UgRay

An Efficient Ray-Tracing Renderer for UniGrafix

Donald M. Marsh

Master's Project Report
Under Direction of
Professor Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
May 1987

ABSTRACT

Ray-tracing has proven to be an elegant and versatile algorithm in the pursuit of increased realism in computer-generated images. In order to reduce the computational expense of general ray-tracing, we utilize a uniform spatial subdivision. We report on a number of practical issues in designing a ray-tracing renderer, including a fast algorithm for inserting polygons into the spatial subdivision, results of optimizations to the ray/polygon intersection routine, and inexpensive shading and anti-aliasing routines.

Acknowledgements

This work was supported in part by Tektronix, Inc. and by the Semiconductor Research Corporation.

I would also like to thank my research advisor, Dr. Carlo Séquin, for his many useful suggestions and insights during the development of *UgRay*. He was especially helpful in keeping progress on track when it was tempting to get lost in the details of particular portions of the renderer.

Chris Goodman wrote parts of the user interface and performed much-needed testing during the early stages of development.

Rick Speer provided me with many useful references and suggestions from the computer graphics literature. Greg Couch, Henry Moreton, Ziv Gigus, and Mark Segal also offered useful advice and criticism at different times during the project.

Table of Contents

1. Introduction

The UNIGRAFIX modeling system has been evolving at Berkeley during the past five years. Berkeley UNIGRAFIX consists of a modeling language capable of describing wires and polygons, *generator programs* which generate UNIGRAFIX descriptions of various complex objects, *modifier programs* which accept UNIGRAFIX scene descriptions and modify them in some meaningful way, and *renderers* which read a UNIGRAFIX scene description and some viewing parameters in order to produce the corresponding graphic image.

The UNIGRAFIX renderers to date have been based on scan-line algorithms. Scan-line algorithms divide an image into a number of rows (usually determined by the number of scan-lines to be displayed on a raster device), and then render one complete row (or scan-line) at a time until the whole image is displayed. Because typical images change little from scan-line to scan-line, much of the geometric information can be retained and updated incrementally, thus reducing the amount of computation necessary [SSS74]. Further optimizations can usually be employed within a single scan-line, increasing rendering speed even more.

Scan-line algorithms are ideally suited for some of the display devices we have used in the past. A full-sized plot on a Versatec printer, for example, requires the computation of nearly 50 million pixels. This amount of computation can quickly overwhelm many algorithms that don't exploit image coherency.

Despite these advantages of scan-line algorithms, it became obvious that a new type of renderer was needed to supplement the capabilities of the existing UNIGRAFIX renderers. Here are some of the issues that led to this realization:

Large scenes on small devices. When a complicated scene containing many polygons (10,000 to 100,000 polygons) is displayed on a device such as a video terminal (displaying 250,000 to 1 million pixels), each polygon covers few pixels. Most of the scan-line algorithm optimizations are based on the assumption that a polygon covers many pixels. Scan-line algorithms spend a significant amount of time initializing variables containing incremental values that are rarely used when a polygon shrinks to the size of a few pixels or less.

Increasing user sophistication and expectations. As the science of computer graphics advances, users are becoming more demanding regarding the quality and realism of computer-generated images. At the least, a renderer should be able to produce shadows, reflective surfaces, and transparent objects that refract light realistically. The difficulty of adding such effects to scan-line algorithms will be explained below.

Increased computer resources. The availability of more computational power (faster machines, idle workstations) has made it possible to investigate more expensive rendering techniques. Although we must still be concerned about the speed of different algorithms, the computational power and memory required to render complex scenes (10,000 polygons or more) is now available in some of the more advanced personal computers!

Our first response to these issues was to examine methods for modifying and upgrading our existing scan-line algorithms. Unfortunately, the optimizations that make scan-line algorithms attractive preclude certain types of realistic effects. For example, it is essential to perform a

perspective transformation before the scan-lines are rendered. The transformed scene may then be orthogonally projected in one direction. Unfortunately, light that is reflected or transmitted through refractive material can travel in any direction, and no convenient mechanisms are available to help us in this computation. Furthermore, these computations become more difficult to perform in the transformed coordinate space.

Approximate techniques to solve these difficulties have been developed by a number of researchers. Shadows can be computed by performing a separate hidden-surface computation from the view of each light source [Ber86, HaG86]. Environment maps can be texture-mapped onto shiny surfaces [BlN76, Cat78]. Transparency can be simulated without refraction by mixing the colors of foreground and background polygons. Unfortunately, each of these methods requires implementation of unrelated mechanisms, and each has limitations that degrade realism in ways that cannot be remedied easily.

Ray-tracing, on the other hand, presents a simple and extensible approach to these difficulties. The basic algorithm is nearly as easy to program as it is to describe: simply determine the equation of a semi-infinite line segment (a *ray*) passing through the eyepoint and a point on an image plane corresponding to a pixel on the output device. Then calculate the point of intersection between this ray and every object in the scene. If the ray intersects more than one object, the point of closest intersection is the point that is displayed at the appropriate pixel. If no intersections are found, the background color is displayed.

Once a surface point has been identified in the above manner, ray-tracing allows us to calculate realistic shading effects using the same mechanisms. Shadows are calculated, for example, simply by generating new rays between the visible surface point and each light source. If one of these shadow rays intersects an opaque object, the surface point is in shadow with respect to the corresponding light source (figure 1.1).

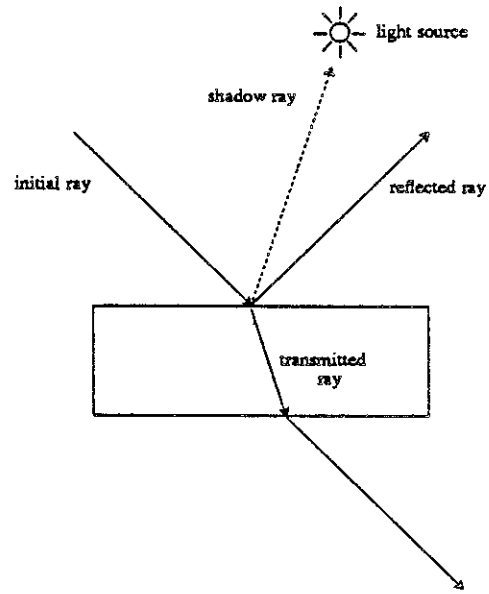


Figure 1.1 – Realistic shading using additional rays

If the surface is reflective, we can generate a new ray in the reflection direction. If this secondary ray intersects another object in the scene, the reflection of that object will be visible in the initial surface point. By applying this process recursively, multiple reflections can be simulated that would be virtually impossible to calculate using other methods [Whi80].

In a similar manner, a ray that intersects a transparent object can generate two secondary rays: a refracted component that is bent in the proper direction depending on the index of refraction, and a reflected component that behaves as described above. Some very stunning images have been generated using this technique to model the behavior of light in an environment containing both reflective and refractive objects.

Notice the absence of several traditional computer graphics problems. No perspective transformation is needed, for example. Correct perspective is automatically generated when we calculate the equations of rays fanning out from the eyepoint. No clipping is required either. Objects outside our field of view present no special problems. Backface elimination, a technique for eliminating faces on the back sides of objects, is not performed because it could lead to erroneous results (the back side of an object can cast a shadow on, or be reflected in a visible surface).

The beauty of ray-traced images and the conceptual and practical simplicity of the ray-tracing algorithm has led to its growing popularity, despite one critical problem: speed.

2. Techniques for Accelerating Ray-tracing

The speed disadvantages inherent in the ray-tracing algorithm become apparent when we examine the inner loop of the algorithm, where the program can spend up to 95% of its execution time [Whi80]:

```
for each scan-line do
    for each pixel in this scan-line do
        form ray through eyepoint and pixel;
        INTERSECTION := infinity;
        OBJECT := null;
        for each object do
            NEWINTERSECT := intersect ray with this object;
            if NEWINTERSECT < INTERSECTION then
                INTERSECTION := NEWINTERSECT;
                OBJECT = this object;

        if OBJECT = null then
            color pixel with background color;
        else
            color pixel with OBJECT color;
```

The critical portion of the above algorithm occurs when a ray is intersected against each object in the scene. For a scene of moderate complexity (10,000 polygons) and resolution (512 by 512 pixels), 2.6 billion intersection tests are required. If each intersection test requires 20 floating-point operations, this would take 52 billion floating-point operations (over 5 days' worth of computation on a Vax 750 with floating-point accelerator!)

The situation is even more dismal, however. The above estimates do not account for additional rays needed to perform anti-aliasing and display of shadows, reflections, and light transmitted through transparent volumes. These requirements typically increase the amount of computation by at least an order of magnitude.

Many different strategies have been implemented to reduce this astronomical number of intersection calculations. Here is a brief review of some of the software solutions (hardware solutions have also been proposed [Ull83]):

The Hybrid Approach. It is initially tempting to use a faster scan-line or Z-buffer algorithm to calculate visible surfaces, then use ray-tracing to add reflections, shadows, and other desirable ray-tracing effects. Unfortunately, unless some other scheme is incorporated to accelerate the ray-tracing part of such a program, it will spend most of its time there. Furthermore, the types of data structures needed for one part of the hybrid program are incompatible with the other part. For example, a scan-line algorithm requires many pointers (edges pointing to faces, faces pointing to edges, vertices pointing to edges, etc.) to operate efficiently. A ray-tracing algorithm requires minimal support in this area, but requires other structures such as

bounding boxes. A hybrid program could therefore consume a substantial amount of memory trying to accomodate the needs of the differing algorithms.

Coherent Ray-Tracing. One reason scan-line algorithms are fast is that, unlike ray-tracing, it isn't necessary to examine each pixel individually. Often a span of pixels is known to be covered by a single polygon, and the whole span can be colored without additional computation. Furthermore, a span of similar length is usually covered on the following scan-line by the same polygon. The performance of ray-tracing might be improved if rays could "talk" to each other. After one ray is tested against the objects in the scene, it could tell neighboring rays which objects it came close to intersecting, and the other rays could restrict their intersection tests to such objects.

One of the first efforts to concentrate on coherence between one ray and the next appears in [SDB85]. Cylinders defining *safety regions* through which multiple rays can pass unobstructed were created dynamically in order to reduce the number of required intersection calculations. Unfortunately, the overhead of creating and testing such cylinders made the algorithm slower than conventional ray-tracing in all but the most trivial cases. Enhancements to the coherent ray-tracing algorithm have been proposed more recently [Han86], but the technique has not yet proved to be competitive with other methods available at this time.

Beam-Tracing. Beam-tracing is another attempt to use coherency to improve rendering performance [HeH84]. A cluster of rays contained in a polygonal cone (a *beam*) is traced through the scene. In simple scenes, many rays follow similar paths, and this strategy cuts computation cost significantly (by a factor of 20 to 100, according to the authors). Unfortunately, as scene complexity increases, beams are fragmented to such an extent that all speed advantages are lost. Furthermore, the method must use linear approximations to non-linear refractive distortion, and curved surfaces cannot be handled well.

Bounding Volumes. The use of bounding volumes placed around primitives was proposed as a technique to reduce intersection computations as early as Whitted's first paper on ray-tracing [Whi80], and this continues to be a fruitful area of research [KaK86]. A relatively inexpensive intersection calculation is performed on a bounding volume, and the ray is intersected with the primitive(s) inside only if the ray hits the bounding volume. Spheres and rectangular solids are typically used as bounding volumes because the corresponding intersection tests can be performed quickly.

The major trade-off to be considered when using bounding volumes is *tightness* versus *intersection complexity*. Obviously, a bounding volume that does not conform well to the shape of the enclosed primitives will intersect many rays that miss the primitives. In this case, the bounding volume causes computational overhead that it was designed to alleviate. A sphere bounding a single polygon exhibits this problem. The solution is to use a more complicated bounding volume that more closely approximates the shape of the enclosed primitives (such as an arbitrary polyhedron). But such volumes usually require more expensive intersection tests. If the bounding volume intersection test gets too expensive, it is better to calculate the intersections with the primitives instead!

The main advantage of the bounding volume approach is that the volumes can be organized hierarchically. The number of primitives that must be intersected can be progressively reduced during descent of the bounding volume hierarchy. Unfortunately, the optimal organization of this hierarchy doesn't always conform to the hierarchical description of the scene. For maximum speed, a bounding volume hierarchy must be generated for a scene that is not described hierarchically (*i.e.*, a *flattened* scene), and this is a difficult problem. Furthermore, it is difficult to know how deep to extend the bounding volume hierarchy. A deeply-nested bounding volume hierarchy can contribute significant computational overhead to each intersection test that may not be recovered by reducing the number of intersection calculations.

Spatial Subdivision. Spatial subdivision techniques subdivide a scene into small regions (usually rectangular or cuboidal volumes, although not necessarily [DiS84]). When a ray enters a region, intersection tests are performed against all primitives that are contained in the region. If no intersections are found, the ray is propagated into the next region that intersects the ray.

Spatial subdivision accelerates ray-tracing in two ways. First, regions are usually small enough so that only primitives in the immediate vicinity of the ray are tested. Distant primitives fall into regions that are simply not considered. Second, the occurrence of intersections within a region removes the necessity for further intersection testing. Since regions are considered in depth-order, any closer intersections would have been identified in previously processed regions. By contrast, basic ray-tracing requires testing of all remaining primitives to find closer intersections.

Octree Subdivision. Octree subdivision is a spatial subdivision scheme in which the scene is placed in an adaptive grid of cuboidal cells [FTI86, Gla84]. The size of individual cells is determined by the number of primitives in the corresponding volume of the scene. A cell is recursively subdivided into eight smaller cells whenever the number of primitives passing through it exceeds some threshold. The main advantage of the octree subdivision is this ability to concentrate cells in the regions of highest scene complexity. The main disadvantage is the computational expense of traversing the octree structure when propagating rays from cell to cell. For a scene of relatively uniform complexity, the octree structure imitates a uniform spatial subdivision (below), but it requires more memory and more computation for traversal by the rays.

Uniform Subdivision. Uniform subdivision is a spatial subdivision technique that uses a uniform grid of cuboidal cells with fixed size [FTI86]. The adaptive nature of the octree subdivision is sacrificed to achieve faster cell-to-cell propagation of rays, simple cell addressing, lower memory requirements (in most cases), and fast insertion of primitives (to be discussed later). The main disadvantage of the uniform subdivision occurs in scenes with widely varying complexity (two dense collections of primitives separated by a large distance of empty space, for example). Such scenes cause poor distribution of primitives in the cells, resulting in poor computational performance.

Dynamic Subdivision. The performance of spatial subdivision schemes is dependent on the time spent performing intersection tests within each cell of the subdivision. This time is the product of the number of primitives within a particular cell, the time required to intersect each primitive, and the number of rays that enter the cell. This product can be defined as the *load* experienced by the cell. Equitable distribution of this load between cells will improve the overall

performance of subdivision schemes. However, since statistics on the number of rays entering a cell cannot be easily predicted, statistics must be gathered during execution. An algorithm that tries to optimize the total load must do so by dynamically growing cells that are sparsely used, and shrinking cells that are heavily loaded. This has been performed with some success on an array of processors [NeO86], because a processor that is idle can be put to work analyzing and redistributing the load. This is more difficult to accomplish efficiently on a single processor. It is also difficult to know exactly when and how to split or coalesce cells, and the dynamic redistribution of load must be performed quickly and inexpensively or the potential savings will be overwhelmed by the overhead. Furthermore, it is difficult to preserve simplicity that allows fast cell-to-cell propagation of rays.

This concludes our survey of recent techniques used to accelerate ray-tracing. Each of these techniques attempts to accelerate ray-tracing primarily by reducing the number of ray intersections. Although this is the most important determinant of ray-tracing speed, two other areas should also be considered: 1) the number of rays needed to accurately render the scene (this affects anti-aliasing and secondary ray generation), and 2) the speed of intersection tests. These issues will be discussed in later sections of this report.

After careful consideration of each alternative and possible extensions, we concluded that the speed and simplicity of the uniform spatial subdivision outweigh the performance disadvantage in nonuniformly clustered scenes. The uniform subdivision lies at the heart of the implementation of the new ray-tracing renderer, *UgRay*.

3. Overview of Program Operation

In this section, we present an overview of the operations *UgRay* performs to preprocess and render a scene. Many of the operations mentioned here will be discussed in greater detail later, but it is important to understand the operation of the program as a whole before examining the details of its parts. This is especially important because many of the design decisions made in one part of *UgRay* have affected the design of other parts.

3.1. Initialization and Parsing of Rendering Options

To control the quality and computational expense of a rendered image, *UgRay* accepts numerous user-specified options (see the manual pages in Appendix B). *UgRay* first assigns default values to all options. The default values can be overridden by the user in a scene-independent configuration file, a scene-dependent command file, or on the command line.

3.2. Scene File Processing

UgRay reads each scene file specified as input. Scene files are text files containing polygon descriptions in the UNIGRAFIX language, with some restrictions and extensions. The major restriction is that polygons must be convex with a single contour (no holes). Extensions allow specification of surface properties (reflectivity and transparency, for example), volume properties (light attenuation and index of refraction), and colored light sources. See the manual pages in Appendix B for further infromation. As *UgRay* reads a scene file, it builds temporary data structures describing each vertex and polygon. The polygon data structure contains a list of pointers to the vertices that define a given polygon. Numerous polygons may share the same vertex.

3.3. Flattening

The scene hierarchy is now flattened. Polygons within UNIGRAFIX definitions are instantiated as directed by the *instance* and *array* statements in the UNIGRAFIX scene file. Although it is not absolutely necessary to transform the scene to eye coordinates for ray-tracing, *UgRay* performs this transformation to simplify computations involving the spatial subdivision, the image plane, and most importantly, ray intersection calculations. Since the transformation to eye coordinates can be incorporated in the flattening matrix, the transformation does not require any additional computation. When complete, the scene is defined in a coordinate system with the eye at the origin (for a perspective view), the z axis pointing in the direction of view, and the image plane at $z = 1$ (figure 3.1). As each vertex is instantiated in the eye coordinate system, the minimum and maximum coordinates of the scene are updated so that the bounds of the space to be subdivided is known.

3.4. Allocation of Spatial Subdivision

At this point, *UgRay* knows the number of polygons in the scene, the bounds of the scene, and how many cells to allocate in the subdivision. From this information, it can compute the size of an individual cell in the spatial subdivision and how many cells to allocate along each axis.

