Ray Tracing Complex Models Containing Surface Tessellations

John M. Snyder
Alan H. Barr
California Institute of Technology
Pasadena, CA 91125

Abstract

An approach to ray tracing complex models containing mathematically defined surfaces is presented. Parametric and implicit surfaces, and boolean combinations of these, are first tessellated into triangles. The resulting triangles from many such surfaces are organized in a hierachy of lists and 3D grids, allowing efficient calculation of ray/model intersections.

The technique has been used to ray trace models containing billions of triangles and surfaces never before ray traced. The organizing scheme developed is also independently useful for efficiently ray tracing any complex model, whether or not it contains surface tessellations.

KEYWORDS: Ray tracing, parametric surface, tessellation, triangle, list, 3D grid

1 Introduction

In the past, models suitable for ray tracing have contained too few and too simple primitives. Much work has been focused on solving these two problems independently.

To extend the set of "ray-traceable" surfaces beyond polygons and quadric surfaces, several schemes for intersecting rays with surfaces have been developed. Kajiya [Kajiya 82] has described an algorithm for ray tracing bicubic patches. Toth [Toth 85], Barr [Barr 86], and Joy and Bhetanabhotla [Joy 86] have studied algorithms for intersecting rays with general parameteric surfaces. These schemes are slow, require expensive evaluation of surface parameterizations, and are hard to robustly implement.

Alternatively, mathematically defined surfaces can be broken down into simple pieces. The resulting tessellation is an approximation to the real surface which can be made arbitrarily close to it by using tiny enough pieces. This approach has been avoided because ray tracers were unable to handle the vast numbers of primitives needed to approximate a surface.

Recently, organizing structures for large and complex collections of primitives have been proposed which make feasible ray tracing of models containing many fine tessellations. These structures fall into three categories — lists, octrees, and 3D grids. Each organizes a collection of objects into a single unit which may later be incorporated into a higher level structure. Each allows the ray tracing algorithm to determine which objects in the collection can potentially be intersected by a ray.

 ACM-0-89791-227-6/87/007/0119 $00.75

Lists were used in early ray tracers such as developed by Rubin and Whitted [Rubin 80]. A list is simply a grouping of objects. Hierarchies are built by grouping lists into higher level lists. Kay and Kajiya [Kay 86] investigated an algorithm to traverse the list hierarchy so that objects are considered in the order that they occur along the ray. This requires sorting of objects that can potentially be intersected by the ray.

Octrees and 3D grids partition space rather than objects and thus avoid object sorting. In these structures, each cell, a rectangular volume in space, contains all the objects that occur within it. The difference between the two structures is that octrees are hierarchical with variable sized cells, while 3D grids are nonhierarchical with cells of uniform size. Glassner [Glassner 84] and Kaplan [Kaplan 85] investigated octrees. Fujimoto, et al. [Fujimoto 86] developed 3D grids and compared their efficiency with octrees.

Fujimoto found the 3D grid structure superior to an octree for ray tracing models containing large numbers of primitives homogenously scattered through space. This finding can be explained in light of two 3D grid properties. First, because grid cells are of uniform size, tracing a ray from one grid cell to the next is an extremely fast, incremental calculation. Second, because grids are nonhierarchical, determining which cell contains the ray origin can be done in constant time, while the same operation is logarithmic in the number leaf cells in an octree. In fact, both lists and octrees require hierarchy traversal; lists through a hierarchy of bounding volumes around objects, and octrees through a hierarchy of octree cells. Set up time for a ray/grid intersection is large, however, making it impractical for collections of a few objects. A single grid is also impractical for organizing objects at widely varying lengths of scale.

The proposed algorithm uses a hybrid, hierarchical approach to organizing a complex model. In it, both lists and 3D grids are used to organize model elements, which are primitives, or themselves lists or 3D grids. Grids are used to organize large collections that are evenly distributed through space. Lists are used to organize small collections that are sparsely distributed through space. This scheme can adapt to complexity in a model at many scales; in fact, a hierarchy of 3D grids can be viewed as a generalization of an octree, in which arbitrary branching ratios are possible instead of a fixed branching ratio of eight.

Using this technique, we have ray traced a model containing 400 billion triangles, more primitives than have previously been rendered into a single image. We have generated complex images containing such shapes as teapots, grass blades, clover leaves, flower petals, and bumpy, twisted, and self-intersecting parametric surfaces. In short, this technique has established a new state of the art in the complexity of ray traced images.

2 Surface Tessellation

A surface *tessellation* is a connected mesh of pieces which approximates the surface. A *triangle* is the tessellation piece; a surface is thus approximated by a polyhedron with triangular faces. Triangles were chosen because their simplicity allowed fast con-

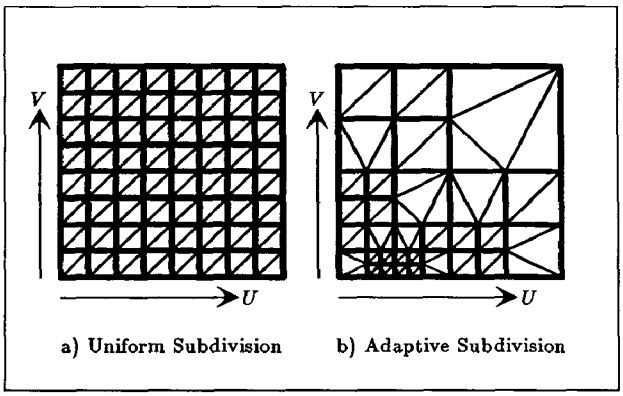

Figure 1: Parameter Space Tessellated Into Triangles

struction of surface tessellations and fast ray intersection with the mesh pieces (see Appendix).

Tessellation of surfaces is accomplished in a program separate from the ray tracer. Currently, this program tessellates several types of parametric surfaces $S(u,v)$: $R^2 \rightarrow R^3$ by uniformly sampling a rectangular region of parameter space using a specified number of divisions in u and v (see Figure 1a). Each set of four adjacent samples is then used to create two triangles. Surfaces can also be tessellated using adaptive sampling techniques (see Figure 1b) in which the fineness of the subdivision can vary over the parameter space. A technique for adaptive subdivision of parametric surfaces and boolean combinations of parametric surfaces is discussed in [Von Herzen 85]. In addition, a technique to tessellate implicit surfaces is currently being developed at Caltech [Kalra 86].

