the July 1983 issue of **acm Transactions on Graphics.**

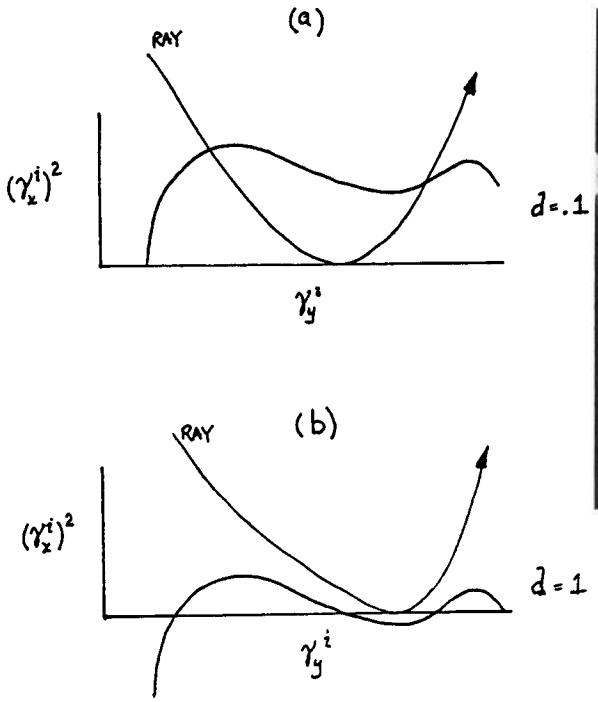

Figure 13 **Intersecting a ray in bent space.** (a) The bent space ray is now curved. The two intersection points represent intersections with the different γ_z^i. (b) Changing the cut plane to base point distance simply translates the radius curve.

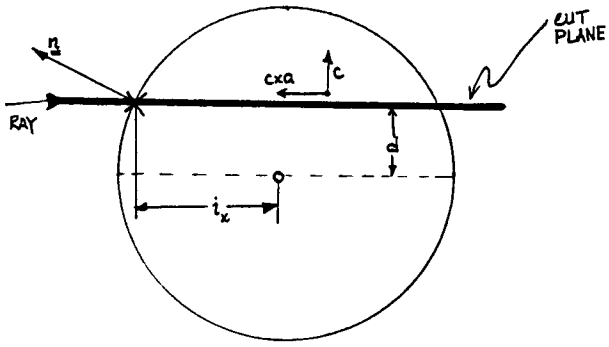

Figure 14. **Calculating the three space normal.** This is a top view looking down the axis vector.

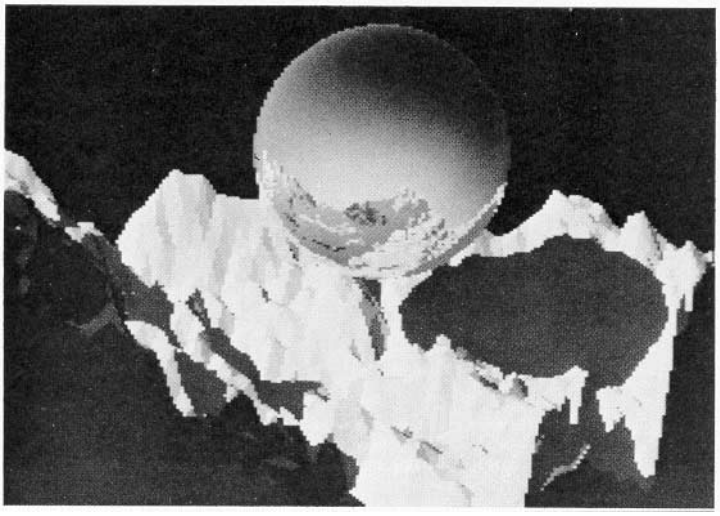

Figure 15. **A fractal surface.** The surface is shadowed by and reflected in a mirrored sphere. Note that in the reflection, the surface can be seen to be shadowing itself.

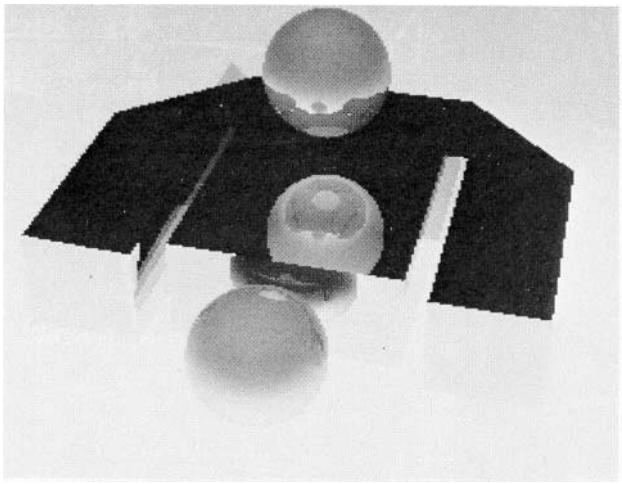

Figure 16. **A prism.** The prism is surrounded by two mirrored spheres.

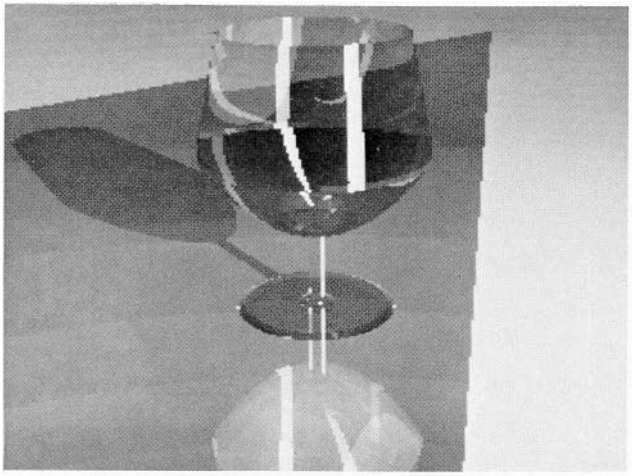

Figure 17. **A surface of revolution.** This glass is defined by only 18 real numbers.

A revised version of this paper will appear in the July 1983 issue of **acm Transactions on Graphics.**