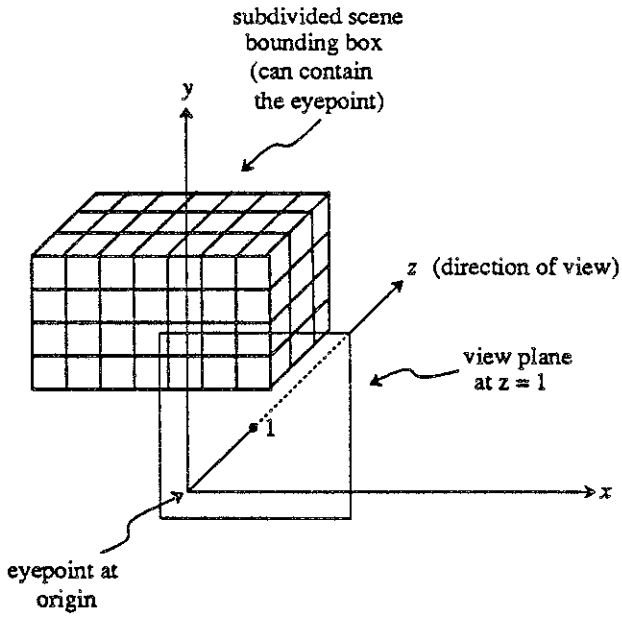


Figure 3.1 – Eye coordinate system

3.5. Vertex Preprocessing

The entire scene is now scaled so that the side of each cuboidal cell in the spatial subdivision becomes one unit in length. Although this requires a moderate amount of computation (one division for each of three coordinates per vertex), it simplifies later calculations when *UgRay* must find the cell containing a particular vertex. A surface normal is calculated at each vertex by averaging the surface normal vectors of the faces that include the vertex, in case Gouraud shading or normal interpolation is desired to simulate smooth surfaces [Gou71, Pho75].

The field of view is also calculated during vertex preprocessing. For each vertex, we calculate the horizontal and vertical angles between the the direction of view and a ray containing the vertex and the eyepoint. The maximum horizontal and vertical angles are used to determine an image scaling factor so that the entire scene is displayed on the display device. The user can specify a *view angle* which overrides this automatic scaling calculation.

3.6. Polygon Preprocessing

Preprocessing of polygons occurs in the same procedure that vertex preprocessing is performed, because information is shared between the two operations. Polygon preprocessing accelerates ray intersections and simplifies insertion of polygons into the spatial subdivision. The main objective is to discover which of the *xy*, *xz*, or *yz* planes in the eye coordinate system a particular polygon can be orthogonally projected onto such that the projection contains the maximum area (intuitively, the projection that is least distorted). We will refer to this plane as the *major projection plane* for a particular polygon. The major projection plane can be determined for a given polygon simply by comparing the magnitude of the components of the vector normal to the polygon. The component with the largest magnitude is the direction of projection. For

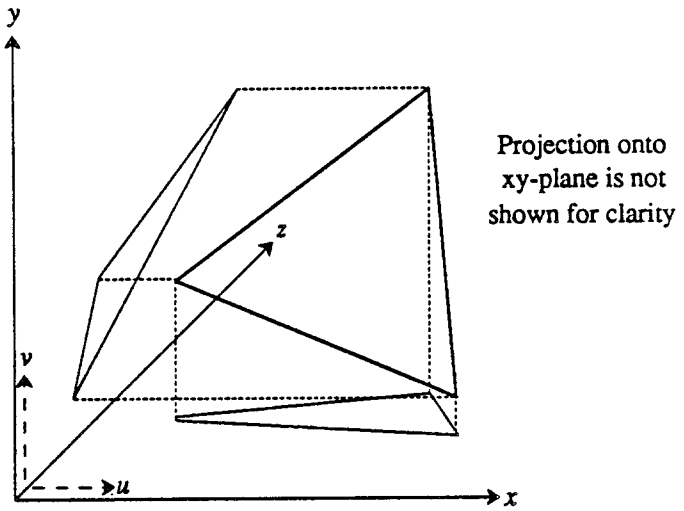


Figure 3.2 – Determining a polygon's major projection plane

example, if a polygon's normal vector has a z component that is larger than the x and y components, the polygon will be projected along the z axis onto the xy major projection plane (figure 3.2).

The two basis vectors contained in the major projection plane are renamed u and v for uniformity between the three planes (for example, in the xy projection plane, the x axis is the u axis and the y axis is the v axis). Depending on the direction in which a particular polygon is projected onto its major projection plane, the vertices of the projected polygon may be listed in a clockwise or counterclockwise order. If necessary, the list of vertices is reversed so that the vertices trace the border of the projected polygon in a clockwise direction when the v axis points up and the u axis points to the right (figure 3.3).

The vertices and edges of each polygon can be classified as follows using the major projection of the polygon. The vertex within the polygon that has the maximum projected v coordinate is the *top* vertex (if more than one vertex has the same v coordinate, we define the rightmost (maximum u coordinate) as the *top* vertex). Similarly, the vertex that has the leftmost minimum projected v coordinate is the *bottom* vertex. The edges leading from the *top* vertex to the *bottom* vertex are *right* edges, and the edges leading from the *bottom* to the *top* vertices are *left* edges (figure 3.3).

3.7. Insertion of Polygons in the Spatial Subdivision

The classification of vertices and edges helps in the insertion of polygons in the spatial subdivision. This process will be discussed in section 4.

3.8. The Anti-Aliasing Routine

At this point, we are ready to start casting rays into the scene. Which rays are actually generated depends partly on the anti-aliasing strategy chosen. *UgRay* contains several different anti-aliasing routines that invest increasing amounts of computation to produce images of better

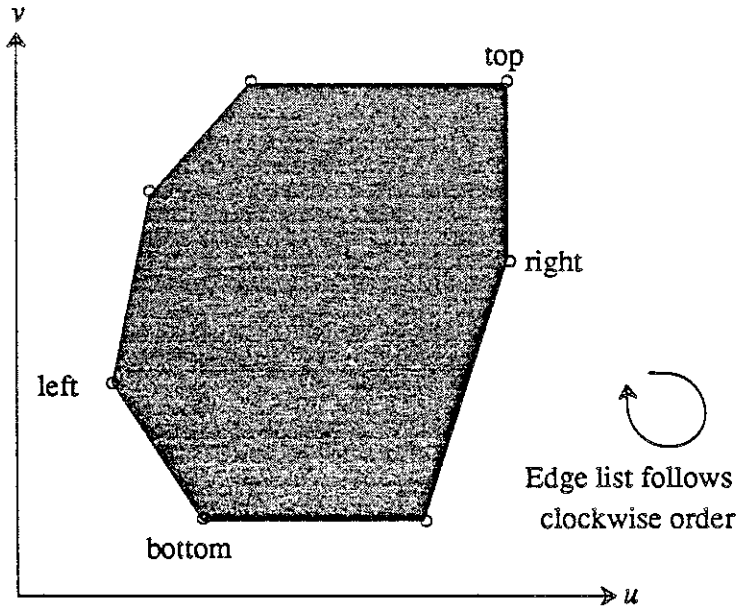


Figure 3.3 – Classification of vertices

quality. Each anti-aliasing routine identifies a point on the image plane that it would like to sample. If this point has not been sampled previously, the appropriate ray-casting routine is called to return the color at that point. Further processing (adaptive sampling and weighted averaging of samples) may be performed by the anti-aliasing routine, resulting in a pixel value that is written to the image file.

3.9. The Ray-Casting Routine

The ray-casting routine is responsible for constructing the equation of a ray passing through the eyepoint and a given point on the image plane. *UgRay* contains two ray-casting routines: one generates rays for a scene viewed in perspective; the other generates rays for a scene projected orthogonally. Although there is no fundamental difference between these routines, certain optimizations can be applied to each case if they are treated individually.

3.10. Cell Identification

If the eyepoint lies outside the spatial subdivision, it is necessary to calculate the first point at which a particular ray enters the subdivision. The ray is then propagated through the subdivision one cell at a time. This requires ordered identification of each cell that the ray passes through. Cells are identified using simple integer arithmetic similar to that used to draw lines on a raster device.

3.11. Ray Intersection

Once a cell has been identified by the cell identification procedure, every polygon within the cell must be intersected with the ray that enters the cell. In reality, some polygons span several cells and need only be intersected once with a given ray. Therefore, the ray intersection

routine is called only for those polygons that have not been intersected with the ray in previous cells. The routine that performs the intersection calculation is called millions of times in a complicated scene, so it is one of the most highly optimized routines in the renderer. Four separate routines are used to intersect four types of rays: primary rays generated for a perspective view, primary rays generated for an orthogonal view, and secondary rays (reflected and transmitted rays), and shadow rays. Like the ray-casting routine, there are no fundamental differences between these intersection routines, but various optimizations that are applicable in individual cases make separate treatment worthwhile. Each routine uses the classification of vertices and edges produced by polygon preprocessing to accelerate the intersection test.

3.12. Shading

Once a visible surface has been identified, it is necessary to calculate its color. In the simplest case, a color can be calculated just once for the given polygon, and this color can be returned whenever future rays strike the polygon. Unfortunately, this is rarely the case in realistic scenes. Shadow rays must be generated in the direction of each light source to determine if any object casts a shadow on the surface point. A reflected ray determines which polygon (if any) reflects in a shiny surface. A transmitted ray is cast to determine if any polygon is visible through a transparent surface. Rays passing through volumes or atmospheric fog must be properly attenuated. The shading procedure is executed recursively to simulate multiple reflections and can be very expensive.

4. Initializing the Spatial Subdivision

As noted earlier, the most important factor determining the speed of a ray-tracing renderer is the degree to which it can restrict the number of ray intersection tests needed to render a scene. Effective techniques for accomplishing this task must balance the cost of performing superfluous intersection tests with the cost of eliminating them. In other words, it might be possible to develop techniques for eliminating almost all unnecessary intersection calculations, but the computational cost of doing so would be greater than the cost of the eliminated intersection tests would have been.

The uniform spatial subdivision used by *UgRay* requires minimal computational overhead to restrict the number of intersection calculations. Allocation and initialization of the spatial subdivision is therefore an important step in the preprocessing of the scene.

4.1. Allocating Cells in the Spatial Subdivision

The computational overhead required to propagate a ray from one cell to another is an important factor in determining the speed of a ray-tracing algorithm based on the spatial subdivision technique. We use cuboidal cells to simplify this propagation function (discussed in detail in the next section). Rectangular cells could be used instead, but cell-to-cell propagation of rays would be slightly more complicated.

Although each of the cells in our subdivision is cuboidal, we allow the overall shape of the subdivision to be rectangular. If the overall shape were required to be cuboidal, many cells would be wasted in scenes with bounding boxes that deviate somewhat from a cube. The poor distribution of primitives in this case would expose a major weakness in the uniform subdivision technique.

The number of cells in the subdivision can be specified by the user so that performance can be tuned for a particular scene. If the user does not specify the number of cells, *UgRay* employs a heuristic based on empirical observations and allocates 50 times the total number of polygons in the scene.

Once *UgRay* knows how many cells to allocate and the extents of the scene along each axis (which are calculated during the flattening of the scene hierarchy), it must determine how many cells to distribute along each axis. First, the total volume of the scene bounding box is calculated. The total volume is simply the product of the lengths of the scene extents along each axis. Next, *UgRay* calculates the volume contained in each cell by dividing the total volume by the number of cells to be allocated. The length of the side of each cell is found by taking the cube root of the cell volume. Finally, the length of the scene extent along each axis is divided by the length of the side of a cell to find how many cells to allocate along each axis:

$$cellsize = \left\lfloor \frac{extent_x \; extent_y \; extent_z}{totalcells} \right\rfloor^{\frac{1}{3}}$$

$$cells_x = \max\left(1, \left\lceil \frac{extent_x}{cellsize} \right\rceil\right)$$

$$cells_y = \max(1, \left\lceil \frac{extent_y}{cellsize} \right\rceil)$$

$$cells_z = \max(1, \left\lceil \frac{extent_z}{cellsize} \right\rceil)$$

Special cases are required if one of the extents is zero (this can happen when viewing a single polygon from an orthogonal direction, for example). In this case, the equation for calculating the size of a cell becomes a square root of the product of the non-zero extents.

After the appropriate number of cells has been allocated along each axis, the entire scene is scaled so that the length of the side of a cell is one unit. This simplifies various calculations throughout the renderer. The cell containing a particular vertex can be found without performing any multiplications or divisions, for example. This scaling is performed by dividing the coordinates of each vertex in the scene by *cellsize*.

4.2. Inserting Polygons in the Spatial Subdivision

Several different strategies can be employed to insert primitives (polygons, in the current implementation) in the spatial subdivision. It is not necessary to orient the subdivision in any particular manner with respect to the eye coordinate system, as we have done. Therefore, insertion of primitives could be done only once as a preprocessing step independent of rendering. Presumably, the results of this preprocessing step would be saved in a disk file that would serve as input to the renderer. This method is particularly attractive for a fly-by animation of a static scene (no moving parts within the scene), since the overhead of insertion could be subtracted from the rendering time required by each frame in the animation.

The disadvantage of such a scheme is that the preprocessed file can require several megabytes for a scene of moderate complexity. Such a large file is not only unattractive in terms of disk usage, but also requires a significant amount of slow disk I/O to read at rendering time. If the insertion procedure can be made very fast, we will not lose much by recomputing this information instead of reading it from disk. Furthermore, the preprocessed file presents a version-control problem. The user must remember to reprocess his scene whenever any changes are made, or an outdated image will be produced.

Fortunately, we have developed a fast algorithm for inserting convex polygons in the uniform subdivision. In a complicated scene with many reflective and transparent surfaces, this insertion procedure accounts for about 0.1% of the total execution time.

The insertion procedure adds a polygon P to a list of polygons in each cell that P passes through (figure 4.1). It is easy to design fast insertion procedures that insert a polygon in too many cells (we could simply insert P in all cells contained in the rectangular solid bounding P, for example). No image degradation will result in this case, but rendering speed will be reduced as rays entering a cell are intersected with polygons that lie entirely outside the cell. Therefore, it is worth spending extra computation during the insertion phase to avoid much greater computational expense incurred during the rendering phase.

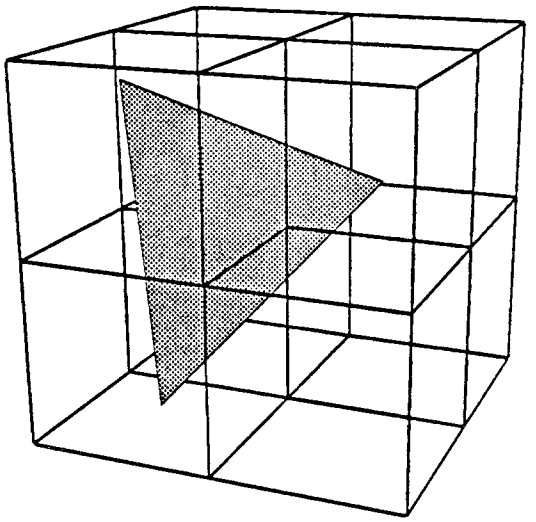


Figure 4.1 – Inserting a polygon in the spatial subdivision

Before insertion, each polygon is preprocessed to facilitate fast intersection testing. This procedure will be discussed in greater detail in a later section. We use one result of this preprocessing here: each convex polygon has an orientation that allows us to identify the top and bottom vertices of the polygon, and the left and right edges.

The insertion algorithm is as follows: First, trace the right edges of the polygon, inserting the polygon in every cell penetrated by a right edge. Cells pierced by edges can be identified quickly using an algorithm similar to that used to trace rays through the subdivision during rendering. This will be discussed in greater detail in the following section.

Next, the left edges are traced in a manner similar to the right edges, and the polygon is inserted in each cell pierced by a left edge.

If the algorithm stopped here, a polygon spanning many cells would now be inserted in each cell pierced by one of its edges. However, the interior portion of the polygon can pass through cells that are not pierced by any of the edges. To identify such cells, *UgRay* constructs a series of line segments that start at a left edge, run across the interior of the polygon, and end on a right edge. The polygon is inserted in each cell pierced by such a line segment (figure 4.2). The only tricky part is to make sure that the interior line segments sample the interior of the polygon frequently enough to identify all interior cells. *UgRay* uses information about the major projection plane associated with the polygon to guarantee correct sampling.

The major projection plane of a polygon identifies the two axes of greatest variation within the polygon. The variation in the u and v coordinates of a given polygon (for example, the variation in x and y coordinates for a polygon with an xy major projection plane) is guaranteed to be greater than or equal to the variation of the polygon along the third axis (the w axis). Therefore, if a left edge traverses the length of one cell in the v direction, it will not traverse more than the length of one cell in the w direction.

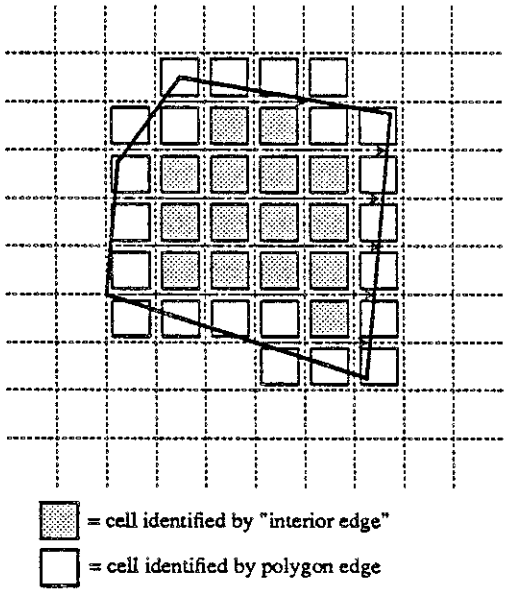


Figure 4.2 – Cells that a polygon passes through

This observation leads to the final definition of our polygon insertion algorithm. As we trace the left edges of the polygon, we note whenever we enter a cell with a new v coordinate. At this point, left-edge tracing is temporarily suspended, and a line segment is constructed across the interior of the polygon. This line segment lies in the constant v plane that we just crossed. As we trace the line segment across the polygon, the polygon is inserted in cells on both sides of the constant v plane.

Note that some cells may be identified more than once using this method (once during edge tracing, and again during interior tracing). Before inserting the polygon in a cell, we check to make sure it isn't already there. This check is simple, because we insert only one polygon at a time, and the current polygon will appear at the head of the list of polygons in a cell if it has already been inserted.

The above algorithm is heavily influenced by the data structures used to represent polygons and vertices. Each polygon data structure contains a circular list of pointers to the vertices that define the polygon. We thought it would be faster to process the polygon in a circular fashion (right edges followed by left edges and interior line segments) than to rearrange the circular list and process right, left, and interior line segments simultaneously during a single vertical scan of the polygon. The latter is a traditional approach in similar scan-line processing algorithms, but requires tracing of multiple edges simultaneously. This is harder to program, and does not present any apparent advantages.