2.1 Tessellation Artifacts And Solutions

Experimentation has shown that surface tessellations can be ray traced without noticeable artifact. The organizational scheme can easily handle tessellations which are fine enough so that no silhouette or shadow polygonal segmentation is visible. Moreover, as the number of triangles in a tessellation is increased, the time to ray trace the tesselation grows slowly (see Section 6). In practice, while the algorithm had the capability to ray trace tessellations containing many more triangles in an allotted rendering time, some surfaces in very complex models remained inadequately tessellated because of memory limitations (typically about 16 megabytes). Visible artifacts were the result.

Artifacts take the form of polygonal shading facets, silhouettes, and shadows. Polygonal shading facets are largely controlled by using normal interpolation across triangles. Silhouette and shadow artifacts are most pronounced in regions where the surface has high curvature, and in regions where triangles in the tessellation have long edges. One solution is to tessellate the surface adaptively using variation in normal vector, and linear length of triangle edge as criteria for subdivision. In this way, parts of the surface requiring further sampling may be more finely tessellated without increasing the overall number of triangles.

Information concerning how the surface is positioned with respect to the camera, the lights, and other surfaces can also be used in a subdivision scheme to reduce artifacts. For example, given an eye position, surfaces can be subdivided more in regions where the normal to the surface is nearly perpendicular to the direction to the eye. Silhouette edges of objects will then appear less choppy. This approach has not been pursued because it depends on properties not inherent in the surface, creating tessellations which are

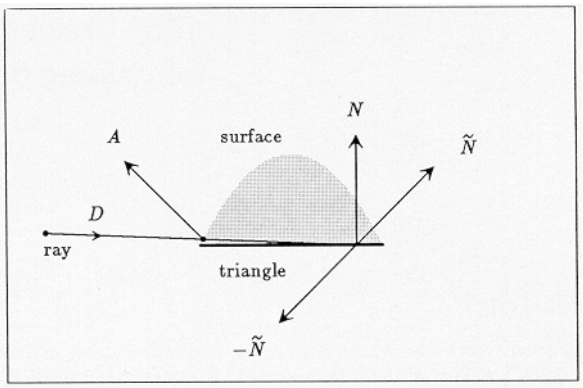

Figure 2: The Surface Sidedness Problem — The triangle normal vector N faces slightly toward the ray origin ($N \cdot D \leq 0$ implies outside intersection) but the interpolated normal vector $\tilde{N}$ faces away ($\tilde{N} \cdot D > 0$ implies inside intersection). The vector A is the normal to the surface at the intersection of the ray with the surface; it clearly indicates that the ray intersects the outside of the surface.

only good in a particular scene. Also, determining the location of silhouettes and shadows is complicated by ray tracing effects such as reflections, refractions, diffuse shadows, and depth of field.

Artifacts can also be reduced by intelligent shading and ray casting techniques as well as by intelligent subdivision. The following sections describe two examples.

The Surface Sidedness Problem

When a ray intersects a triangle, the triangle's interpolated normal, $\tilde{N}$, is passed to the shader as the actual normal to the surface at the point of intersection. Let the ray be parametrized by $O + Dt$ where t is a scalar greater than 0 and O and D are vectors; O is the ray origin, and D, the ray direction. The vector $\tilde{N}$ is used to determine whether the inside or outside of the surface was hit according to the sided intersection test:

$$\tilde{N} \cdot D \leq 0 \implies \text{outside intersection}$$
$$\tilde{N} \cdot D > 0 \implies \text{inside intersection}$$

For an inside intersection, $\tilde{N}$ is flipped (scaled by -1), since the intersection algorithm always returns outward facing normals. The final $\tilde{N}$, flipped or unflipped, is used by the shader to compute diffuse (Lambert) shading, specular highlights, and directions for recursively generated reflection and refraction rays.

Let N be the normal to the plane embedding the triangle. It is the normal of the polyhedral tessellation of the surface at the point of intersection, whereas $\tilde{N}$ is an approximation to the actual (pre-tessellated) surface's normal. A problem arises if the result of the sided intersection test is different when applied to $\tilde{N}$ and N, as in Figure 2. Large shading discontinuities result when $\tilde{N}$ is erroneously flipped in this situation since it indicates an inside intersection. As Figure 2 shows, the unflipped $\tilde{N}$ is also a bad approximation to the actual surface normal.

Experimentation has shown that this problem can be made less severe by using the actual triangle normal N instead of $\tilde{N}$ whenever the sign of $\tilde{N} \cdot D$ is not equal to the sign of $N \cdot D$. For outside intersections (as in Figure 2), the normal must have some component toward the ray origin, rather than away from it. Thus N is closer to A than is either $\tilde{N}$ or $-\tilde{N}$. Although not a completely

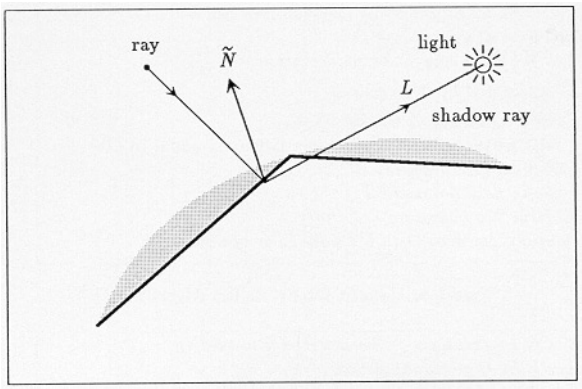

Figure 3: The Terminator Problem — Although $\tilde{N}$ indicates that a surface point is not self-shadowed, (i.e. $\tilde{N} \cdot L \leq 0$ where L is opposite to the shadow ray direction) the shadow ray is launched starting from inside the surface. The point is therefore found to be in shadow when the shadow ray intersects the surface on its way out. This problem manifests itself in a polygonal, segmented terminator, particularly evident with point light sources.

satisfactory solution, this technique restored the brightness of a few unnaturally dark pixels along silhouettes of highly curved surfaces in experimental pictures. It is expected that adaptive surface tessellation which samples more highly in regions where the surface normals vary largely, coupled with this technique, will be an effective solution to surface sidedness artifacts. Such adaptive sampling will limit the maximum angular difference between N and A.