Floating-point arithmetic is used to trace all edges and interior line segments. This is in contrast to the tracing of rays during rendering, where integer arithmetic is used for speed. There are two reasons for this: First, accuracy is crucial during insertion. Small errors that can be tolerated during ray-tracing can cause serious problems during insertion. For example, if an edge

is traced from vertex *v1* to vertex *v2*, but it fails to identify the cell containing *v2*, it is difficult to determine the cause and magnitude of this error. Similar errors can be caused by concave or non-planar polygons as well as numerical error. In the best case, no image degradation will result. In the worst case, the polygon will not be inserted in the appropriate cells, and holes or cracks will be visible in the final image. Furthermore, rendering performance will be degraded (albeit slightly), because the polygon may be inserted in cells that it does not really pass through.

Secondly, edges are typically short, and the overhead of converting floating-point values to integers negates any advantage gained by performing a few computations with integers (on most modern computers).

There are numerous calculations that can be performed just once or incrementally. For example, the direction of the interior line segments can be initialized just once since polygons are planar. The cell coordinates of cells pierced by right edges are saved so that we can quickly calculate the number of cells that must be traced by an interior line segment for a particular constant *v* plane.

The insertion algorithm we have described is a fast and accurate solution to the polygon insertion problem, but it turned out to be tricky to achieve a flawless implementation. Approximately one-third of the development time spent on the renderer was used to ensure the speed and soundness of the insertion algorithm. Entirely different algorithms for performing the insertion of polygons can also be envisioned. A concurrent planar sweep of all vertices was investigated briefly during research of the insertion algorithm. Although such an approach has certain merits, the main disadvantage is the requirement that each vertex store a list of the polygons it is used in. The additional memory required by this information (which is not used elsewhere in *UgRay*) discouraged further investigations of this method.

5. Tracing Rays through the Spatial Subdivision

Another essential ingredient required for fast operation of the renderer is a fast method for identifying which cells in the spatial subdivision are pierced by a ray, a polygon edge, or an interior line segment (see the preceding section). When the spatial subdivision was first researched as a technique to accelerate ray-tracing [Gla84], a ray was intersected with each side of a cell to determine the point where it left the cell, and hence, the next cell it entered. Because the intersection calculation with the cell wall was performed using floating-point arithmetic, the function that propagated rays from cell to cell required significant overhead.

Shortly afterwards, numerous researchers (including the author) realized that the cell identification problem is nearly equivalent to the pixel identification problem that arises when lines are displayed on a raster device. The pixel identification problem simply asks which pixels should be displayed to represent a continuous line. Solutions based on incremental calculation and integer arithmetic were discovered by Bresenham and are now universally used in line generators. With slight modifications (we do not need to worry about line density or symmetry), the same algorithms can be used to identify cells pierced by a line in three dimensions.

The *Digital Differential Analyzer* (*DDA*) is a popular method for generating lines on raster devices [NeS79]. We will describe a modified DDA algorithm that identifies cells in two dimensions, then extend it to three dimensions.

The two-dimensional DDA algorithm begins by identifying the axis of greatest change along the line. This is called the *driving axis*, and the other axis is referred to as the *passive axis*. The distances to the next transitions (cell walls, in our case) along each axis are computed. For ease of comparison, both distances are measured along the driving axis. In other words, the *distance counter* associated with a particular axis indicates how far we can travel along the driving axis before a step is required in the direction of the axis of interest (figure 5.1).

At each step, distance counters are compared, and we step to the cell in the direction of the axis associated with the minimum distance counter. The distance counters are then updated. If the comparison indicates a step in the direction of the driving axis, we simply subtract one unit from the passive axis distance counter (the driving axis distance counter does not need to be changed because it maintains the same distance to the next transition in the new cell). If we step in the direction of the passive axis, a precomputed value is added to the passive axis distance counter. This precomputed value is equal to the distance traversed along the driving axis per unit distance traversed along the passive axis (simply the inverse slope of the line).

The extension to three dimensions is easy. One axis is identified as the driving axis, and the other two axes are passive axes. All three distance counters are measured along the driving axis, and a three-way comparison is performed during each iteration to determine which axis to step. If a step is taken along the driving axis, one unit is subtracted from each of the passive axis counters. If a step is taken along a passive axis, the inverse slope value for that axis is added to its distance counter. A more abstract discussion of three-dimensional DDA can be found in [FTI86].

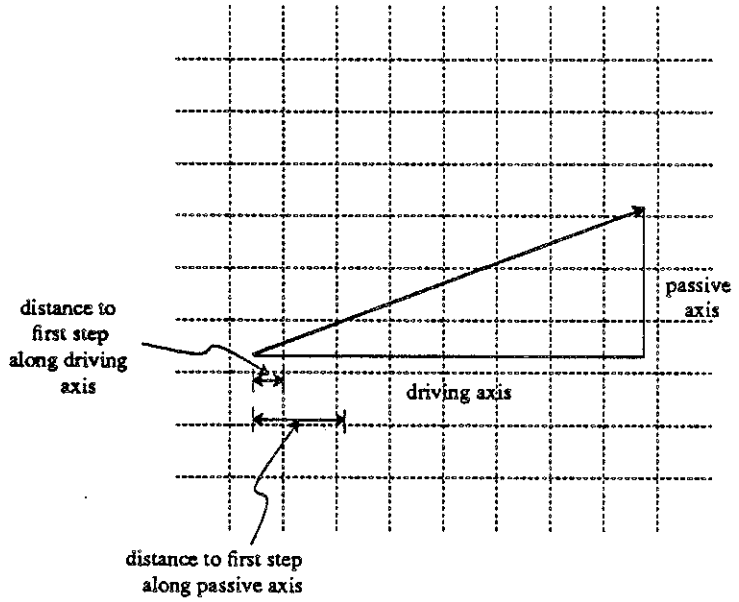


Figure 5.1 – Tracing a line with DDA

Software (and hardware) line generators use integer arithmetic when performing operations similar to those mentioned above. For a display with finite resolution, it is only necessary to maintain enough precision so that the distance counters accumulate less error than half a pixel when a line is drawn across the full screen. In our case, precision is required to ensure that a ray is propagated through the proper cells, and hence, the correct polygons are tested for intersection.

Each distance counter can be divided into two parts: an integer part and a fractional part. Together, these values indicate the distance in terms of cell units to go before the next step is required. The integer and fractional parts can be maintained in separate variables, but speed can be gained if they can be represented with adequate precision in a single 32-bit integer variable. The range of values that must be represented is -1 cell unit (the distance counters associated with the passive axes can become negative temporarily during an intermediate transition) to 255 cell units (the maximum number of cells we permit along an axis), exclusive. Instead of wasting a bit on the sign bit, we can bias this range by adding one cell unit. Except for an extra addition in the initialization step, this bias does not affect any other part of the algorithm since it depends only on comparsions of the relative magnitude of each distance counter. The range of biased values is 0 through 256, exclusive. The integer part therefore requires 8 bits. The remaining 24 bits of a 32-bit integer variable can be allocated to the fractional part of the distance counter.

We will now analyze the worst-case error. To facilitate fast updates, the inverse slope calculated for each of the passive axes is converted to the 24-bit fractional representation used by the distance counters. Suppose that this inverse slope has been truncated so that the least significant bit of the 24-bit fraction is incorrect. In the worst case, the inverse slope will be added to the corresponding distance counter 255 times as a ray is traced along an axis containing the maximum number of cells. Therefore, an error of $\frac{255}{2^{24}} = 1.52 \cdot 10^{-5}$ cell units will accumulate. If one

of the other distance counters has a value within this tolerance of the given distance counter (this happens when a ray passes very close to an edge or a corner of a cell), the DDA algorithm may step along the axes in an incorrect order. An incorrect cell will be identified, and a correct cell will be missed.

What are the consequences of missing a cell by this small amount? The cell that is identified instead may contain a different list of polygons, therefore a different series of intersection tests will be performed. It is likely, however, that any polygons that would have been intersected in the correct cell will also be contained in the incorrect neighboring cell. The worst scenario occurs when this is not the case: a polygon that should intersect the ray ends close to the boundary separating the correct cell from the incorrectly identified neighboring cell. If the ray does not pass through any other cell containing this polygon, the intersection test will never be performed, and a crack could result in the final image. This is, however, unlikely. Furthermore, the numerical conditions that produce such a crack typically do not persist over a significant area in the final image, so the defects are usually diminished by anti-aliasing.

For these reasons, we decided that the speed increase to be gained by representing distance counters and inverse slope values as single-precision integers was worth the trade-off in precision. During months of use, we have observed no defects in our images caused by this precision trade-off. If precision becomes a problem in the future, it can be increased by limiting the subdivision to 127 cells per axis, and allocating the additional bit to the fractional part. Few scenes require subdivision as fine as 255 cells per axis (this requires many megabytes of memory besides).

6. The Ray/Polygon Intersection Test

Up to this point, this report has concentrated on ways to reduce the number of ray/polygon intersections that must be calculated in order to identify the surface that is visible along any particular ray. Effective techniques reduce this number from hundreds of billions (for a complicated scene) to merely tens of millions.

But ten million executions of any routine is still an expensive proposition. Every operation within the intersection routine must be carefully considered and optimized. We are aware of at least one ray-tracing system in which the intersection routine is written in assembly language to achieve good performance. To maintain portability between different machines, we have resisted the temptation to use assembly language or machine-dependent optimizations in this critical routine. We have examined many other optimization techniques, however. Some of these have resulted in significant speed improvements; others have had little effect and were discarded. In this section, we report on some of the optimizations we have tried, and which have been successful in reducing rendering time.

6.1. Ray/Plane Intersection

The ray/polygon intersection calculation consists of two steps: First, the equation of the ray is intersected with the equation of the plane containing the polygon. Second, this point of intersection is determined to be inside or outside of the edges that define the polygon.

The intersection of the ray with the plane containing the polygon is straightforward (figure 6.1). Suppose that we are given the origin of the ray, $\vec{R}_o$, and the direction of the ray, $\vec{R}_d$. The point of intersection is some point $\vec{P}$ on this ray:

$$\vec{P} = \vec{R}_o + t\vec{R}_d, \quad t \geq 0 \tag{6.1}$$

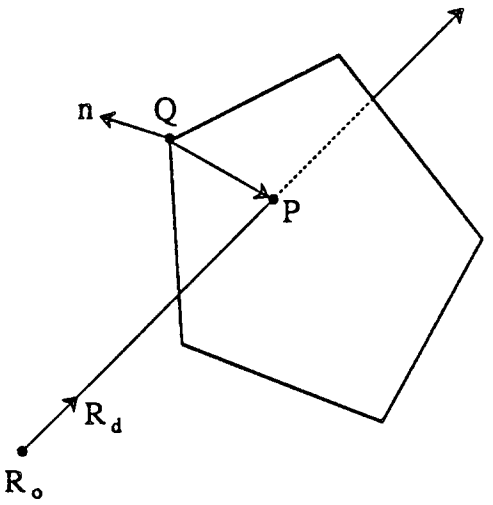


Figure 6.1 – Ray/polygon intersection

The point $\vec{P}$ must also lie in the plane, so that the difference of $\vec{P}$ and a point $\vec{Q}$ known to be in the plane (one of the polygon's vertices, for example) is a vector perpendicular to the plane's normal vector $\hat{n}$. In other words, the dot product of the vector $\vec{P}-\vec{Q}$ and the normal vector is zero:

$$(\vec{P}-\vec{Q})\cdot\hat{n}=0$$

$\vec{Q}\cdot\hat{n}$ is constant for the given plane (it equals D in the $Ax+By+Cz+D=0$ formulation of the plane). Therefore, the equation can be rewritten

$$\vec{P}\cdot\hat{n}=D$$

Substituting the first equation for $\vec{P}$ and solving for t, we get

$$t=\frac{D-\vec{R}_o\cdot\hat{n}}{\vec{R}_d\cdot\hat{n}}$$

If the denominator of the above equation is zero, the ray is parallel to the plane, and no intersection is possible. The intersection test can also be aborted if t is negative (the intersection lies behind the origin of the ray), or if t is greater than the least value of t computed for a previous polygon that intersects the ray (we are only interested in finding the closest intersection). Otherwise, t can be substituted in equation (6.1) to find the coordinates of the intersection.

Now let's consider possible optimizations. The calculation of t requires 12 floating-point operations (each dot-product requires 5 floating-point operations). In certain cases, less computation is necessary. For example, if the origin of the current ray is the eyepoint, which lies at the origin of the coordinate system for a perspective view, then $\vec{R}_o\cdot\hat{n}=0$. If we take this into account, the computation of t can be reduced to 6 floating-point operations. Unfortunately, this optimization applies only to primary rays, so the acceleration accomplished by this optimization is most noticeable when the renderer is *ray-casting* (not producing any secondary rays).

A similar optimization is possible for primary rays generated for an orthogonal view. In this case, each ray has an origin $\vec{R}_o=(x,y,0)$ (the transformation to eye coordinates places the scene entirely in front of the eye) and direction vector $\vec{R}_d=(0,0,1)$. Using these facts, one addition and one multiplication can be eliminated from $\vec{R}_o\cdot\hat{n}$, and $\vec{R}_d\cdot\hat{n}=\hat{n}_z$. The number of floating-point operations is reduced to 5.

The potential savings of these computations (and several other special cases that can be applied to primary rays generated for an orthogonal view) led us to separate the ray/polygon intersection routine into four different routines. One routine handles primary perspective rays, another handles primary orthogonal rays, the third handles reflected and transmitted rays, and the fourth handles shadow rays. In the latter two routines, the full computation of t must be performed.

6.2. The Inside/Outside Test

Once the point of intersection between a ray and the plane containing a given polygon is found, it is necessary to determine whether the point lies inside the edges of the polygon. If so,

an intersection is recorded. Otherwise, the ray misses the polygon.

We attempted to develop new methods for calculating the inside/outside test in the plane containing the polygon, but it is difficult to come up with fast, numerically accurate methods that can compete with traditional methods. Traditional methods project the polygon and the intersection point onto some plane where the inside/outside test can be conveniently performed. The main question to be resolved, therefore, is which plane to project the polygon onto.

An obvious and popular choice is to project the polygon onto a plane perpendicular to the given ray (figure 6.2). The point of intersection projects onto a point that we will define as the origin of a coordinate system u,v in this plane. The inside/outside test can then be performed simply by counting the number of times the projected edges of the polygon cross the positive u axis in the projection plane (the u axis is only an example – any ray contained in the plane starting at the origin will work). If the number of crossings is odd, the point is inside the polygon. If the number of crossings is even (or zero), the point is outside the polygon. Care must be taken to correctly count vertices and edges that fall exactly on the u axis.

Let us analyze this algorithm a little more closely. To define a projection plane perpendicular to a ray, it is necessary to generate the u and v basis vectors in the plane. The u vector can be generated simply by picking coordinates that will produce a zero dot-product with the ray vector. The v vector is the cross-product of the u and ray vectors. This cross-product requires 9 floating-point operations, but since it is only done once per ray, the overhead is not too worrisome.

Two dot-products (10 floating-point operations) are necessary to project each of the polygon's vertices onto the projection plane. Then we simply examine the v coordinate of successive vertices. Whenever a sign change is noted in the v coordinates of a pair of vertices connected by an edge, the edge may cross the positive u axis (figure 6.3). If both vertices have

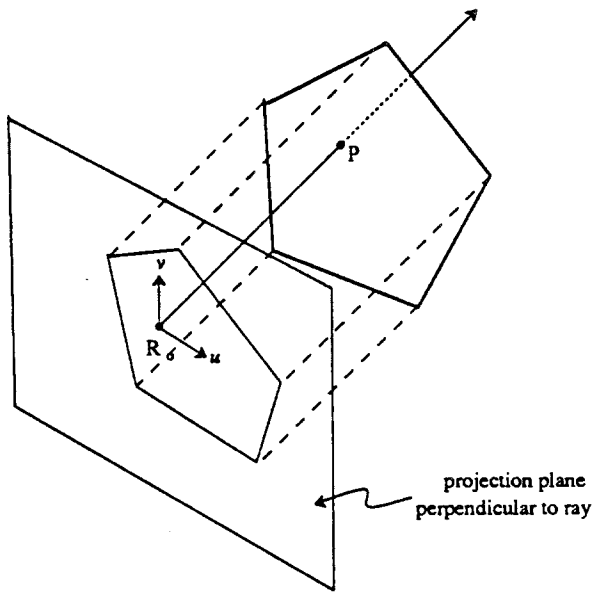


Figure 6.2 – Inside/Outside test in a plane perpendicular to the ray

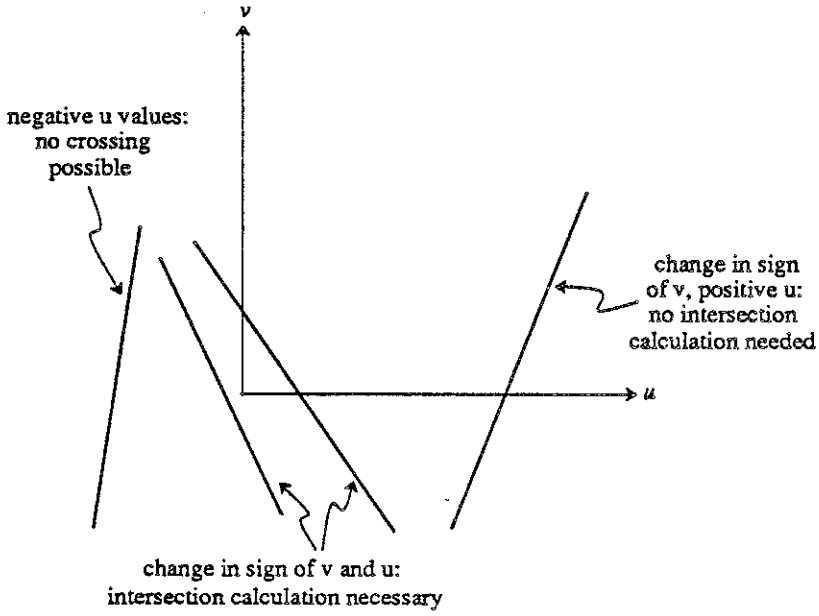


Figure 6.3 – Finding polygon edge crossings

positive u coordinates, a crossing occurs. If both vertices have negative u coordinates, no crossing occurs. Otherwise, the sign of the u coordinates must change between the two vertices. Here, 5 floating-point operations must be performed to calculate the u coordinate of the intersection of the edge and the u axis. If this value is negative, no crossing occurs; otherwise, a crossing is counted. Once again, the boundary conditions must be treated with care to achieve correct results.

In total, at least 30 floating-point operations are necessary to perform an inside/outside test on a triangular polygon. For polygons with more edges, each edge must be examined to complete the inside/outside test. Also, no method is available to trivially reject a polygon that lies entirely within one quadrant of the projected coordinate system. Such a trivial rejection method is desirable to limit the amount of computation spent on polygons that have no possibility of containing the intersection point.