The Terminator Problem

A terminator is an area on a surface separating lit and self-shadowed areas. Let L be the direction of the light from a point on the surface. Whether or not a point on a surface is self-shadowed is determined according to the self-shadowing test:

$$\tilde{N} \cdot L \geq 0 \implies \text{potentially lit}$$
$$\tilde{N} \cdot L < 0 \implies \text{self-shadowing}$$

As in the previous case, artifacts occur in regions where the result of the self-shadowing test is different when applied to the actual triangle normal N and the interpolated normal $\tilde{N}$. In the case that N indicates that an intersection point is lit, and $\tilde{N}$ indicates that it is in shadow, we can use the solution discussed for the surface sidedness problem. Merely substituting the actual triangle normal N for the interpolated normal $\tilde{N}$ in subsequent shading calculations reduces terminator artifacts. In the case that N indicates that a surface point is in shadow and $\tilde{N}$ indicates that it is lit, as in Figure 3, a different solution is required. Here, the problem is that the shadow ray is launched from inside the surface so that the point is always in shadow, even though the actual surface point may be lit.

To solve this problem, the shadow ray is launched further from the point of intersection so that it can "escape" to the outside of the surface. Ray tracing algorithms incorporate a tolerance, called the shadow tolerance, which controls how far from the point of intersection to shoot the shadow ray. For most surfaces, simply making this number a parameter of the surface instead of a global constant eliminates terminator artifacts. When the shadow tolerance can not be made large enough over the whole surface to eliminate terminator artifacts without simultaneously creating other artifacts, a variable shadow tolerance can be used. Figure 4 shows one such scheme.

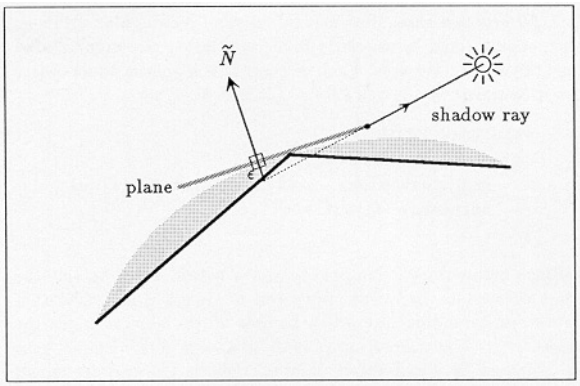

Figure 4: Solving The Terminator Problem Using Variable Shadow Tolerance — To allow the shadow ray to escape from inside the surface, we can shoot the shadow ray starting from its intersection with a plane ϵ from the point of intersection along the interpolated normal direction $\tilde{N}$.

3 Organization Of The Model

Once the modeler has tessellated surfaces in the model into triangles, he must organize these triangles and other model components using lists and 3D grids. This organization takes place during the preprocessing phase. When preprocessing is complete, the model may be ray traced to generate an image. This section describes the structure of lists and 3D grids, and their use in organizing complex models. Section 4 describes how model components are inserted into lists and grids during preprocessing. Section 5 describes how the preprocessed model is ray traced.

A component of a model is called an *object*. The following is the C language definition of an object:

```
structure object {
    double bounding_box[3][2];
    structure transformation *trans;
    char *root_object;
    int object_type;
}
```

Each object can be transformed using an 3×3 transformation matrix and a 3×1 translation vector, pointed to by the `trans` field. Each object is also bounded by a simple box formed by three pairs of extents in the x, y, and z directions in the `bounding_box` field. The `root_object` field is a pointer to a structure containing parameters of a specific object, called the *root object*. Examples of root objects are polygons, spheres, cylinders, triangles, lists, and 3D grids. The `object_type` field indicates the type of the root object.

3.1 Structure Of Lists And 3D Grids

A *list* is a linked list of objects. Its C language definition is

```
structure list {
    structure object *list_object;
    structure list *next;
}
```

A *3D grid* is a three dimensional array of rectangular volumes, called *cells*, formed by regularly dividing a larger rectangular solid along the coordinate axes. Each cell contains a pointer to an object that is bounded by the cell extents. It is defined as:

```
structure grid {
    double grid_extent[3][2];
    int x_divisions, y_divisions, z_divisions;
    structure object *cells[];
}
```

Since many objects can occupy space within a cell extent, the object pointed to by a nonempty cell is always a list. This list has its own bounding box which bounds all the objects inside the grid cell. Its transformation pointer is always null. Empty cells are indicated by a null object pointer. Cells in the grid are stored in the `cells` field. The `grid_extent` field stores the extent of the volume that was divided into cells using `x_divisions` x divisions, `y_divisions` y divisions, and `z_divisions` z divisions.

3.2 Building the Model with Lists and Grids

The modeler specifies lists and 3D grids by opening a list or 3D grid and inserting a series of objects into it. Only one list or grid can be open at a time. When a grid is opened, the modeler specifies the number of x, y, and z divisions in the grid, and the x, y, and z extent of the grid. Opening a list requires no parameters. The specification of a list or grid also includes a unique name so that lists or grids built by the modeler can later be instantiated into other lists and grids. The entire model is hierarchically built in a bottom-up fashion using instantiation.

Triangles in a single surface tessellation are usually inserted into a single grid. This grid can then be instantiated many times in the model, and can be separately tranformed in each instance. This is accomplished by creating several objects whose `root_object` fields all point to a single copy of the grid, but whose `trans` field point to different transformation structures. In the same way, the modeler can also replicate lists by multiple instantiation.

Model building is currently a heuristic, modeller directed process. More work still remains to develop fully automatic algorithms that can organize complex models for efficient ray tracing. On the other hand, lists and 3D grids often naturally fit the model's organizational structure. For example, our model of a grassy plain (see image in Section 6) is a list containing a plain polygon and a grass field grid. The grass field was hierarchically constructed using two different grass blade surface tessellations. First, a grass patch was built by replicating these two blades many times with various rotations, scales, and translations and inserting them into a grid. Two of these patches were then replicated and inserted into a larger grid to form a field of grass. Fields were then replicated into a grass plain. In this way, without much modeler effort, we constructed a very complex model (4×10^{11} triangles) which could be ray traced quite quickly (12 hours on an IBM 4381).

4 Preprocessing Algorithm

Figure 5 describes the "generic" algorithm to insert a object into a list. The term "generic" is used because the algorithm works for any object that can be bounded in a simple xyz extent bounding box. Figure 5 refers to transforming and enlarging bounding boxes. A bounding box is transformed by transforming each vertex of the original bounding box, and bounding the result in x, y, and z. A bounding box is enlarged by another bounding box with simple maximum/minimum operations to produce a bounding box that

Let O be an object to be inserted into list L
Let B be O's bounding box
Let T be the current transformation
Transform B by T to give $\dot{B}$
Create an object $\hat{O}$ *whose*
  `root_object` *and* `object_type` *fields are equal to O's*
  `bounding_box` *field is* $\dot{B}$
  `trans` *field points to T*
Enlarge L's bounding box by $\dot{B}$
Add a pointer to $\hat{O}$ *to L's linked list of objects*

Figure 5: Generic Object Enlist Algorithm

Let O be an object to be inserted into grid G
Let B be O's bounding box
Let T be the current transformation
Transform B by T to give $\dot{B}$
For each cell in G within or intersecting $\dot{B}$ *Do*
  clip $\dot{B}$ *to this grid cell yielding a bounding box* $\ddot{B}$
  create an object $\hat{O}$ *whose*
    `root_object` *and* `object_type` *fields equal O's*
    `bounding_box` *field is* $\ddot{B}$
    `trans` *field points to T*
  add $\hat{O}$ *to the cell's object list, creating this*
  list if the cell was previously empty
Endfor

Figure 6: Generic Object Engrid Algorithm

bounds both. Figure 6 describes the generic algorithm for inserting an object into a grid.

The generic algorithms work for any primitive. Several optimizations can be made, however, to speed ray tracing of triangle and polygon primitives. First, instead of transforming objects by inverse transforming the ray (see Section 5), we can transform the primitives directly during preprocessing. This avoids many ray transformations and yields tighter bounding boxes around the primitives, allowing the ray tracing algorithm to cull more objects from ray intersection consideration.

Second, instead of clipping the primitive's bounding box to each grid cell, the primitive itself can be clipped as in Figure 7 [1]. This yields tight bounding boxes around the triangles and polygons inside of every grid cell, and appropriately ignores grid cells which intersect the bounding box but not the primitive inside. The bounding box of the primitive inside a grid cell becomes its bounding box after clipping to the grid cell's extents. On the other hand, the object inserted into the grid cell's list is still the original unclipped triangle or polygon. The unclipped primitive is inserted to conserve memory since only one copy of a triangle or polygon is stored instead of several clipped versions of the same thing. It is also more efficient to intersect a ray with a triangle than to intersect with the many-sided polygon that may result from clipping a triangle to a volume.

5 Ray/Model Intersection Algorithm

To intersect a ray with an object, the algorithm first determines if the ray intersects the object's bounding box (see Figure 8) [2]. If

[1] [Cyrus 78] discusses clipping polygons to convex volumes.

[2] The ray/bounding box intersection algorithm is adapted from that found in [Kay 86] to avoid intersections with planes behind the ray origin.

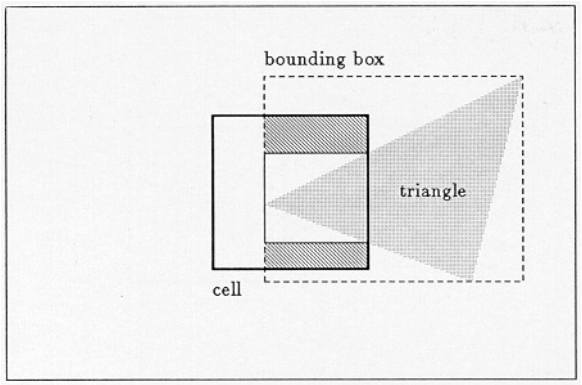

Figure 7: Clipping a Triangle to a Grid Cell — Clipping the triangle yields a bounding box that is smaller (by the diagonally shaded areas) than intersecting the triangle bounding box (dashed lines) with the cell extent.

it does, and the object's **trans** field is non-null, then it transforms the ray. Let A be a 3×3 matrix and B a 3×1 vector that transforms a point P to P' via $P' = AP + B$. Intersecting a ray $O + Dt$ with an object so transformed is equivalent to intersecting the untransformed object with a transformed ray $O' + D't$ where [3]

$$\begin{aligned} O' &= A^{-1}(O - B) \\ D' &= A^{-1}(D) \end{aligned}$$

The transformed ray and the root object pointed to by the object's **root_object** field are then passed to the intersection routine for the specific type of root object. If the root object is a primitive (e.g. a triangle) then this routine computes the ray/root object intersection directly. If the object is a list or 3D grid, then the routine traces the ray through the structure, recursively calling the ray/object intersection routine for individual objects it contains.

5.1 Tracing A Ray Through A List

Computing the intersection of a ray with a list can be accomplished by performing a ray/object intersection (defined in Section 5) on each object in the list. The intersection that occurs at the minimum t parameter of the ray is the desired frontmost intersection.

Alternatively, ray/bounding box intersections can be computed on each object first, so that objects whose bounding box is intersected by the ray can be sorted in increasing order of the ray's $t^{\min}$ intersection with the object's bounding box (see [Kay 86]). Then, when a ray/root object intersection is computed at some t, the algorithm can eliminate any root objects whose $t^{\min} > t$.

The implementation allows the modeler to specify whether sorting takes place in any list. The sort algorithm used is a simple linear insertion sort; [Kay 86] notes that a heap sort is faster for large lists.

5.2 Tracing A Ray Through A 3D Grid

The algorithm to trace a ray through a 3D grid is described in Figure 9. It visits each grid cell intersected by the ray in the order of intersection, and intersects the ray with the cell list in each grid cell visited. At the start of the while loop, t_x, t_y, and t_z are the

[3] The transformation structure should therefore store the matrix A^{-1} and the vector B.