Contemplation of these factors led us to consider projection of the polygon onto one of the basis planes in the eye's coordinate system (figure 6.4). In this case, the polygon and the point of intersection are projected onto the xy, xz, or yz plane in which the polygon is least distorted (its projection covers the most area). This plane is called the *major projection plane* for a particular polygon. A projection of this sort is attractive because it can be done with no floating-point operations. This method doesn't even require additional memory to store the projected coordinates of the vertices. The projected coordinates are accessed by appropriately indexing the original coordinate values. One of the three coordinates is simply ignored.

This projection method has another attractive feature: since a polygon is always projected onto the same plane, no matter what ray is being intersected, further preprocessing is possible. Let us refer to a local u,v coordinate system in the major projection plane (in the xy plane, the x axis is the u axis, and the y axis is the v axis, for example). During preprocessing, we can find the

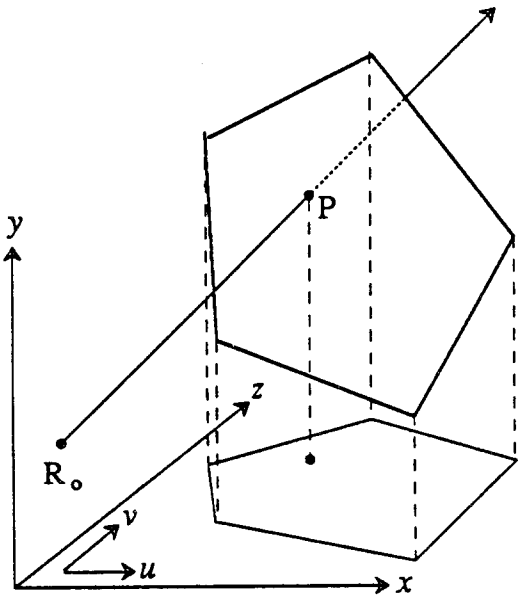


Figure 6.4 – A faster projection for the Inside/Outside test

minimum and maximum *u* and *v* coordinates of the projected polygon. When the point of intersection between the ray and the plane containing the polygon is projected onto the major projection plane (again, simply by ignoring the appropriate coordinate), we can quickly check to see if it lies within the bounding rectangle of the projected polygon. If not, no intersection is possible and further testing is avoided.

At first, we wondered if this bounding rectangle test was necessary when used with the spatial subdivision. After all, the spatial subdivision eliminates polygons from consideration that are far from the current ray. Interestingly enough, although the savings due to the bounding rectangle test diminish as finer spatial subdivisions are used, we never discovered a scene where this test didn't result in some savings.

The new projection also allows us to make some special optimizations for convex polygons. If we restrict polygons to be convex, only a few edges must be examined to determine if the intersection point lies inside or outside. Because a polygon always projects to the same major projection plane, we can identify *right* edges and *left* edges in the projection. A point inside the polygon must lie to the left of one of the right edges, and to the right of one of the left edges.

The convexity restriction may seem severe at first, but most of the polygons used in typical scenes are convex. Concave polygons and polygons with multiple contours (holes) can be submitted to a preprocessing program (such as *UgTess* [Shi86]) to be redefined as several convex polygons. It is conceivable that *UgRay* will be extended to handle concave polygons in the future, but it is likely that convex polygons will continue to be treated as an important special case for maximum speed.

6.3. Further Edge Optimizations

Now let us consider the calculation that determines whether an edge lies to the left or to the right of the intersection point. If a and b are vertices defining an edge, and if the v coordinate of the intersection point p lies between the v coordinates of a and b, then the horizontal distance from p to the edge is

$$a_u + \frac{p_v - a_v}{b_v - a_v}(b_u - a_u) - p_u$$

If the above expression is greater than zero, the edge lies to the left of the intersection point. Otherwise, the edge lies to the right.

Notice that the above calculation requires 6 or 7 floating-point operations (the last subtraction can be changed into a comparison if a comparison is less expensive than a subtraction on a particular machine). This compares to 5 floating-point operations for edge-testing in the old routine. The additional floating-point operations are required to translate p to the origin (the projected point already lies at the origin in the old routine). Considering the speed of our intersection test so far, it seems reasonable to try to avoid this calculation whenever possible.

Because of the convexity restriction, we know the general orientation of edges in various parts of a polygon. Therefore, a quick comparison with the u coordinate of just one of the vertices of an edge will indicate when an edge lies entirely to the left or to the right of the intersection point, thus eliminating the need for a more general calculation. This optimization seemed particularly attractive on the 68000 microcomputer that *UgRay* was developed on. For this particular computer, a floating-point comparison is 3 times faster than a floating-point arithmetic operation. Therefore, a single comparison could alleviate the need to perform computations that are nearly 20 times more expensive than the comparison.

To our great surprise, optimizations based on these facts produced negligible performance improvements on the 68000 system, and actually decreased rendering speed slightly in some cases on our Vax systems. Apparently, the overhead of performing the extra comparison during every execution of this critical routine negated the speed gained in cases where additional computation was bypassed. Closer examination of these results revealed that this optimization is used infrequently for intersection points that have passed the earlier polygon bounding rectangle test. If the bounding rectangle test is removed, we expect that the edge testing optimization will produce better results.

We attempted similar optimizations that reduce the number of comparisons necessary to intersect a ray with a triangle (a common polygon that deserves special attention), but these efforts also produced negative results. Therefore, we believe that further optimizations to the ray/polygon intersection routine must be examined carefully and critically. Optimizations that initially appear promising often fail to perform well in this routine unless they reduce computation significantly in a large percentage of the calls made to the routine.

6.4. Speed Analysis of the Intersection Routine

The intersection routine we have proposed in this section is susceptible to numerical error. Therefore, a speed versus accuracy trade-off must be made. Before we describe the causes of this numerical error and methods to cure it, let us examine more closely the speed improvement to be expected from the new routine.

In both the old routine (projection onto a plane perpendicular to the given ray) and the new routine, the parameter t (related to the distance along the ray) must be calculated. This requires 12 floating-point operations in the general case, and 3 comparisons to determine if the ray is parallel to the plane containing the polygon, or if the intersection is *behind* the origin of the ray, or if a closer intersection has already been found.

In the new routine, t is substituted in equation (6.1) to find two of the coordinates of the intersection point, which will be used in the inside/outside test. This requires 4 floating-point operations. In the old routine, this calculation can be delayed until after the inside/outside test has been satisfied, because the intersection point always projects to the point (0,0) in the projection plane. The new routine expends 4 additional comparisons at this stage to determine if the intersection point lies within the bounding rectangle of the projected polygon. If the point lies outside of the bounding rectangle, the intersection fails. This trivial rejection case has been detected using a total of 14 or 16 floating-point operations (depending on which axis of the bounding rectangle the point lies outside of) and between 4 and 7 comparisons. No trivial rejection is possible in the old routine.

The polygon is now projected onto the projection plane. The old routine requires 10 floating-point operations per vertex. For the best case (a triangle), 30 floating-point operations are required in all. The new routine requires no floating-point operations during this stage.

The inside/outside test requires 1 comparison for each edge to determine if the edge crosses the u axis. An additional comparison and 5 (old routine) or 6 (new routine) floating-point operations are required for edges that cross the u axis. The old routine must, on the average, look at all the edges in a polygon, even if it is known to be convex. If polygons are required to be convex, the new routine needs to look at less than all the edges, on average, because pointers to the top and bottom vertices allow the algorithm to skip edges once the appropriate edge is identified from the left or right edges of the polygon. The effect of this optimization depends on the shape of a particular polygon and the location of the point of intesection. Since this is difficult to measure, we have not accounted for any savings in this analysis. For a triangle, the old routine requires 5 comparisons and 10 floating-point operations. The new routine requires 5 comparisons and 12 floating-point operations.

If the intersection point fails the inside/outside test, the old routine exits with a total expenditure of 52 floating-point operations and 8 comparisons. The new routine requires 26 floating-point operations and 12 comparisons.

Finally, if the intersection point passes the inside/outside test, its coordinates must be found by substituting t in equation (6.1). This calculation might be deferred in a simple ray-tracing renderer until the closest intersection point is known, but discovery of a closer intersection is rare in *UgRay* because of the spatial subdivision. The old routine performs 6 floating-point operations

to calculate the coordinates. The new routine already has calculated two of the coordinates and requires 2 floating-point operations to find the third. This brings the totals to 58 floating-point operations and 8 comparisons for the old routine; 28 floating-point operations and 12 comparisons for the new routine.

If a floating-point arithmetic operation and a floating-point comparison take about the same amount of time, we can see that the new routine computes the intersection with a triangular polygon about 1.7 times faster than the old routine. If the point lies outside of the bounding rectangle of the polygon, the new routine can reject it 2.6 times faster.

The above analysis has been carried out for a three-sided polygon, because this is the best case for the old routine. For a convex four-sided polygon, the new routine computes a successful intersection nearly twice as fast as the old routine, and trivially rejects 3.1 times faster (figure 6.5). Since *UgRay* typically spends 30 to 50 percent of the total execution time in the intersection routines, these optimizations will accelerate rendering speed by 50 to 100 percent (depending partly on the average number of edges per polygon).

6.5. Error Analysis of the Intersection Routine

The major disadvantage of the intersection routine proposed in this section is the fact that important topological information is lost when a polygon is projected onto its major projection plane for inside/outside testing. To understand this, consider an edge that is shared between two polygons. Using the old inside/outside test, this edge is projected onto a plane perpendicular to the given ray. Although this projection is performed separately for the two polygons that share the edge, the edge is projected identically in both cases. The projected point of intersection will lie unambiguously to the left or to the right of the projected edge.

Unfortunately, this result can no longer be guaranteed using the new routine. The polygons that share the edge may project onto different planes, and thus the shared edge will be projected differently in each case.

Let us consider how this might produce erroneous results along a silhouette edge. The silhouette edge is shared between two polygons: the *front* polygon is visible on the front side of

case	Number of floating-point operations		speed increase
	old routine	new routine	
trivial rejection for triangle	60	23	2.6
average successful triangle	66	40	1.7
trivial rejection for quadrilateral	71	23	3.1
average successful quadrilateral	77	41	1.9

Figure 6.5 – Speed of Inside/Outside test

the object, and the *back* polygon is not visible. A ray passing near a silhouette edge should either intersect both polygons, or it should intersect neither. If the intersection lies arbitrarily close to the edge, a small numerical error might cause the point of intersection to lie outside of the edge when the inside/outside test is performed on the front polygon. If this is the case, the old intersection routine guarantees that the back polygon will also be missed, since the inside/outside test is performed on the same edge. But the new routine might perform the inside/outside test for the back polygon in a different plane, and the numerical error will not be the same. If the test succeeds, the ray will have missed the front polygon and hit the back polygon. Therefore, the back polygon may be visible in some pixels along the silhouette edge.

A similar problem can occur for non-silhouette edges that are projected into the *same* projection plane (figure 6.6). In this case, the ray should hit either the left polygon or the right polygon. Because the ray passes unambiguously to the left or right of the edge, this result is guaranteed using the old routine. In the new routine, however, small numerical errors can cause the point of intersection to fall inside both polygons, or miss both, since two independent ray/plane intersections are calculated. In the former case, either polygon may be selected for display. But if this choice varies from scan-line to scan-line, a ragged edge will result. If the ray misses both polygons, an even more bothersome defect is produced: a crack between the polygons.

These shortcomings are not easily cured. Various attempts were made to recover the topological information that is lost. Any time a ray comes close to an edge, the topology of the scene must be examined to produce correct results. Unfortunately, this requires time that negates some of the speed advantages we had hoped to gain with use of the new routine.

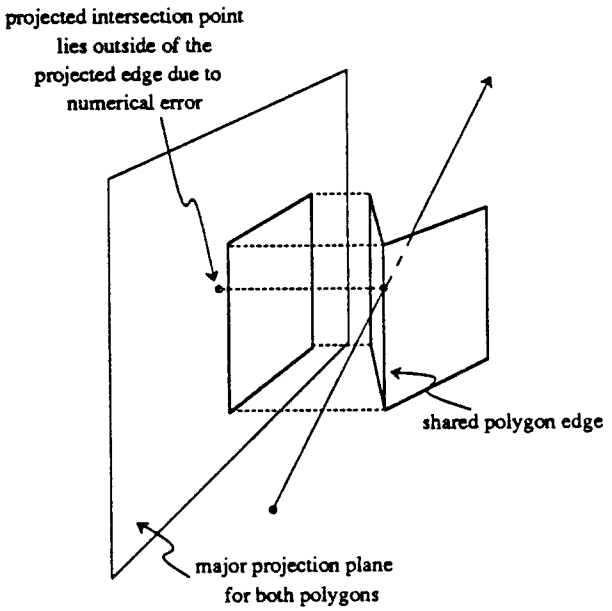


Figure 6.6 – Causes of cracks between polygons

However, many of these problems disappeared once anti-aliasing was incorporated in *UgRay*. The reason is that many of these defects are very localized. The numerical inaccuracies that cause them do not persist over more than one sample. Therefore, many static images have been produced with no visible defects. Animation presents more serious challenges, however. If many defects such as cracks and "sparkling silhouettes" are apparent when *UgRay* is used to produce animated sequences, the slower and more robust intersection algorithm may need to be restored.

7. Shading

Once a visible surface has been identified using the mechanisms described in previous sections, the intensity and color of light being reflected from or transmitted through the surface at the point of intersection must be determined. We will refer to the collection of routines that perform these calculations as the *shader*. The methods used by the shader are of central importance in determining the quality and realism of the final image. Because the shader is responsible for generating additional rays to determine shadows, reflections, and transmitted light, its design will also have a major impact on the execution time of the renderer.

The development of methods to accurately and efficiently compute shading of surfaces has been an area of active research for many years. The most realistic shading models require huge amounts of computation that can even overwhelm the ray-tracing visible surface calculation.

We have tried to strike a compromise between realism and speed during the development of *UgRay*. In this section, we will briefly discuss some of the components of an ideal shading model, and the approximations to reality we have implemented to achieve reasonable performance.

7.1. Types of Rays

During our discussion of the shader, we will often distinguish between two different types of rays: *viewing rays* and *light rays*. Viewing rays originate at the eye, and are traced through the scene as they are reflected or transmitted by surfaces in the scene. Light rays originate at each light source and are traced into the scene. The distinction between these ray types is somewhat artificial, because ideally we would have only one type of ray. Light rays would originate at the light sources, be reflected by surfaces in the scene, and some of them would eventually hit the eyepoint, when the appropriate color would be displayed. This, of course, is the way it happens in nature. Unfortunately, a corresponding computational model would be absurdly wasteful: billions of rays would be generated, but only a small percentage would ever reach the eye.

The strategy employed by *UgRay* is to trace viewing rays from the eye to identify visible surfaces in the scene (including reflections and transmission through transparent surfaces). For each visible surface point, a single light ray is traced from each light source to determine the illumination at the point. (In reality, we trace the light ray backwards from the surface point towards the light source, and refer to this as a *shadow ray*.) This produces reasonable images without expending excessive amounts of computation.

7.2. Reflection and Transmission of Viewing Rays

A viewing ray that strikes a reflective or transparent surface causes new viewing rays to be generated in the directions of reflected and transmitted light. Although these new rays can be thought of as parts of the initial ray, they are given unique identifiers and treated independently of the parent ray.

It is important to limit the depth of recursion when viewing rays are reflected back and forth between several shiny surfaces in the scene. Imagine two parallel perfect mirrors in a scene. Rays can be reflected between these mirrors indefinitely if 100% of the ray energy is reflected at each surface and if there is no atmospheric attenuation (fog). The renderer is prevented from performing infinite recursion in this situation by simply limiting the maximum number of bounces to be traced. A *depth* is associated with each ray, and a particular ray is not reflected if it has already achieved the maximum allowable depth. Maximum depths of between 5 and 10 seem reasonable for most scenes.

UgRay incorporates another depth-limiting strategy, which is usually more effective than the simple cut-off stated above. Each time a ray is reflected (or transmitted), the contribution of the reflected ray is estimated. If the coefficient of specular reflection is 0.3 at a given surface, for example, the reflected ray will contribute 30% to the final color of the surface. If the new ray strikes another reflective surface that has a specular reflection coefficient of 0.2, another reflected ray can contribute only $0.3 \cdot 0.2 = 0.06$ to the color of the initial surface. When the contribution of a ray falls below a certain predetermined threshold, no further reflection rays are generated. The threshold value depends somewhat on the accuracy of the device that the image will be displayed on. Since *UgRay* eventually outputs red, green, and blue components with 8 bits of resolution, it is wasteful to trace any ray that contributes less than 1 part in 256 to the final image. If a particular display device cannot display 16 million colors simultaneously (8 bits for each of red, green, and blue), the threshold may be increased further. Note that fog or attenuation of a ray passing through a translucent material can also diminish the contribution made by the ray.

Rendering of transparent surfaces is a particularly expensive operation. Since transparent surfaces are usually reflective, a viewing ray that strikes a transparent surface generates two new rays: one in the direction of ideal reflection, and one in the transmitted direction. Each of these new rays may also hit a transparent surface, requiring another splitting of rays. This process may continue, forming a fully-expanded binary tree of rays that terminates only when the maximum ray depth is reached on each branch. For a maximum depth of 5, $2^5 - 2 = 30$ new rays may be generated from a single primary ray. By comparison, a maximum of 4 new rays can be generated in a scene containing reflective objects that are not transparent.

7.3. Light Incident upon a Surface

Researchers have generally characterized light reflected from a surface as containing two components [CoT81]. The *diffuse* component accounts for internal scattering of incident light, which is radiated equally in all directions. The *specular* component accounts for reflection of light from a smooth, shiny surface. The specular component is highly directional and depends on the angle of the incident light.

To correctly model the color and intensity of light reflected both diffusely and specularly from a given surface, we must first determine the color and intensity of *all* light incident upon the surface. Light can reach the surface in three ways: it can come directly from a light source, it can be indirectly reflected toward the given surface by other surfaces in the scene, or it can be transmitted through a transparent object in the scene (figure 7.1).