Let the ray be $O + Dt$
Let t be bounded by $t^{\min} \le t \le t^{\max}$
Let the bounding box be $B^{\min}$,$B^{\max}$ where
$B^{\min}$ ($B^{\max}$) is a vector containing the minimum (maximum) xyz extents of the box

For $i \leftarrow$ x-index to z-index Do
If $D_i \ge 0$ Then $b^{\min} \leftarrow B_i^{\min}$, $b^{\max} \leftarrow B_i^{\max}$
Else $b^{\min} \leftarrow B_i^{\max}$, $b^{\max} \leftarrow B_i^{\min}$
If $b^{\max} - O_i < 0$ Then Return no hit
$t \leftarrow (b^{\max} - O_i)/D_i$
If $t \le t^{\max}$ Then
If $t < t^{\min}$ Then Return no hit
$t^{\max} \leftarrow t$
Endif
If $b^{\min} - O_i > 0$ Then
$t \leftarrow (b^{\min} - O_i)/D_i$
If $t \ge t^{\min}$ Then
If $t > t^{\max}$ Then Return no hit
$t^{\min} \leftarrow t$
Endif
Endif
Endfor
Return hit (intersection at $t = t^{\min}$ and $t = t^{\max}$)

Figure 8: Ray/Bounding Box Intersection Algorithm

Let the ray be parameterized by $O + Dt$
Let t be bounded by $t^{\min} \le t \le t^{\max}$
Let the grid volume origin be M
Let the cell extent be C i.e. each grid cell has extent C_x in x, C_y in y, and C_z in z

Compute t^0 — the ray's minimum intersection with the whole grid volume
Compute the position P of this intersection
Compute the grid cell g where this intersection occurs
For $\phi \leftarrow$ x-index to z-index Do
Initialize t_ϕ such that
$M_\phi + iC_\phi \le P_\phi < M_\phi + (i+1)C_\phi$ and
$O_\phi + D_\phi t_\phi = M_\phi + iC_\phi$
$\Delta_\phi \leftarrow C_\phi / D_\phi$
$t_\phi \leftarrow t_\phi + \Delta_\phi$
Endfor

While $t^0 \le t^{\max}$ and g is in grid Do
Let ψ be the index such that $t_\psi = \min(t_x, t_y, t_z)$
$t^1 \leftarrow t_\psi$
If g is nonempty Then
Intersect ray with the list at cell g ($t^0 \le t \le t^1$)
If intersects Then Return intersection
Endif
$t^0 \leftarrow t^1$
$t_\psi \leftarrow t_\psi + \Delta_\psi$
Update g depending on ψ and the sign of D_ψ
Endwhile

Return no intersection

Figure 9: Ray/3D Grid Intersection Algorithm

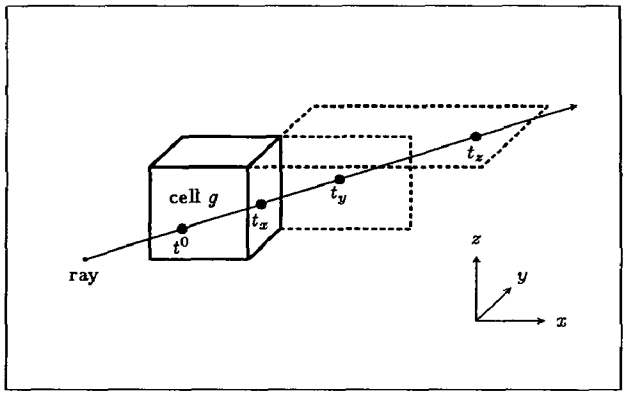

Figure 10: Tracing a Ray Through a Grid — The minimum t_ψ occurs when $\psi = x$, so the next grid cell intersected is adjacent in the increasing x direction. In this case, the algorithm will increment t_x by Δ_x after intersecting the ray with the list in grid cell g.

t values of the ray's maximum (second) intersection with the x, y, and z bounding extents of the current grid cell, g (see Figure 10). The next grid cell may be computed incrementally by finding the minimum of these (t_ψ). This gives the t value of the ray's second intersection with g. It is also the t value of the ray's minimum (first) intersection with the next grid cell intersected. The index ψ indicates which grid cell is intersected next — if D_ψ is positive, the next cell is adjacent in the increasing ψ direction; otherwise, it is adjacent in the decreasing ψ direction.

This grid traversal algorithm is different than the 3DDDA algorithm described in [Fujimoto 86]. Like 3DDDA, no multiply operations are used in the inner loop. Also, the algorithm can be performed using integer arithmetic by scaling the t variables by $1/(t^{max} - t^{min})$. Double precision arithmetic was actually used in the implementation, however, to eliminate inaccuracies in tracing the ray through the grid. Unlike 3DDDA, for each grid cell visited this algorithm computes t^0 and t^1 — the t extents of the ray through the grid cell. This is useful to check that root object intersections actually occur within the cell extent, and in further processing to cull objects in the grid cell list.

Checking Intersections in a 3D Grid Cell

Since a single object may occupy several grid cells, the ray's intersection with the object inside a grid cell should be checked to ensure it is actually within the grid cell. A ray/object intersection should occur in the cell where the ray actually intersects the object, not in the first grid cell visited which contains the primitive. The check for this situation is illustrated in Figure 11.

Culling Inside a 3D Grid Cell

The grid intersection algorithm must intersect the ray with the list in each grid cell the ray intersects. Two optimizations to the algorithm in Section 5.1 can be made for this list intersection.

The first concerns determining whether any object in the cell list is hit by the ray. A simple optimization speeds up detection of situations in which a ray intersects a nonempty grid cell, but misses the cell list bounding box, as in Figure 12. Many times, most of the extents of the cell list bounding box are identical to the cell extent. Since the grid traversal algorithm has already computed the ray's intersection with the cell extent, the algorithm need only process the cell list bounding planes that are different from the cell extent.

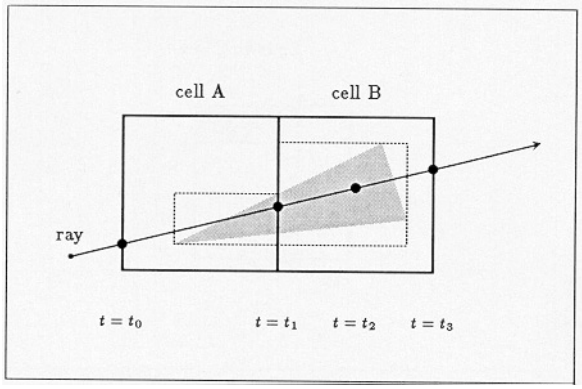

Figure 11: Intersecting a Ray with an Object in Multiple Grid Cells — A ray intersects a triangle in cell B at $t = t_2$. The ray also intersects the bounding box (dashed lines) of the triangle in cell A, but not the triangle itself. When it processes cell A, the algorithm checks that the ray intersection with the triangle is between t_0 and t_1. Since it is not $(t_2 > t_1)$, it correctly returns the intersection of the ray with the triangle only after processing cell B, where $t_1 < t_2 < t_3$.

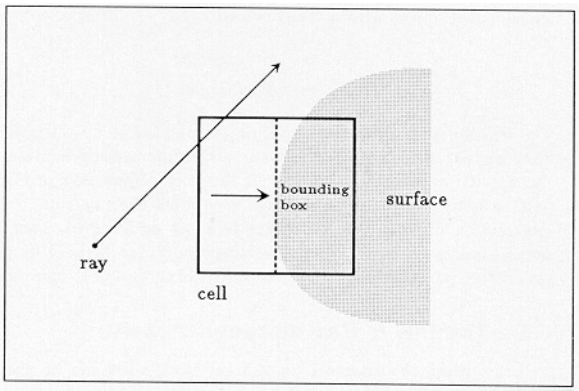

Figure 12: Bounding Box Inside a Grid Cell — The bounding box around the surface inside a grid cell differs from the cell extent in only one extent, identified with an arrow. We need only check one extent to see that the ray misses the bounding box inside this cell.

This is accomplished by storing six flags along with the cell list bounding box extents, which indicate whether or not the extent differs from the cell extent. These flags are trivially computed during preprocessing. The algorithm in Figure 8 is modified to disregard extents whose flag is false.

A second optimization concerns determining which objects in the list are intersected by the ray. A simple cull called the *ray box cull*, shown in Figure 13, determines if a ray misses an object's bounding box using only six comparison operations. The ray box cull is much faster than the bounding box intersection algorithm, but is less strict (note Object B, whose bounding box is not intersected by the ray, but whose bounding box does intersect the ray box). In practice, for very simple primitives like triangles, it has been effective enough to replace the bounding box test, since the bounding box test has complexity on the order of a ray/triangle intersection.

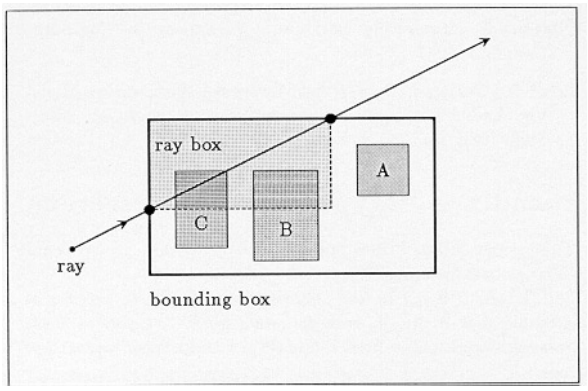

Figure 13: The Ray Box Cull — The t extents of the ray (t^{min} and t^{max}) through the cell list bounding box are used to construct a box, called the *ray box*, with corners at the ray's two intersections with the bounding box. If an object inside the cell extent is intersected by the ray, then its bounding box must overlap the ray box. By computing the ray box once for all the objects in the cell list, any object whose bounding box does not overlap the ray box can be culled, like Object A.

6 Results

Figures 14 and 15 show how time to render a tessellation of a single surface depends on the number of triangles in the tessellation. Times for both graphs are given in seconds to render a non-antialiased (one ray per pixel) 128 by 128 pixel resolution picture. Reported times are for an IBM 4381/Group 12 running Amdahl UTS. An example picture produced is shown in upper left corner of the graph. The dashed line is the time to render the non-tessellated surface using an analytic algorithm in the case of the sphere [4], and an iterative algorithm in the case of the superquadric (see [Barr 84] for an explanation of superquadrics). The solid lines represent graphs of time vs. number of triangles for grids of various cell sizes.

The graphs demonstrate that the time to render a tessellated surface grows quite slowly with increasing number of triangles. Further, in the case of superquadrics, the iterative approach is slower than tessellating and rendering. Only about 2000 triangles were required to produce an image of the superquadric which was indistinguishable from that produced by the iterative algorithm, while the rendering time for this tessellation was half that for the iterative algorithm. Tessellations containing up to 50,000 triangles were still faster than the iterative algorithm.

Yet, superquadrics are very simple parametric surfaces. Tessellation is even more advantageous for complex surfaces whose evaluation can cost hundreds of times more than a superquadric. In experiments, rendering time for tesselations depended on the number of triangles, surface area, and projected screen area of the tessellation. It was relatively independent of the mathematical definition and shape of the parametric surface. Numerical techniques, in contrast, depend on the complexity of the parameterization of the surface.

For example, the grass blade rendered in several included pictures is a parametric surface defined by an integral of a specified Jacobian function that governs how the surface normals behave.

[4]The sphere graph is included for comparison purposes only; tessellation is not necessary for quadric surfaces for which ray intersections may be computed analytically.