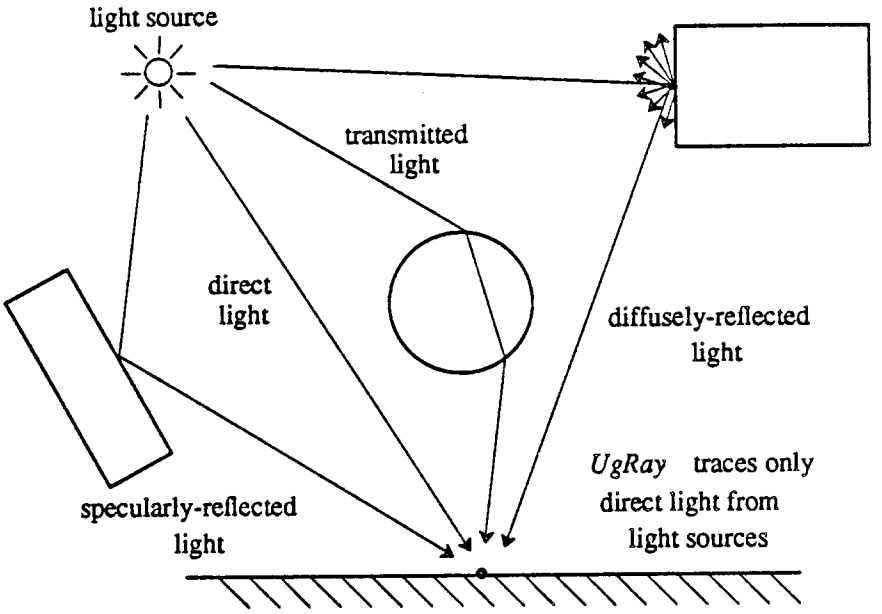


Figure 7.1 – Light incident upon a surface point

Light coming directly from a light source is fairly easy to model. This case will be considered in more detail shortly.

Light that is reflected indirectly from other surfaces in the scene is difficult and expensive to model accurately. One approach is the *radiosity* method described in [CoG85, ICG86]. The radiosity algorithm divides the entire scene into small patches. A hidden surface calculation is performed from the location of each patch to determine which patches in the scene are *visible* from the current patch. The solid angle subtended by patch B from patch A determines how much light B will reflect towards patch A. If there are n patches in the scene, it is necessary to solve n equations in n unknowns to calculate the light incident upon each patch. Since many patches are required to accurately model a complicated scene, the resulting system of equations can be large indeed. Many of the elements of the matrix are zero, however, so sparse matrix solution techniques offer some hope of solution.

The images produced by the radiosity technique are very realistic. They contain fuzzy shadows and bleeding of color from one surface onto another (*i.e.*, the pink color of a white wall next to a red wall). Unfortunately, the large expense of computing these effects can be justified only in a few applications. The calculations become even more unwieldy when indirect specular reflection of light must be modeled as well as diffuse reflection. Since specular reflection contains much higher spatial frequencies than diffuse reflection, many more patches are required to model it accurately, and computation times become unreasonable.

Another method that models light reflected between surfaces is a modified ray-tracing technique in [Kaj86]. When a visible surface is identified, a random collection of rays are created that sample the integral describing light incident upon the surface from all directions in the hemisphere above the surface. This technique is also expensive, and produces noisy images.

Presently, all known methods for modeling the interreflection of light between surfaces are too expensive for most applications. A traditional approximation of indirect lighting effects is to introduce an *ambient* term in the shading equation. The ambient term is simply a constant added to the intensity of light incident upon a surface, regardless of the orientation of the surface. The result is that surfaces that receive no light directly from a light source are slightly illuminated by *ambient light* instead of being completely black.

The final contribution to the light incident upon a surface is light transmitted through transparent objects in the scene towards the current surface. If the transmitting material does not refract light that passes through it, this is simple to model (in the context of ray-tracing) with extensions to the basic shading model. The intensity of light passing through a transparent material is attenuated as an exponential function of the distance the light travels through the material and the translucency of the material.

Unfortunately, problems arise when we allow transparent materials that refract light. A glass lens, for example, will form *caustics* (areas where light is focused after passing through the lens). In a general case, it is a difficult problem to determine if a given surface point is within an area being illuminated by light refracted through a transparent medium. This problem is solved for the specific case of transparent spheres in [Ina86].

A more general solution uses *backward ray-tracing* to trace light rays from each light source into the scene [Arv86]. These light rays are refracted and reflected by the surfaces in the scene, depositing light energy at each surface point, thus creating an *illumination map* that is used during the subsequent tracing of rays emanating from the eyepoint. This method is not efficient (many light rays are traced that will have no effect on the visible scene) and can exhibit sampling problems when a reflective or refractive curved surface spreads light rays over a large area.

Caustics can also be calculated using probablistic distributions of rays to evaluate the integral corresponding to the light incident on a surface over the hemisphere above the surface point [Kaj86]. As mentioned earlier, this is an expensive technique to use in situations where the added realism is not worth an increase in computation time by at least an order of magnitude.

In *UgRay*, light passing from a light source through a transparent object is not refracted in order to avoid these problems (but viewing rays emanating from the eyepoint *are* refracted). The sacrifice in realism is noticeable for a sphere, but is not objectionable in the case of flat glass panels. The shadow cast by a transparent object varies according to the distance a light ray travels through the object and the angle at which the light ray encounters the surfaces of the transparent object. At grazing angles, most of the energy of the light ray will be reflected (depending on the surface properties of the object), and the shadow will therefore be darker.

7.4. Operation of the Shader

The shader is called to determine the color to be returned for each viewing ray that is cast. As we will see shortly, the shader may even call itself recursively when additional viewing rays are generated to determine reflections and transmission through a transparent surface.

The shader is given information about the direction and origin of a particular ray, the surface (if any) that the ray intersects, and the coordinates of the intersection. In the current

implementation of *UgRay*, the intersected surface is simply a pointer to a polygon.

If a ray leaves the spatial subdivision without intersecting any surface, the intersected surface will be a null pointer. In this case, the shader immediately returns the user-specified background color for the current ray sample.

The shader uses the following formula to calculate the color intensity returned from a polygon in the most general case:

$$\begin{aligned} \mathbf{I} = [\, & \mathbf{i}_a k_d \mathbf{c} \\ & + \left[\sum_{lights} \mathbf{i}_l k_d \mathbf{c} (\hat{n} \cdot \hat{l}) + \mathbf{i}_l k_s (\hat{n} \cdot \hat{h})^p (m\mathbf{c} + (1-m)) \right] \\ & + \mathbf{c}_s k_s (m\mathbf{c} + (1-m)) \\ & + \mathbf{c}_t k_t \mathbf{c} \,]\, e^{(d\,\mathbf{f})} \end{aligned} \tag{7.1}$$

where

$\mathbf{I}$ = the red, green, and blue intensities of the color returned

k_d = coefficient of diffuse reflection

k_s = coefficient of specular reflection

k_t = coefficient of transmission

$\mathbf{c}$ = color of the surface at the point of intersection

$\mathbf{c}_s$ = color returned by a specularly reflected ray

$\mathbf{c}_t$ = color returned by a transmitted ray

$\mathbf{i}_a$ = intensity of ambient light

$\mathbf{i}_l$ = intensity of light arriving from a light source

$\hat{n}$ = unit vector normal to surface at the point of intersection

$\hat{l}$ = unit vector pointing toward light source

$\hat{r}$ = unit vector in direction of ray

$\hat{h}$ = unit vector half-way between $\hat{r}$ and $\hat{l}$

p = constant determining the sharpness of specular highlights

m = *metalness* of the surface

d = distance between *origin* of ray (or point of last intersection) and current intersection

$\mathbf{f}$ = attenuation factor for fog or translucent material

Note that the above calculation must usually be performed separately for the red, green, and blue components of the final color. The values that vary between these three components in the current implementation of *UgRay* appear in bold type in the above formula.

We will now explore each part of the shading formula in greater detail, noting optimizations that can be made wherever possible.

7.5. Ambient Light

The first term of equation (7.1), $\mathbf{i}_a k_d \mathbf{c}$, calculates the intensity of ambient light diffusely reflected from a surface. As noted above, the intensity $\mathbf{i}_a$ of ambient light and the fraction of light $\mathbf{c}$ reflected from the surface are separated into distinct values for red, green, and blue. The coefficient of diffuse reflection k_d models surface *roughness*, specifying what percentage of light incident upon the surface is scattered diffusely in all directions. This value is between 0 and 1. A diffuse surface such as chalk will have a diffuse coefficient approaching 1. A mirror will have a value close to 0, because most of the light incident upon a mirror surface is reflected specularly rather than diffusely.

The intensity and color of ambient light that is diffusely reflected from a surface does not depend on the orientation of the surface or the angle from which the surface is viewed. Therefore, this term of the shading equation can be calculated just once for each color in the scene as a preprocessing step. This eliminates a few multiplications that would otherwise be performed redundantly during rendering.

The ambient term of equation (7.1) models only diffuse reflection of ambient light. It might be argued that specularly reflected and transmitted ambient light should also be accounted for. The original implementation of the shader included a term that computed the integral of ambient light specularly reflected as well, but this term tended to reduce the contrast of the final image and make shiny surfaces appear too bright. Because ambient light is only an approximation to indirect reflection effects, we eventually discarded calculations involving ambient light that did not enhance the realism of the final image (in our opinion).

7.6. Shadows and Direct Illumination by a Directional Light

The terms inside the summation in equation (7.1) compute the diffuse and specular reflection of light coming directly from a particular light source. Therefore, the summation must be iterated over each light source in the scene.

A light source contributes to the light intensity reflected from a surface in the direction of view if it illuminates the same side of the surface that the viewing ray hits. At the time the intersection calculation is performed, the sign of the dot product between the ray vector and the surface normal vector $\hat{n}$ is noted. If the dot product between $\hat{n}$ and a vector in the direction of a given light source generates the same sign, the light source *may* illuminate the surface being viewed. If the opposite sign is generated, the light source can illuminate only the opposite side of the surface, and it is ignored. This does not correctly model direct illumination passing through a transparent surface from the opposite side, and this deficiency should be corrected in a future revision of the renderer. A percentage of this light should be scattered in all directions if the

surface is not perfectly smooth. Therefore, the surface would appear to glow.

We must now determine if any other surface in the scene shadows the current surface point. In the current implementation of *UgRay*, this is accomplished by generating a new ray originating at the current surface point, and oriented in the direction of the light source. We will refer to this ray as a *shadow* ray (it travels in the opposite direction of a particular *light* ray that might illuminate the surface point). The shadow ray is submitted to the ray casting routine. If it intersects any opaque surface, the light from the corresponding light source is blocked, and no further diffuse or specular reflection calculations are performed for this light source.

The situation becomes more difficult when a shadow ray intersects a transparent object. As mentioned earlier, a refractive material should bend the shadow ray. Unfortunately, if we allow the shadow ray to be refracted, it will no longer correspond to the same light ray travelling in the opposite direction. The shadow ray will be bent away from the direction of the light source, and we will not know whether light from the light source illuminates the surface point or not (figure 7.2). To avoid these problems, the shadow ray is propagated through the transparent volume without bending. Each time the shadow ray encounters a surface, the amount of energy that would be transmitted through the surface for the corresponding light ray travelling in the opposite direction is calculated. (The relative index of refraction, which can be used to determine the coefficient of transmission, is inverted if the calculation is performed in the wrong direction.) The transmitted energy is also attenuated according to the distance that the shadow ray (or light ray) travels within a translucent material.

The intensity and hue of light from a light source can change if it is attenuated in the above manner. The altered hue and intensity of the light must be used in the diffuse and specular reflection calculations. It is this altered light intensity that is $\mathbf{i}_l$ in equation (7.1).

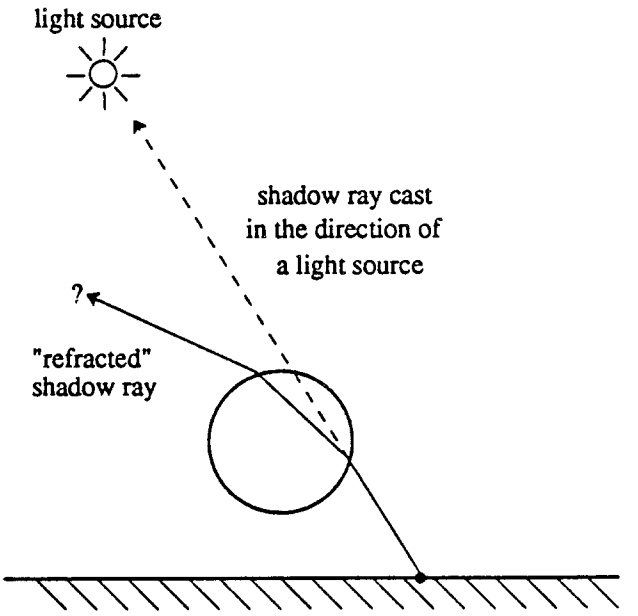


Figure 7.2 – Shadow ray passing through refractive volume

We should mention one slightly tricky aspect of the shadowing calculation. If a surface point lies on an edge between two polygons, the surface point may be contained by both polygons. If we naively test for intersections with the shadow ray originating on this edge, we may detect an intersection with one of the polygons containing the edge, and we will falsely conclude that the surface point is in shadow. This spurious intersection will be very close to the origin of the shadow ray, however, so we simply ignore intersections within a small error tolerance of the shadow ray origin.

7.7. Light Source Model

Before examining the diffuse and specular reflection calculations in greater detail, let us examine the effects of different light source models on the shadow and reflection calculations.

Directional light sources are currently modeled as infinitely-distant point light sources, as in older UNIGRAFIX renderers. This is a reasonable approximation to an environment in outer space illuminated by a light source such as the sun. In such an environment, all light rays originate from a light source occupying a small solid angle. Since all light rays travel in the same direction, various calculations can be precomputed. The constant direction vector $\hat{l}$ pointing toward each light source can be pre-normalized, for example. If no shadows are desired, the diffuse reflection from each polygon can also be precomputed, instead of recomputing this quantity for every pixel. Furthermore, the attenuation of such light due to the inverse square law is negligible, since a light ray has travelled large distances compared to the extent of our scene by the time it reaches a surface.

Unfortunately, this light source model is not realistic in many situations, and is particularly ill-suited for interior lighting. The edges of shadows are sharp (no penumbrae), and it is difficult to reduce the contrast between shadowed and fully illuminated parts of the scene without excessive use of ambient light. The ambient light approximation we have described reduces orientation cueing available from directional shading if used too zealously.

A reasonable extension to the current renderer would be the addition of local point light sources. Local light sources require a little more computation, but the additional overhead does not seem unreasonable compared to the computational expense of ray intersection calculations. Specifically, the light direction vector must be computed and normalized each time a new surface point is found. The light intensity must also be attenuated with the square of the distance from the light source. The diffuse reflection from each polygon can no longer be precomputed, but this optimization is rarely applicable in the context of ray-tracing anyway.

The most expensive but realistic light source model is *area light sources* at finite distances. If we allow a polygon to emit light, for example, we can model fuzzy shadows with penumbrae [Coo84]. From a given surface point, a number of shadow rays are cast toward a light-emitting polygon. These rays are randomly distributed so that they accurately sample the area of the light source. In the umbra (dark part) of a shadow, all the shadow rays will be blocked. In the penumbra, the light source will only be partially eclipsed at the surface point, and only a percentage of the rays will be blocked. Unfortunately, many shadow rays must be cast to adequately sample a large light source. If too few rays are generated, the resulting image will contain unacceptable noise.

7.8. Diffuse Reflection

Once the light intensity $\mathbf{i}_l$ incident on a surface point from a particular light source has been found, we must calculate the intensity of the light that is reflected in the viewing direction. Because we assume that diffusely reflected light is reflected equally in all directions, the intensity of light diffusely reflected in the viewing direction is not dependent on the direction of view. The reflected intensity is proportional to the projected area of the polygon as seen from the direction of the light source. This area is proportional to the cosine of the angle between the surface normal vector and a vector in the direction of the light source, or equivalently, the dot product of these vectors if they are normalized. Thus, the diffusely reflected light is the product of the incident light (after possible attenuation by shadowing transparent material), the coefficient of diffuse reflection, the color of the surface (the fraction of red, green, and blue light that the surface reflects), and the aforementioned dot product: $\mathbf{i}_l \, k_d \, \mathbf{c} \, (\hat{n} \cdot \hat{l})$.

UgRay contains an optimization that is rarely used in full ray-tracing, but increases rendering speed for simple images containing surfaces without shadows or specular highlights. In this case, the diffuse reflection from a polygon can be calculated just once. Since diffusely reflected light will not vary across the surface of the polygon (for infinitely-distant point light sources), and does not depend on the viewing direction, the reflected intensity is stored for simple lookup in case future rays hit the same polygon.

7.9. Specular Reflection of Light Rays

If a shiny surface is modeled as a perfect reflecter, incident light will be specularly reflected in only one direction (such that the angle between the reflected ray and the surface normal equals the angle between the incident ray and the surface normal, and all three of these vectors lie in the same plane). Therefore, we will see a specular highlight on a surface only in the rare case when our viewing direction corresponds with the direction of the reflected light beam. This presents two problems: the resulting image will lack realism because we rarely encounter such perfect reflection even in highly polished surfaces, and the high spatial frequency of such a highlight occurring on a curved surface presents an aliasing problem.

A shiny surface can be modeled more realistically if we look at the interaction between incident light and the surface at a microscopic level. At this level, the surface can be thought of as composed of tiny microfacets, each of which is a perfect reflecter. Although each microfacet is oriented randomly, the overall distribution of microfacet orientations is determined by the roughness of the surface. For a smooth surface, the vector normal to each microfacet will not vary much from the normal of the macroscopic surface. A rough surface will have a wider distribution of microfacet normals, and it will therefore spread the intensity of specularly reflected light over a wider range of directions around the direction of perfect reflection.

The most realistic models of specular reflection account for the distribution of microfacet orientations as well as shadowing and occlusion of microfacets when the surface is viewed or illuminated at grazing angles [Bli77, CoT81]. *UgRay* incorporates an older and simpler model introduced by Phong [Pho75]. This method calculates the intensity of light that is specularly reflected in the viewing direction by raising to some constant power the cosine of the angle between the viewing direction and the direction of ideal reflection. As the viewing direction

deviates from the direction of ideal reflection, the intensity of specularly reflected light falls off according to this "Phong term." Typical values of the exponent range from 10 to 100. Higher values produce a sharper highlight with more rapid fall-off, corresponding to shinier surfaces.

The Phong model can be improved by noticing that the total reflected energy and its distribution varies with the angle of the incident light. Glass, for example, reflects much more light at grazing angles than at normal incidence. In *UgRay*, the variation in reflected energy is approximated by varying the specular coefficient k_s as a function of the angle of incident light (the calculation of the reflection coefficients k_d, k_s, and k_t will be discussed shortly). The distribution of reflected energy should be modeled by varying the Phong exponent with the incident angle, but this feature is not currently supported in the renderer. Before such a feature is added, more realistic reflection models should be investigated [Bli77, CoT81]. Although these models require more computation, they can be justified by underlying physical principles. The Phong model is based primarily on empirical observations and does not produce realistic results at grazing angles.