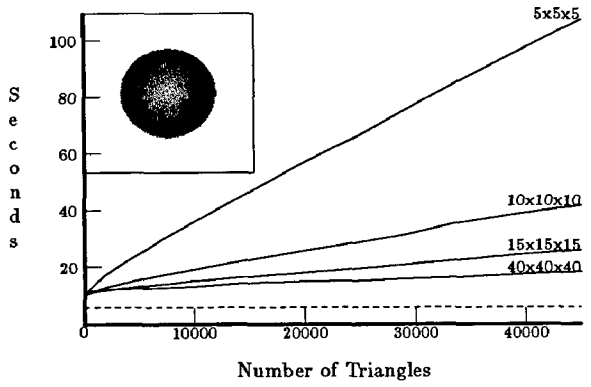

Figure 14: Time to Render Sphere Tessellation

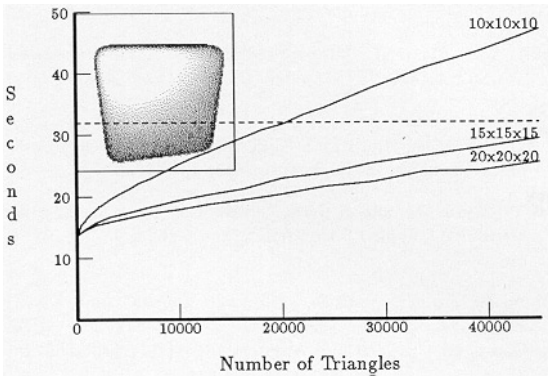

Figure 15: Time to Render Superquadric Tessellation

Evaluation of surface points requires numerical integration and is very expensive. After tessellating once, this surface was incorporated into many models at a rendering cost roughly equal to a tessellated sphere or superquadric of equal size, surface area, and number of triangles. Use of a numerical technique to ray trace the blades would be prohibitively slow, if it can be made to work at all.

The algorithm has been effective for fast rendering of models. For simple pictures (< 100,000 primitives), it consistently performed about twice as fast as the current implementation of the algorithm developed in [Kay 86], which claimed to out-perform competing algorithms such as octrees. It was also able to render complex models that have never been attempted using conventional ray tracers. Table 1 shows the rendering time in CPU hours and number of primitives for pictures included in this paper. All pictures were computed at 512 by 512 pixel resolution.

The times for ray tracing these images are comparable to times for conventional ray tracers to generate images containing a few hundred polygons and spheres. In this same rendering time, our ray tracer has generated pictures containing huge numbers of primitives, and surfaces that would require much greater rendering time using other published techniques.

7 Acknowledgements

Tim Kay and Jim Kajiya provided advice on ray tracing; Brian Von Herzen on tessellation of surfaces. Figures and monochrome

Title	Primitives	Rays/Pixel	Hours
graphics lab	100	16	12
teapot museum piece	10,000	16	8
reflective bristles	15,000	16	12
statue of liberty	100,000	16	14
brass ornament	100,000	16	9
flowers, grass, clovers	200,000	16	3.5
glass museum piece	400,000	16	8.5
grass and trees	$2x10^9$	16	16
field of grass	$4x10^{11}$	16	12

Table 1: Rendering Time For Pictures

raster images in this paper were incorporated using software written by Wen King Su and Brian Von Herzen.

8 References

[Barr 81] Barr, Alan H., "Superquadrics and Angle Preserving Transformations," *Computer Graphics and Applications*, 1(1).

[Barr 86] Barr, Alan H., "Ray Tracing Deformed Surfaces," Computer Graphics, 20(4), August 1986, pp. 287-296.

[Cyrus 78] Cyrus, M. and J. Beck, "Generalized two and three dimensional Clipping," Computers and Graphics, 3(1), 1978, pp. 23-28.

[Fujimoto 86] Fujimoto, Akira, Takayuki Tanaka, and Kansei Iwata, "ARTS: Accelerated Ray-Tracing System", IEEE Computer Graphics and Applications, 6(4), April 1986, 16-26.

[Glassner 84] Glassner, Andrew S., "Space Subdivision for Fast Ray Tracing," IEEE Computer Graphics and Applications, 4(10), October, 1984, pp. 15-22.

[Kalra 86] Kalra, Devendra, M.S. dissertation in preparation.

[Kaplan 85] Kaplan, Michael R., "The Uses of Spatial Coherence in Ray Tracing," ACM SIGGRAPH '85 Course Notes 11, July 22-26 1985.

[Kajiya 82] Kajiya, James T., "Ray Tracing Parametric Patches," Computer Graphics, 16(3), July 1983, pp. 245-254.

[Kay 86] Kay, Timothy L., James T. Kajiya, "Ray Tracing Complex Scenes," Computer Graphics, 20(4), August 1986, pp. 269-278.

[Joy 86] Joy, Kenneth I., Murthy N. Bhetanabhotla, "Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques and Ray Coherence," Computer Graphics, 20(4), August 1986, pp. 279-286.

[Rubin 80] Rubin, Steve M. and T. Whitted., "A Three-Dimensional Representation for Fast Rendering of Complex Scenes," Computer Graphics 14(3), July 1980, pp. 110-116.

[Toth 85] Toth, Daniel L., "On Ray Tracing Parametric Surfaces," Computer Graphics 19(3), July 1985, pp. 171-179.

[Von Herzen 85] Von Herzen, Brian P.,"Sampling Deformed, Intersecting Surfaces with Quadtrees," Caltech CS Technical Report 5179:TR:85, pp. 1-40.

[Von Herzen 87] Von Herzen, Brian P.,"Accurate Sampling of Deformed, Intersecting Surfaces," to appear in Computer Graphics, 1987.

[Whitted 80] Whitted, Turner, "An Improved Illumination Model for Shaded Display," Communications of the ACM, 23(6), June 1980, pp. 343-349.

Appendix — Ray/Triangle Intersection

This appendix describes an efficient algorithm to compute ray/triangle intersections.

Let P_i for $i \in 0, 1, 2$ be the coordinates of the three vertices of the triangle. Let R_i be the corresponding normal vectors at these vertices which are to be used for normal interpolation across the triangle.

During the preprocessing stage, the above information is used to construct a triangle structure, the tessellation unit root object. We first compute and store the normal vector to the plane containing the triangle, N, by

$$N = (P_1 - P_0) \times (P_2 - P_0).$$

We also compute and store a scalar d such that any point, P, in the triangle's plane satisfies $P \cdot N + d = 0$. This scalar is computed by

$$d = -P_0 \cdot N.$$

Lastly, we compute and store an index i_0 such that

$$i_0 = \begin{cases} 0 & \text{if } |N_x| \text{ is maximum} \\ 1 & \text{if } |N_y| \text{ is maximum} \\ 2 & \text{if } |N_z| \text{ is maximum} \end{cases}$$

The triangle structure also stores the three vertices and normals, P_i and R_i. To conserve memory, the triangle structure should store pointers to these since, on average, each vertex in a tessellation is shared by six triangles.