Before 1981, it was commonly believed that the color of specularly reflected light was the color of the light source. In other words, specularly reflected light did not interact with the reflecting surface in any way that would change its spectral composition. Cook and Torrance showed that this conventional wisdom was only true for some materials (such as plastics), but other materials (particularly metals) affect the wavelengths of specularly reflected light. The model that they proposed accurately models these effects for many materials, including materials with nonhomogenous reflectance properties.

Unfortunately, the Cook-Torrance model requires more computation and many parameters to accurately model the reflectance of a particular material. A library of parameters for various materials is needed by all but the most experienced user.

For these reasons, we chose to stick with the simpler Phong model in the initial implementation of *UgRay*. We wished to incorporate the important result regarding the color of specularly reflected light, however, so we introduced a *light interaction* coefficient m (this parameter has been referred to as *metalness* in other renderers we are aware of). The light interaction coefficient specifies the degree to which specularly reflected light interacts with the surface. A value of 0 means that no interaction takes place, and the color of the specular highlight is not affected by the color of the surface. A value of 1 means that the color of the specular highlight is the product of the incident light and the red, green, and blue reflectances of the surface color. Fractional values within this range cause the reflected color to be linearly interpolated between these extremes.

The final intensity of specularly reflected light is therefore calculated as a product of the intensity of the incident light, the specular reflection coefficient (which can vary as a function of the angle of the incident light), the cosine of the angle between the viewing direction and the angle of ideal reflection (or equivalently, the dot product of these two normalized vectors) raised to a power that determines the rate of intensity fall-off, and a linear interpolation between the color of incident light and the color of the surface (depending on the value of the light interaction coefficient): $\mathrm{i}_l\, k_s (\hat{n} \cdot \hat{h})^p \; (m\,\mathrm{c} + (1-m))$.

7.10. Specular Reflection of Viewing Rays

We have just shown how light is specularly reflected from a light source on a shiny surface. To model reflections of other surfaces in the current surface, we must trace a ray in the direction of ideal reflection. If this new viewing ray intersects another surface, we will see the new surface being reflected in the initial surface. This process is repeated recursively to model multiple reflections on several shiny surfaces in a scene.

The third term in equation (7.1), $\mathbf{c}_s k_s (m\,\mathbf{c} + (1-m))$, models this reflection. The color returned by the reflected ray is $\mathbf{c}_s$ in this term. The light interaction coefficient is used here, as above, to determine the actual color of the reflected light. The specular reflection coefficient k_s determines the fraction of light reflected for a given angle of incident light.

Because only one ray is traced in the direction of ideal reflection, all reflections are perfectly sharp. The problem encountered here is identical to the problem of perfect specular reflection of light sources. In the case of light sources, specular reflection from a roughened surface is relatively inexpensive to compute using the Phong approximation. Unfortunately, the corresponding approximation to produce blurred reflections is not trivial: multiple rays must be cast around the direction of perfect reflection [Coo84]. Ideally, these rays should be randomly distributed according to the distribution of microfacet orientations specified by the surface roughness. The colors returned by all these rays would then be averaged to produce the color of the final reflection. Images produced using such techniques are very realistic, but expensive to compute.

Together, these differing models of specular reflection (Phong approximation for non-ideal reflection of light sources, and perfect reflection of viewing rays) give us contradictory models of a shiny surface. We have found this contradiction acceptable owing to the added realism of specular highlights available from the Phong model, and the reasonable computational cost of ideally reflected viewing rays. Images produced using these reflection models look surprisingly good, even though blurred reflections would be desirable in situations where the computational expense could be justified.

7.11. Transmission of Viewing Rays

The final term in equation (7.1), $\mathbf{c}_t k_t \mathbf{c}$, determines the color of light transmitted through a transparent surface. The transmission coefficient k_t determines the transmitted fraction. Similarly to the specular reflection coefficient, k_t is allowed to vary as a function of the incident angle, and in some cases, the index of refraction of the transparent material. The color $\mathbf{c}$ of the surface multiplies the color $\mathbf{c}_t$ returned by a transmitted ray to filter the color of the transmitted light (*i.e.*, a red filter will not transmit green light). The color returned by a transmitted ray has already been properly attenuated if it passes through a material that is not perfectly transparent. The surface color $\mathbf{c}$ and the attenuation characteristics of a transparent volume must be chosen with care so that reasonable filtering is performed.

The direction of the transmitted ray is the same as the incident viewing ray unless the transmitted ray enters a material with a differing index of refraction. Because only one ray is transmitted, objects viewed through a transparent material will be perfectly sharp. Ideally, we

would like to cast several rays that would sample a distribution of transmitted rays [Coo84]. This would allow more realistic simulation of translucent materials.

7.12. Attenuation by Fog

The intensity of a ray that passes through fog or a translucent material is attenuated exponentially. The exponential term in equation (7.1) attenuates the color returned by a ray as a function of the distance d the light travels and the translucency **f** of the medium it passes through.

UgRay currently contains differing models for attenuation by fog versus attenuation by a translucent material. These differing models should probably be reconciled in a future version of the renderer.

For fog, the user specifies a distance D in world units through which a ray can travel before it is completely attenuated. "Complete attenuation" occurs when the intensity contribution of the ray is less than 1 part in 256 (the output precision of red, green, and blue components). If the distance a particular ray travels is d, the properly scaled exponential attenuating function is

$$e^{\left(\frac{d \ln(\frac{1}{256})}{D}\right)}$$

The attenuation factor **f** in equation (7.1) for fog is therefore $\frac{\ln(\frac{1}{256})}{D}$. Note that this factor does *not* vary for the red, green, and blue components of a color. It may be more realistic to use different attenuation factors for each component, so that light of differing wavelengths could penetrate the fog to varying degrees. In the current implementation of *UgRay*, this is more than we wanted to specify.

Since fog *scatters* light to some degree instead of simply attenuating it, the fractional intensity that is removed by the attenuation calculation is replaced by the background color (which is assumed to be the color of the fog) using linear interpolation. A primary viewing ray originating from the eye is not attenuated until it enters the spatial subdivision so that objects in the foreground of the image will be displayed with little attenuation.

A more realistic model of fog would allow variable density of fog (depending on elevation, perhaps). The varying density must be integrated along the distance the ray travels to calculate the correct attenuation.

Another enhancement that increases the realism of atmospheric effects is to model the scattering of light rays more accurately. In areas where fog is illuminated, a certain percentage of the light is scattered towards the eye, making the fog appear to glow. In [Max86], beams of light passing through fog were modeled by adding invisible polygons to a scene that enclose volumes that are in shadow. The final intensity returned by a ray is calculated by adding light intensity when the ray passes through illuminated fog, and attenuating intensity when the ray penetrates a volume of fog lying in shadow. The additional complexity required to introduce shadow volumes makes this an expensive method to accomplish using ray-tracing. Max used a special-purpose scan-line algorithm to process his scenes efficiently.

7.13. Attenuation in a Translucent Material

UgRay calculates attenuation of light passing through a translucent material a little differently than attenuation for fog. The user specifies the color that would result if white light passed through one unit of the given material. This color is composed of red, green, and blue components that are used to calculate separate attenuation functions for each component. If $\mathbf{c_a}$ is the intensity of a component after this prototypical one-unit attenuation, the attenuation function is simply

$$\mathbf{c_a}^d = e^{(d \ln(\mathbf{c_a}))}$$

and the attenuation factor f in equation (7.1) is $\ln(\mathbf{c_a})$. Unlike fog, three separate attenuation functions are generated during this calculation, so light of varying wavelengths may penetrate a translucent material differently. Also, light propagated through a translucent material is simply attenuated. No attempt is made to model scattering as we tried to approximate for fog.

It is important to factor the amount of attenuation into the contribution made by a ray. We may be able to avoid casting further reflection and transmission rays if the current ray is significantly attenuated by heavy fog, for example. Therefore, the amount of attenuation to be applied to the current ray is calculated before reflection and transmission rays are cast. However, the attenuation is actually applied to the ray only after the reflected and transmitted components have been added in, if necessary. In the case where three different attenuations are to be applied to the red, green, and blue components, we select the largest value (the *least* attenuation) as the estimate by which to modify the contribution of reflected and transmitted rays.

7.14. Calculation of Reflection and Transmission Coefficients

As we have mentioned in previous sections, the coefficients of specular reflection and transmission k_s and k_t can vary according to the angle that light is incident upon a surface. *UgRay* allows the specification of these coefficients in three ways. They may be defined as constants, quadratic functions of the incident angle, or functions of the incident angle and index of refraction.

The second method allows the user to specify two values for a coefficient corresponding to normal incidence and grazing angles of light. Typically, the first value is greater than the second for the specular reflection coefficient (more light is reflected at grazing angles), and vice versa for the transmission coefficient (more light is transmitted at normal incidence). For angles between these extremes, a value is calculated by passing a parabola through the two extreme points (figure 7.3).

There is little physical justification for this scheme. A material such as gold has a reflectance function that does not monotonically increase between the extreme incidence angles. But considering that we are already approximating the specular reflectance function crudely with the Phong approximation, this scheme performs satisfactorily.

If the user does not specify k_s and k_t for surfaces bounding a transparent solid, *UgRay* calculates the values as functions of the incident angle and the relative index of refraction. These calculations are based on equations from basic optics, and produce realistic-looking glass, for

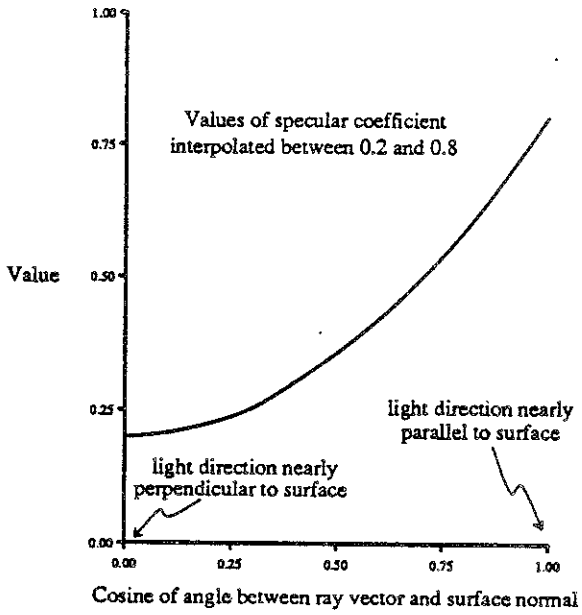


Figure 7.3 – Parabolic interpolation of coefficient values

example:

$$k_s = (1 - k_d)\left[\frac{n_1 \cos\theta_1 - n_2 \cos\theta_2}{n_1 \cos\theta_1 + n_2 \cos\theta_2}\right]^2$$

$$k_t = 1 - k_s - k_d$$

where n_1 is the index of refraction of the medium through which the initial ray travels, n_2 is the index of refraction of the medium that the transmitted ray enters, θ_1 is the angle between the incident ray vector and the surface normal vector, and θ_2 is the angle between the transmitted ray and the negative surface normal (figure 7.4). These values are most conveniently computed at the same time that the direction of a refracted ray is calculated.

7.15. Smooth Shading

The fact that the current implementation of *UgRay* accepts only polygons places restrictions on the types of surfaces that can be realistically rendered with the program. It is difficult to render a smooth sphere, for example, unless it is tessellated into a many polygons. The correct solution to this problem would be to introduce other primitives, such as quadrics (spheres, ellipsoids, cones, cylinders, paraboloids, etc.), superquadrics, and bicubic patches. Ray intersection methods for such primitives have been well-studied, although fast techniques for insertion into the spatial subdivision might require further investigation. The intersection calculation for a higher-order surface is usually more expensive than the intersection calculation for a polygon. But it can be faster to compute one higher-order intersection than to perform many intersection calculations with the numerous polygons necessary to approximate the surface.

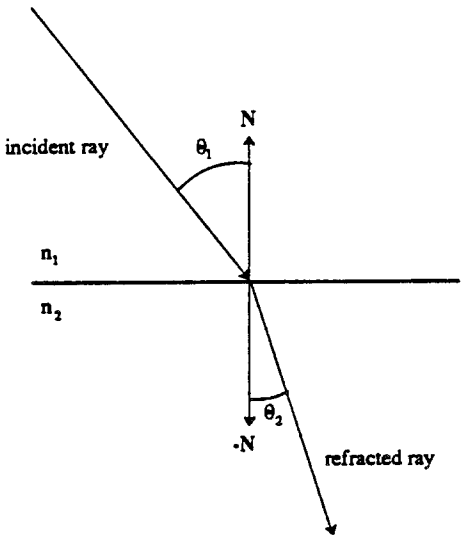


Figure 7.4 – Refraction of a ray

In order to simulate smooth surfaces without implementing additional primitives, we investigated two strategies for smoothly shading polygons. By varying the shade across the surface of a polygon, the human eye can be tricked into believing that it is observing a smoothly curved surface. The underlying polygonal representation is still apparent along the silhouette of an object that is treated in this manner, but the approximation is usually worthwhile if smooth surfaces cannot be generated in any other manner.

The first method is known as *Gouraud shading* [Gou71]. In this method, a *vertex normal* is calculated at each vertex by averaging the surface normal vectors of all polygons that include the vertex. The shade of a surface having this vertex normal is calculated. To determine the shade of a given point within a polygon, the shade along an edge of the polygon is calculated by linearly interpolating between the shades of the vertices defining the edge. The shade is then linearly interpolated between points on two edges to determine the shade of a point on the interior of the polygon. As pointed out in [Hec86], this bilinear interpolation is not accurate for a polygon viewed in perspective, but the error is small enough that it can be ignored in most cases.

The uses of Gouraud shading are limited in the context of ray-tracing. It makes no sense to interpolate shades if one vertex is in shadow and another is fully illuminated, for example. It is also silly to interpolate reflections across the surface of a polygon. Therefore, Gouraud shading is only practical for opaque, non-reflective surfaces viewed without shadows.

A more realistic technique was introduced by Phong [Pho75]. Vertex normals are computed as above, but the surface normal vector itself is interpolated across the surface instead of the shading value. When the normal vector has been calculated for a point inside the polygon, full shading calculations can be performed which may include reflection, transmission, and shadows.

Accurate interpolation of the surface normal can be an expensive proposition. Simple interpolation between the x, y, and z components of two normal vectors will result in a non-uniform

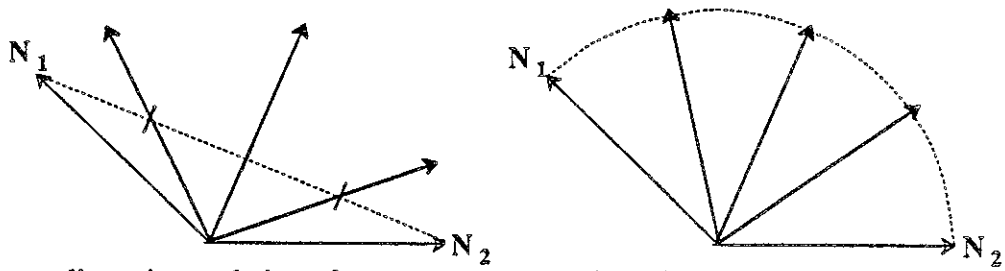


Figure 7.5 – Interpolation of normal vectors

distribution of interpolated normal directions, especially when the angle between the original normals is large (figure 7.5). A better method is to interpolate the *angle* between the original normal vectors. This requires an inverse cosine function call to calculate the angle. A cross-product is necessary to find a third vector perpendicular to both normals. The third vector is used as a rotational axis to rotate one of the normals by the appropriate fraction of the angle between the normals. This rotation requires one sine and one cosine function call, and approximately 30 floating-point operations.

The high cost of these calculations can be amortized in a scan-line renderer using incremental methods [BiW86]. Unfortunately, the same optimizations are more difficult to apply in a ray-tracing renderer. When a series of reflected rays hits a polygon, for example, they may be spaced non-uniformly along an arbitrary direction. Depending on the anti-aliasing method used, even sequential primary rays may not lie in the same scan line.

The computational expense of this method is not the main problem we encountered while using it, however. Many objectionable artifacts caused by the underlying polygonal representation are visible. To illustrate these problems, consider the case of a sphere approximated using a uniform tessellation of triangles. If the sphere is illuminated by an overhead light source, the top half of the sphere will be illuminated, and the bottom half will receive no illumination. Therefore, the upper part of a triangle spanning the equator should receive illumination, and the lower part should receive none. But because the triangle is a flat approximation to the curved surface, the whole triangle may fall in the grazing shadow of the polygon above it (figure 7.6). The result is a ragged triangular shadow around the equator of the "smooth" sphere.

Triangles near the silhouette of the sphere cause similar problems when *UgRay* attempts to determine the direction of reflected or transmitted rays (figure 7.7). The interpolated normal causes the reflected ray to penetrate the reflecting surface! The reflected ray will then intersect the polygon on the back side of the object, producing erroneous results. The problem occurs because we are interpolating a surface normal vector, but continue to use the surface intersection point P instead of the point Q that corresponds to the actual point of intersection with the surface we are attempting to model. Although various techniques were investigated to generate the correct intersection point on the smooth surface, a general solution to this problem is difficult and rather arbitrary. Modeling with bicubic patches or other higher-order primitives seems to be a

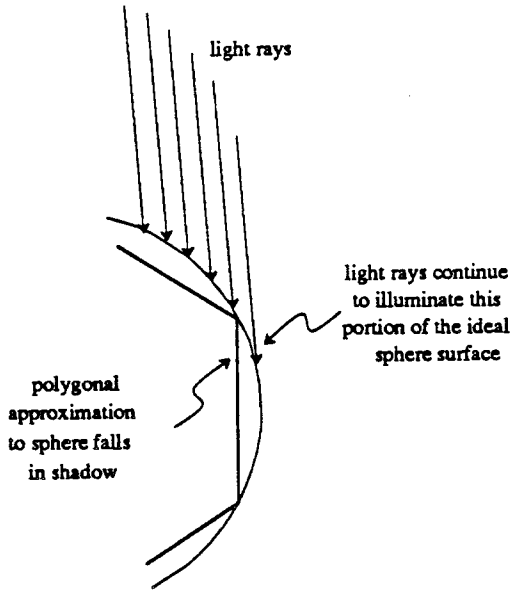


Figure 7.6 – Shadow defect caused by smooth shading

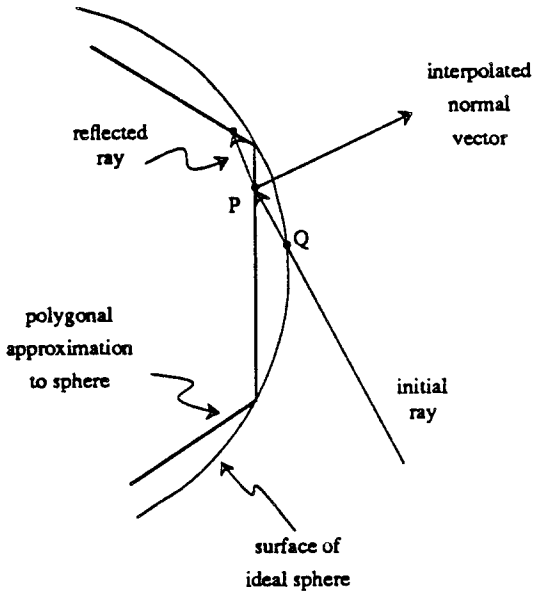


Figure 7.7 – Reflection defect caused by normal interpolation

much more appropriate method to render smooth surfaces.