To intersect a ray parametrized by $O + Dt$ with a triangle, first compute the t parameter of the ray's intersection with the triangle plane:

$$t = \frac{d - N \cdot O}{N \cdot D}. \quad (1)$$

Let i_1 and i_2 $(i_1, i_2 \in \{0, 1, 2\})$ be two unequal indices different from i_0. Using the t value obtained from Equation 1, compute the i_1 and i_2 components of the point of intersection, Q, by

$$Q_{i_1} = O_{i_1} + D_{i_1} t$$
$$Q_{i_2} = O_{i_2} + D_{i_2} t.$$

A point enclosure test can then be performed by computing scalars β_0, β_1, and β_2 according to [5]

$$\beta_i = \frac{[(P_{i+2} - P_{i+1}) \times (Q - P_{i+1})]_{i_0}}{[N]_{i_0}} \quad (2)$$

where addition in subscripts is modulo 3. Note that these β's are the barycentric coordinates of the point where the ray intersects the triangle plane. Only the i_0 component of the cross product is computed; the value of Q_{i_0} is therefore unnecessary. Q is inside the triangle if and only if $0 \le \beta_i \le 1$ for $i \in \{0, 1, 2\}$. Division by N_{i_0} can be eliminated by appropriate rearrangement of the test implied by Equation 2. The interpolated normal $\tilde{N}$ is given by

$$\tilde{N} = \beta_0 R_0 + \beta_1 R_1 + \beta_2 R_2.$$

[5] $[X]_i$ denotes the ith component of the vector X.

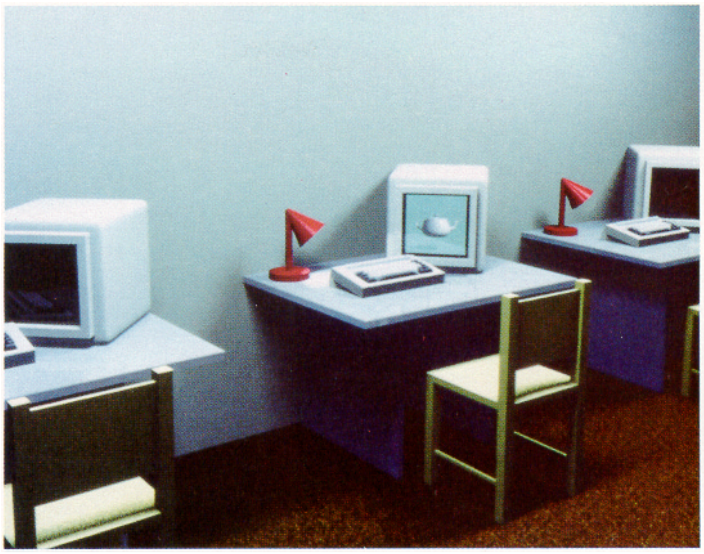

Figure 15: Graphics Lab — The carpet texture map in this image was created by ray tracing a simulated carpet containing roughly 125,000 triangles. Note the diffuse shadows from three extended light sources.

Figure 17: Statue of Liberty — The statue database was created using I-DEAS Geomod from SDRC, and contained about 12,000 triangles after processing. Each tree contains roughly 10,000 primitives.

Figure 16: Field of Grass — This image was rendered from a model description containing more than 400 billion primitives.

Figure 18: Flowers, Grass, and Clovers

Figure 19: Brass Ornament

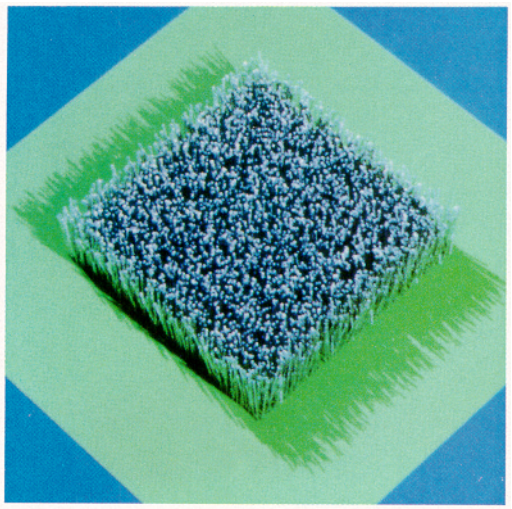

Figure 20: Reflective Bristles

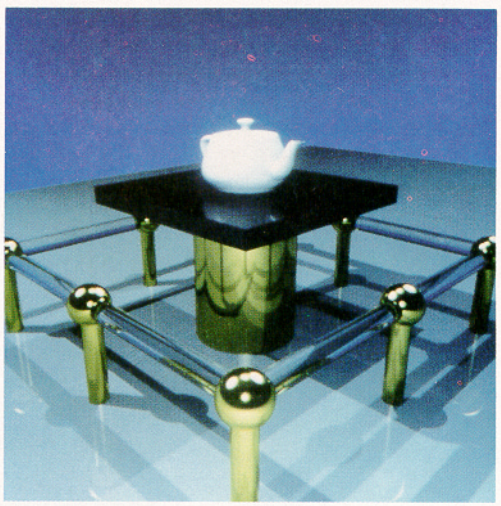

Figure 21: Teapot Museum Piece

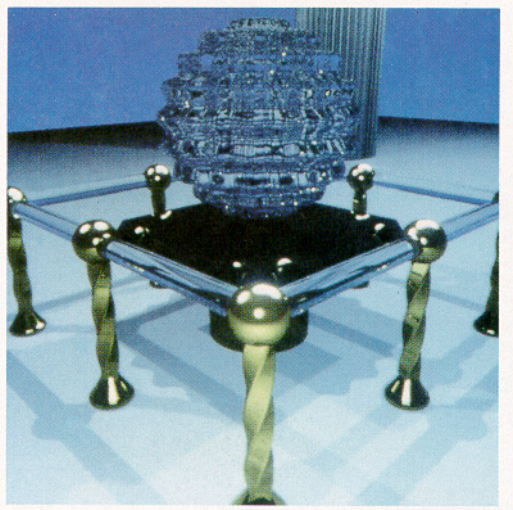

Figure 22: Glass Museum Piece

Figure 23: Trees and Grass