7.16. Optimizations

The computation of various color intensities can be performed using integer arithmetic, since only 8 bits of precision are required for each of the red, green, and blue intensities during final output. *UgRay* uses 15 bits to record each component of the color of a light source, and the

intensity of each component reflected by a surface. The color reflected by a surface can be calculated by multiplying the corresponding components of the incident light intensity and the color of the surface, followed by an arithmetic right shift of 15 bits.

The improvement achieved by this optimization depends on the relative speed of integer multiplication and shifting versus floating-point multiplication on a particular computer. For the 68010 microprocessor on which *UgRay* was initially developed, the integer calculation can be performed three times faster than the floating-point version. On a Vax 750 with floating-point accelerator, the integer calculation is actually 25 percent slower. On a Vax 8600, the penalty is less than 4 percent. The source code for *UgRay* does not currently contain switches that allow this calculation to be optimized for a particular machine. The shading calculation typically requires less than 10 percent of the total execution time, however, so these optimizations do not have a major impact.

A more promising optimization that is not presently implemented concerns shadow testing. For a complicated scene with several light sources, the renderer can spend as much as 30 percent of the execution time testing shadow rays for intersection. When neighboring pixels in the final image fall within a shadow, they are often shadowed by the same polygon. The following optimization would reduce the time required to determine if a particular point is in shadow. Before a shadow ray is cast, a neighboring pixel is consulted to see which polygon is shadowing it. A quick test against this polygon will determine if the new surface point is also in shadow. If it is, we can avoid casting a shadow ray and thereby avoid computations that may be a hundred times more expensive than the test against a single polygon.

Ideally, a whole scan line of shadowing polygon information should be maintained so that this information can be propagated from scan line to scan line as well as from pixel to pixel within a scan line. It might also be beneficial to maintain separate shadowing information for each possible ray depth. This would allow several rays that are reflected from a mirror to share shadowing information about the surfaces they eventually hit.

7.17. Future Enhancements

As we have shown in this section, the realism of images produced by *UgRay* is primarily determined by the implementation of the shader. Many researchers have found that the shader can be modified to produce more surface detail than that which is explicitly available from the geometry of a particular scene [Hec86]. For example, *texture mapping* can be employed to vary the color displayed on a particular surface. The applications of texture mapping are almost endless: clouds, oceans, and continents can be mapped onto a sphere to create a planet; text or repetitious patterns can be mapped onto "man-made" objects in a scene; a bark texture can be mapped onto a cylinder to form the trunk of a tree; wood grain can be mapped onto furniture; and dirt or scratches can be mapped onto otherwise featureless surfaces to enhance realism and depth cueing.

A related enhancement is *bump mapping*. The surface normal of a particular surface point is calculated using a function or table lookup. By perturbing the normal from the actual normal, the shade of the surface will vary as though the surface was bumpy. Bump-mapped objects look realistic, but a close-up view of a silhouette edge will remain smooth.

The incorporation of texture mapping, bump mapping, and related techniques would dramatically increase the realism of images produced using *UgRay*, because it is not feasible to model such fine detail with additional polygons. Texture colors can vary widely within a single pixel, and this can affect the design of the anti-aliasing routine, which we will discuss in the next section.

A final issue that should be considered in a discussion about the modeling of reality is the manner in which the color of reflected light is calculated. Up to this point, we have glossed over the issue when we have said that the color of incident light is specified by its red, green, and blue intensities, and a surface color is defined by the intensities of red, green, and blue that the surface reflects. To calculate the color reflected from a surface, *UgRay* simply multiplies the corresponding components of the incident light and the surface color.

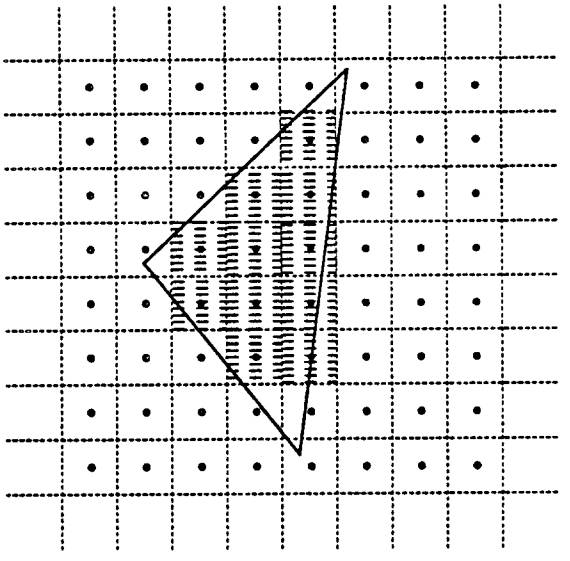
Unfortunately, things are not so simple in reality [CoT81, Ups85]. Physical light is typically composed of a continuous spectrum of wavelengths. Similarly, a surface should be described by a continuous reflectance function. The reflected light is the integral over the product of these two functions. Red, green, and blue intensities could then be calculated that match or approximate the color sensation that the resulting spectrum would produce in a human observer. This requires knowledge of the wavelength spectra and intensity curves of the phosphors in a particular monitor, and the tristimulus response curves of the cones in the human eye. Since these calculations are complicated, it is likely that most renderers will continue to use the approximations that we mentioned previously. It is important, however, to understand that these are only approximations.

8. Anti-Aliasing

The problem of *aliasing* is manifested in several different ways in computer graphics images: edges have a jagged or "staircase" appearance; small animated objects blink or strobe as they move across the screen; and a regular pattern such as a checkerboard exhibits *Moiré* patterns when viewed in perspective. Although these examples may seem unrelated, each is an artifact caused by an attempt to render a continuous phenomenon using discrete samples distributed regularly in space or time. In figure 8.1, a polygon is rendered by casting a ray through the center of each pixel. If a pixel is simply colored with the color of the polygon hit by the corresponding ray, a jagged and badly-shaped figure is produced.

On a device capable of displaying only a few colors, there is little one can do to remedy jagged edges. But if many colors are available, each pixel can be colored with a weighted mixture of colors occuring within a certain neighborhood of the pixel. Ideally, the area of each color that occurs within some distance of the center of the pixel should be multiplied by a weighting factor based on a *sinc* function. This type of filtering is expensive to compute, however, and is rarely implemented in practice. Usually, a less expensive filter such as a triangle or cone is used to weight discrete color samples.

In a scan-line renderer, the colored areas occuring within the neighborhood of a pixel can be computed without too much difficulty (although various techniques are used to approximate the area contributions and limit the computational expense), because a scan-line renderer continuously tracks edges and transitions between polygons. A ray-tracing renderer must resort to other methods for estimating the area covered by different colors, because little scene information is available besides the color information that is returned by individual ray samples. This makes it easy to implement simple anti-aliasing routines for a ray-tracer. On the other hand, the lack of



• = ray sample point within each pixel

Figure 8.1 – Aliasing effects caused by sampling one ray per pixel

geometric information makes it difficult to do high-quality anti-aliasing without generating a large number of rays.

One standard approach is known as *super-sampling*. Instead of casting one ray through the center of each pixel, n^2 rays are cast through an n by n grid within each pixel. A weighted average of the colors returned by these rays as well as rays cast in neighboring pixels is computed to approximate the area covered by each color in the neighborhood of the pixel. For most applications $n = 4$ seems to produce reasonable results (*i.e.*, 16 rays per pixel).

Although super-sampling does a fair job of reducing aliasing in most scenes, it is expensive. If 16 rays are cast for each pixel, the computational expense of the ray-tracing algorithm is multiplied by a factor of 16. Furthermore, much of the effort is wasted: several pixels that are completely enclosed by a uniformly-shaded polygon require no anti-aliasing.

Adaptive super-sampling attempts to expend the effort of anti-aliasing only where it is most needed. A small number of rays (four) are cast within a particular pixel, and the colors that are returned are compared. If little variation is noted between these initial samples, it is assumed that the pixel covers a uniform region in the scene. The colors of the samples are averaged, and the pixel is assigned the result. If the variation between the samples is greater than a predefined threshold, the pixel is subdivided further and more rays are fired. Compared to one ray per pixel, this method increases the computational expense by a factor of at least four.

8.1. An Inexpensive Anti-Aliasing Routine

The expense of the aforementioned methods encouraged us to develop a new anti-aliasing routine better suited to our requirements. We were primarily concerned with speed and the capability to produce static images without very small polygons or textures. Animation and textures will quickly expose inexpensive anti-aliasing methods, so we propose to address these issues with higher-quality anti-aliasing routines in the future. In the meantime, a less expensive strategy will allow the rendering of less demanding scenes in reasonable time. Even when a higher-quality anti-aliasing routine has been developed, the user should always be given a choice of anti-aliasing routines to produce an image of the quality he requires without expending unnecessary resources on the problem.

The key to the development of our inexpensive anti-aliasing routine was the realization that variations within a particular pixel often occur at the edges and corners of the pixel as well as in the interior. For example, a particular polygon edge will usually pass through the edges of a pixel as well as the pixel's interior. Once again, this is not necessarily the case, especially in scenes with tiny polygons or detailed texture patterns. But if we accept these assumptions, we can redistribute rays that would normally sample the interior of a pixel to the corners of the pixel, where the values they return can be shared with neighboring pixels. This increases speed by a factor of four.

This basic scheme is refined using adaptive sampling. If the samples returned at two adjacent corners of a pixel differ by more than 10 percent (an arbitrary choice) in the red, green, or blue components, a new ray is fired in the middle of the pixel's edge to provide a better estimate

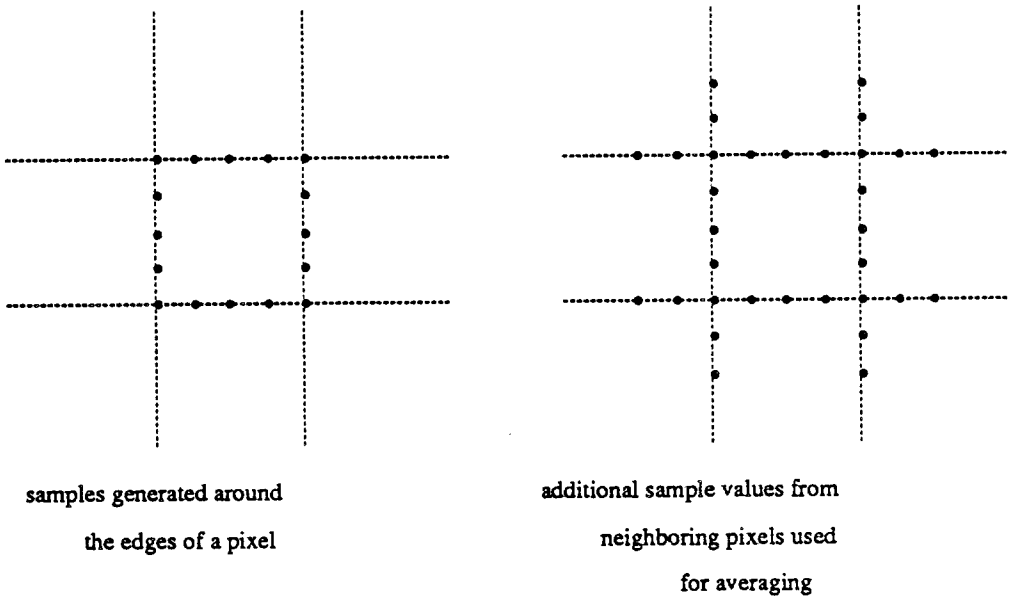


Figure 8.2 – Pattern of samples used for inexpensive anti-aliasing

of the area covered by each color. This sample is compared to both of the corner samples. If the variation is still too large, more rays are generated by further subdividing the distance along the pixel's edge. Whenever the variation between two samples falls within an acceptable range, values for all samples lying between the original samples are linearly interpolated instead of sampled by firing rays into the scene. When this process is complete, 16 values are available around the edges of a particular pixel (figure 8.2). Notice that none of these samples are taken from the interior of the pixel. This fact is responsible for the speed of this method, because all sample values can be shared by neighboring pixels. On the other hand, avoidance of interior samples causes most of the aliasing defects visible in images produced using this anti-aliasing strategy.

The final color of the pixel is found by computing a weighted average of the 16 samples around the edges of a pixel, and 16 additional samples borrowed from the pixel's neighbors. These samples are shown in figure 8.2. The samples are weighted according to their distance from the center of the pixel, such that the sum of all sample weights equals one. This is a rough approximation to the cone filter described in [Bun82]. If the weights are multiplied by a normalization factor, all computations can be performed using integer multiplications and additions followed by an arithmetic shift.

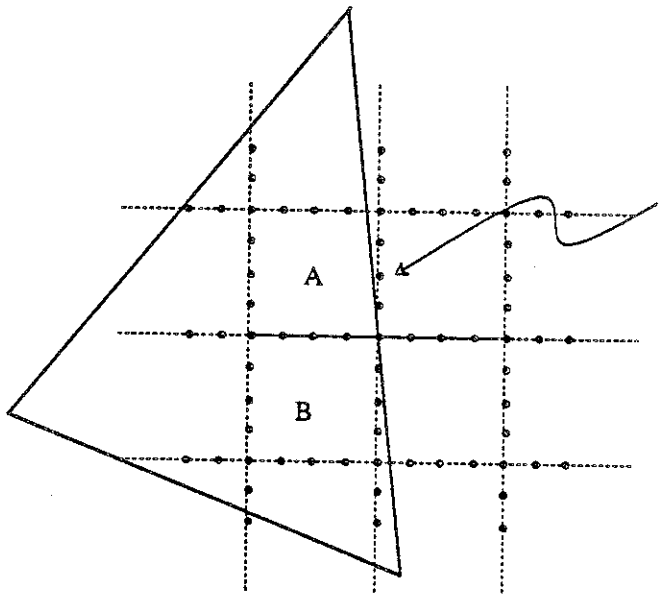
Unfortunately, this anti-aliasing function has a non-linear response that causes problems for nearly-horizontal or nearly-vertical edges. When such an edge crosses several pixels, it will suddenly be weighted much more heavily as it crosses an area where the method computes many sample values (figure 8.3). This results in a "twisted-rope" appearance that cannot be avoided without a more expensive anti-aliasing technique. However, we have not found the aliasing problems too objectionable in the static images we have rendered. Furthermore, the method casts only 30 percent more rays than are required for one ray per pixel (depending on the scene and screen resolution), with a dramatic improvement in image quality. For high-resolution images

(1024 by 768 pixels) the overhead is less than 10 percent for typical scenes.

8.2. Future Anti-Aliasing Algorithms

A more sophisticated anti-aliasing routine will be required to produce high-quality animated sequences and texture mapped surfaces. A simple technique that we believe would produce acceptable results is a weighted average of 5 samples in the interior of each pixel arranged in a pattern similar to the 5-spot on dice. This method lacks the flexibility of adaptive sampling, but further samples may not be necessary for most scenes. This method is, of course, at least 4 times more expensive than our current technique.

Wold and Dippé [DiW85] and Cook [Coo86] have suggested the use of *jittered* sampling. Before a sample is taken, the regularly-spaced location of the sample is randomly perturbed by a small amount. The result is that aliasing errors are traded for noise, which the researchers claim is less objectionable than the false patterns that can arise through aliasing. We have not included a stochastic sampling algorithm in the initial implementation of *UgRay* because the technique is expensive compared to our current techniques. Also, it is our opinion that a very high level of noise caused by low sampling rates is sometimes more objectionable than aliasing artifacts, especially in certain types of scenes where aliasing is not an overwhelming problem to begin with. This situation will change as smaller polygons and textures begin to strain the capabilities of our current techniques.



Although the polygon occupies nearly the same area in pixels A and B, pixel B gets a much larger contribution, because the polygon misses many of the samples along the edge of pixel A. This problem causes aliasing of nearly-horizontal or nearly-vertical edges.

Figure 8.3 – Twisted rope defect caused by inexpensive anti-aliasing

9. Performance Evaluation

At present, no carefully controlled performance comparison of ray-tracing renderers based on various optimization techniques has been done. This is not an easy task, unfortunately, since there does not yet exist a test bed of scenes that can be theoretically guaranteed to represent different types of scene distributions. For example, a scene may be uniformly distributed through space, or it may be concentrated in a few clumps. Single primitives might be local in extent, or span a significant portion of the entire scene. A careful categorization of different scene distributions and characteristics will require some thought.

We expect that such a comparison will show that selection of the most efficient ray-tracing optimization depends on the distribution and characteristics of scene primitives. Therefore, we do not foresee that a single ray-tracing algorithm will satisfy the entire computer graphics community. It would be helpful to know exactly how different optimization strategies perform on different scenes, so that a good algorithm can be selected for rendering a certain class of scenes.

Our experience with the uniform subdivision employed by *UgRay* suggests that uniform subdivision techniques will be difficult to beat in cases where scene primitives are distributed fairly uniformly throughout a scene. As an example of this type of scene, consider the granny-knot lattice shown in figure C.1 (appendix C). Although polygons are somewhat concentrated within each "knot," the overall distribution of primitives is uniform compared to the scene in figure C.2. Polygons in this scene are distributed densely in the planet (which is approximated by 800 smoothly-shaded triangles) and the letters (600 polygons). Because many polygons fall within just a few cells of the uniform subdivision, and more ray intersection tests are therefore required in these regions, the second scene is more costly for *UgRay* to render. By restricting the extent of the tiled plane, a more even distribution of polygons within cells could be achieved, and execution speed would be increased.

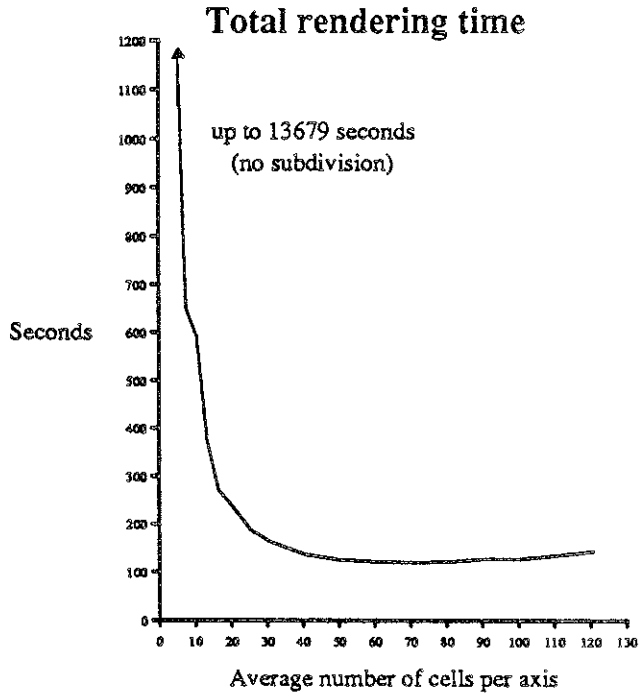
9.1. Effects of Finer Subdivision

The performance of a ray-tracing renderer based on spatial subdivision is obviously dependent on the number of cells allocated in the subdivision. If few cells are used, this technique degenerates into simplistic ray-tracing where each ray must be tested against nearly every primitive. If too many cells are used, the renderer will spend most of its time propagating rays through numerous nearly-vacant cells.

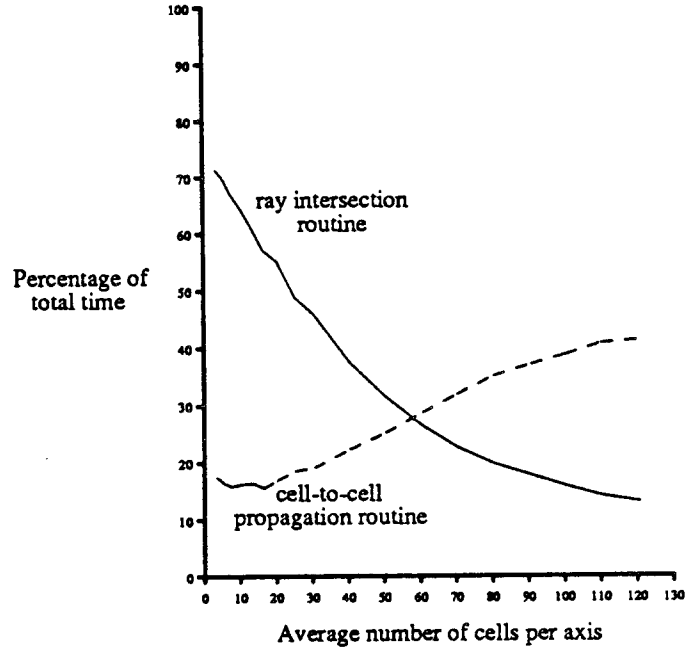
In [Ull83], a theoretical discussion regarding the effects of finer subdivision is presented. Ullner argues that, for most scenes, the number of ray intersection calculations decreases quadratically as the number of cells in a uniform subdivision increases. Obviously, the time required to propagate a ray through the cells increases linearly. Therefore, the algorithm should perform optimally at the point where the sum of these curves is minimized.

Empirical measurements of *UgRay* have confirmed these theoretical results, as have measurements made by other researchers [FTI86]. The graphs in this section show different measurements that are affected by increasing the number of cells in the spatial subdivision. The scene

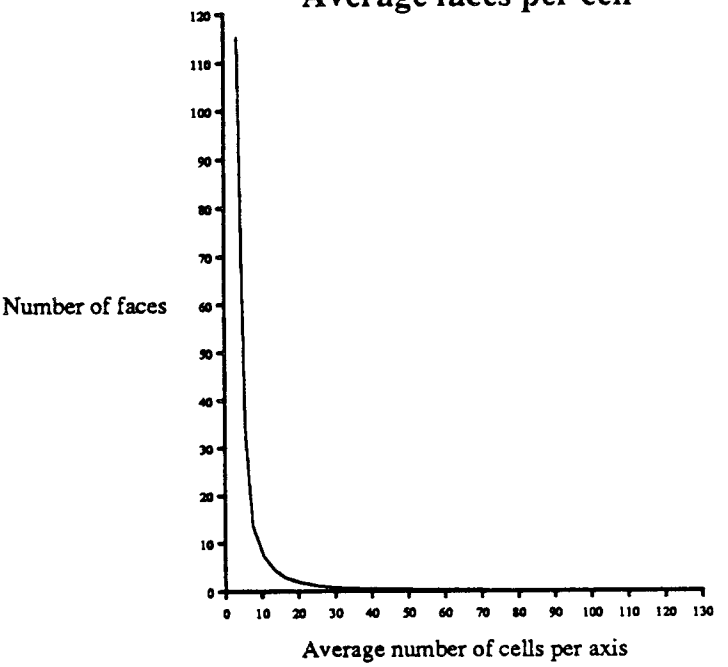
used to gather these measurements is the granny-knot lattice in figure C.1. This scene contains 3864 polygons. In the following graphs, the image was rendered at a resolution of 256 by 256 pixels on a Vax 8650 (about 10 times faster than a Vax 750 with floating-point accelerator) with our inexpensive anti-aliasing technique. No shadows, reflections, or fog effects were computed, because these usually increase rendering time without changing the spatial sudivision performance curves significantly.

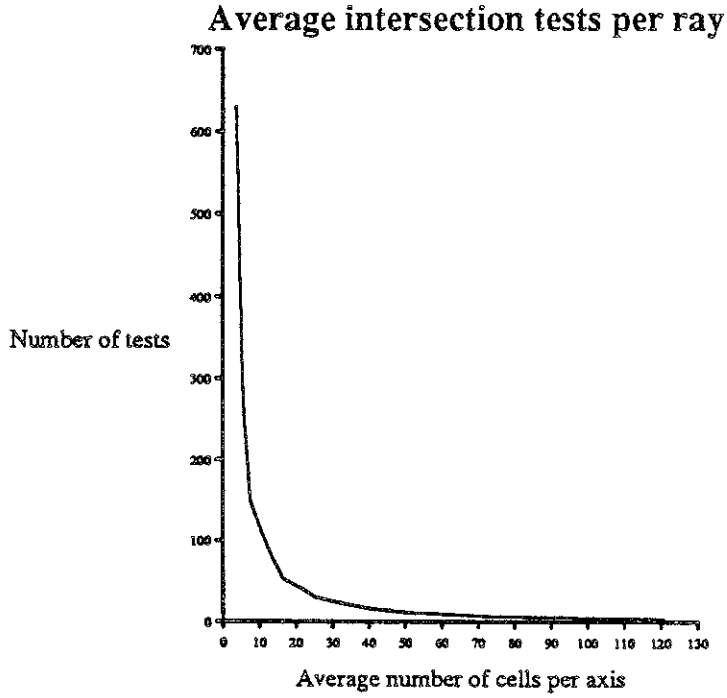


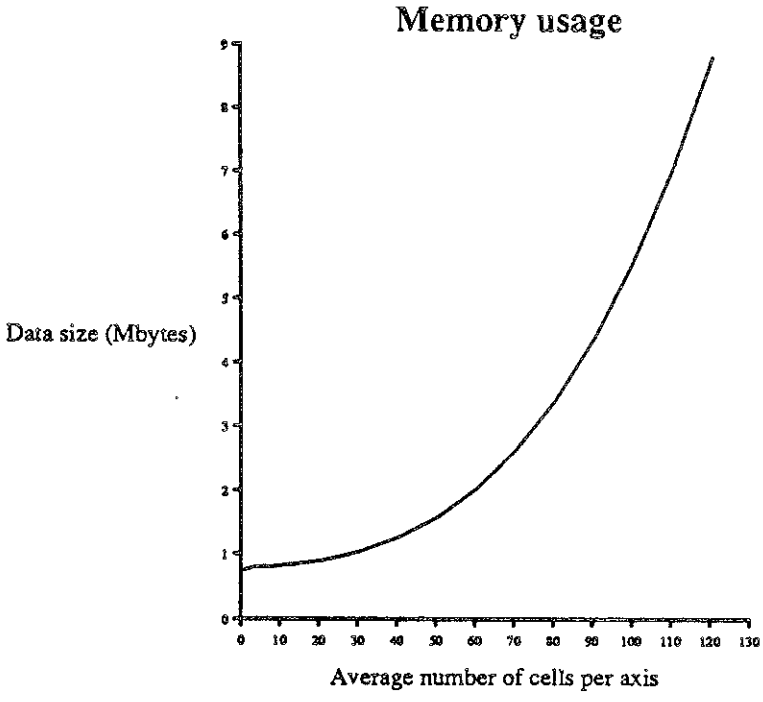
Distribution of execution time



Average faces per cell







Although theoretical and empirical observations confirm that an optimal number of cells minimizes ray intersection and ray propagation times, this number is heavily dependent on the extent and distribution of primitives within the scene. Ideally, the scene would be statistically

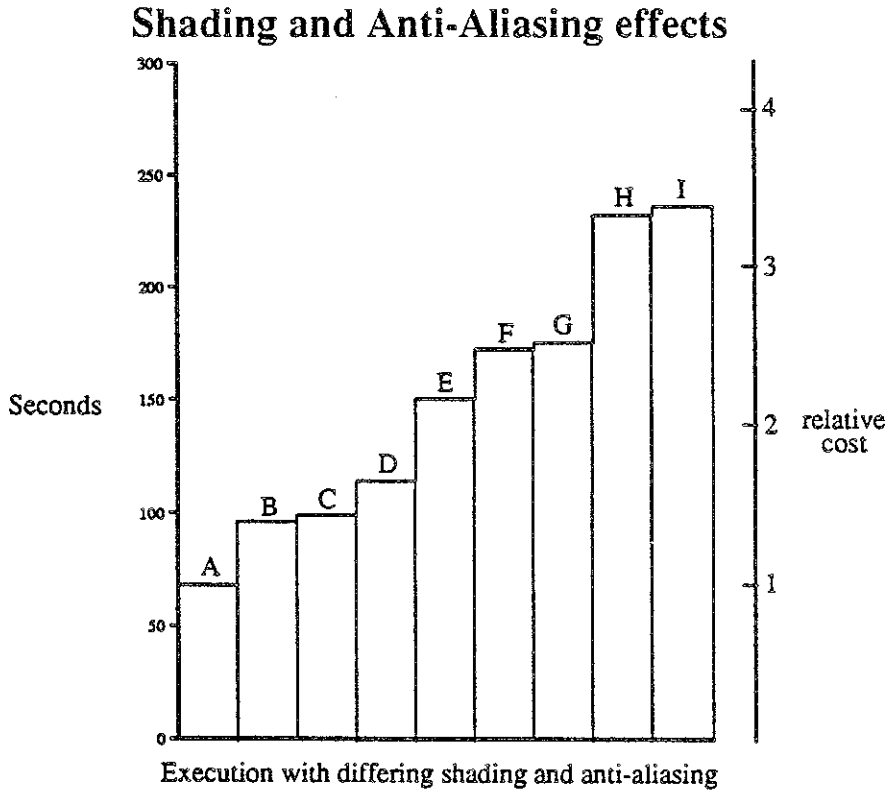
analyzed while it is being flattened, and a near-optimal number of cells could be determined algorithmically. Unfortunately, the statistical analysis of a scene appears to be a difficult problem. The only method available at present is to render several images of a scene at low resolution with different numbers of cells, and observe the resulting execution times. This will give the user an idea of the performance curves, and a good estimate of the optimal cell allocation can be derived for rendering a final image at higher resolution.

Fortunately, the increase in execution time caused by increasing ray propagation time is small. Therefore, it is much more beneficial to overestimate the number of cells required than to underestimate. Finer subdivisions will require more memory, however, and the real rendering time of the algorithm (as opposed to the cpu time) may suffer if many page faults result from the increased memory usage. Since the amount of physical memory and the number of competing processes vary from system to system, the user may have to determine the cost of increased memory usage on his particular system.

9.2. Computational Cost of Shading Effects

Once an optimal number of cells has been found for a particular scene, the shading computations required at each surface point are the major factor determining rendering time. In this section we will consider the scene pictured in figure C.2 (appendix C). We will begin with a very simple shading model, and gradually increase the complexity of the shading model to get an idea of the computational cost of each calculation.

In the following graph, nine images were rendered at a resolution of 256 by 256 pixels on a Vax 8650. In each image, one component of the shading or anti-aliasing routine was changed. Our hope is that careful examination of the height of these bars will give the reader a feeling for the relative cost of each of these computations.



Execution with differing shading and anti-aliasing

Image A: Simple shading, no anti-aliasing.

Image B: Same as *A*, but using inexpensive adaptive anti-aliasing.

Image C: Same as *B*, but smooth shading on the planet.

Image D: Same as *C*, but shadows from one directional light source.

Image E: No shadows, but reflections on tiles and letters.

Image F: No reflections, shadows from two directional light sources.

Image G: Reflections plus shadows from one directional light source.

Image H: Reflections plus shadows from two directional light sources.

Image I: Same as *H*, but with a small amount of fog.

9.3. Computational Cost of Transparent Surfaces

As we noted in section 7.2, transparent surfaces that reflect and transmit light are the most expensive surface types to model, because many rays can be generated from a single primary ray. In this section, we will show how a change in the specification of a surface can affect the renderer's execution time.

Our test scene is figure C.3 (appendix C). This is a particularly expensive image to compute for the following reasons. First, it is very non-uniformly distributed. Most of the 3576 polygons in the scene are concentrated in the gears, but the spatial subdivision must also encompass the large tiled plane. Although 250,000 cells were allocated in the spatial subdivision, some of the cells contained more than 120 polygons. A maximum of 5 to 10 polygons in a particular cell is much more desirable.

The second factor that makes this image expensive is the fact that we are calculating shadows from three directional light sources. Every visible point generates three shadow rays to determine if it is in shadow with respect to the corresponding light source.

Third, the presence of glass in the scene requires the generation of many reflected and refracted rays. We set the maximum ray depth to 10 in this image, which means that potentially $2^{10} - 2 = 1022$ rays could be generated from a single primary ray in the worst case. Although this worst case probably does not occur within this image, the maximum ray depth did reach 10, indicating that cut-off caused by attenuation does not occur for some rays in this image earlier than depth level 10.

Finally, we generated this image at high resolution (1024 by 768 pixels). Obviously, a lower resolution image requires the generation of fewer rays and requires less time. All images were computed on a Vax 8650.

Image *A* was produced without the surfaces that define the glass panels. Image *B* includes transparent surfaces only (no volume properties such as index of refraction or attenuation) and generates only transmitted rays rather than transmitted and reflected rays. Image *C* models the panels as transparent volumes. The directions of the refracted and reflected rays and their relative contributions are calculated from the index of refraction. Rays travelling through the transparent volume are also properly attenuated. Image *D* (figure C.4) models the panels as transparent surfaces that generate transmitted and reflected rays. Interestingly enough, this image was more expensive to compute than the more realistic model in image *C* because rays were not attenuated by a transparent volume and therefore reach greater depths. Image *E* (figure C.5) changes the definition of the gears so that they are modeled as transparent volumes as well as the glass panels.

Computational Cost of Transparent Surfaces

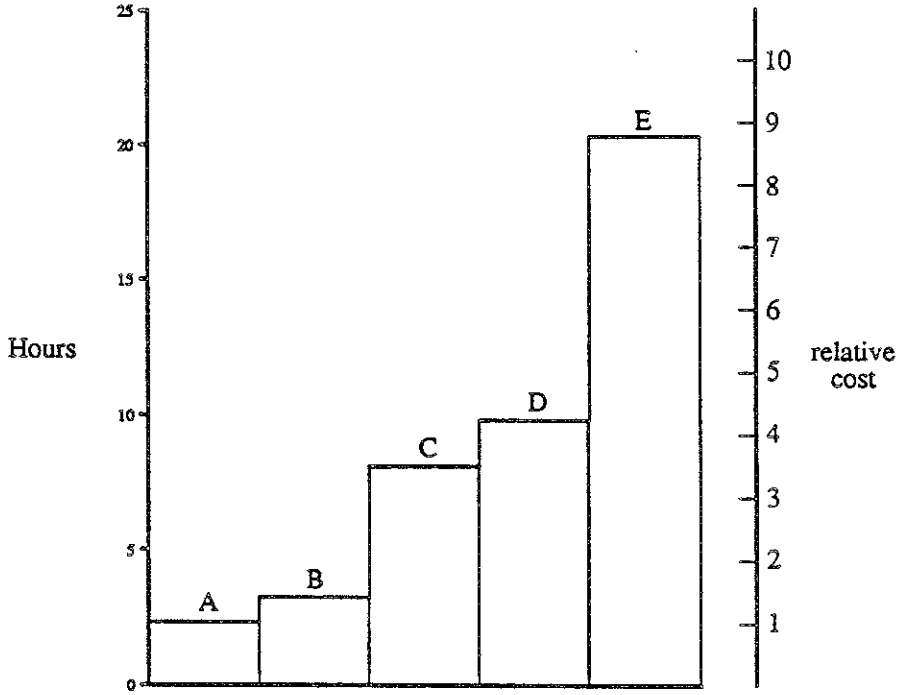


Image A: No transparent surfaces.

Image B: Panels are transparent non-reflecting surfaces.

Image C: Panels are transparent volumes (with attenuation and index of refraction) and reflected and refracted rays.

Image D: Panels are transparent reflecting surfaces.

Image E: Gears and panels are transparent volumes.

10. Conclusions

Our experience has shown that use of a uniform spatial subdivision and careful optimization of the ray intersection routine (for different ray types as well as different primitives) has reduced the cost of ray-casting significantly. Depending on the resolution of the final image, the number of polygons in the scene, and the overall distribution of polygons, the basic ray-casting algorithm can sometimes outperform scan-line algorithms (in both speed and memory requirements). The algorithm is particularly attractive in scenes containing large numbers of polygons that individually cover only a few pixels in the final image. The ever-increasing complexity of computer graphics images requires us to develop algorithms that work well as the number of primitives increases. The favorable performance of optimized ray-tracing in this regard suggests that it will continue to be an important algorithm in the future.

Although the speed of ray-casting is competitive with current scan-line algorithms, the computational cost of full ray-tracing (including shadows, reflected rays, and transmitted rays) is still expensive, because the number of rays to be traced becomes so large. We have attempted to reduce this cost by employing an inexpensive anti-aliasing technique that generates a minimal number of primary rays. We also allow the user to restrict reflections and transmission of light globally and on a polygon by polygon basis to reduce the number of reflected and transmitted rays that must be fired in a given scene. The contribution of each ray to the final image is carefully evaluated to avoid superfluous generation of secondary rays. In addition, we have suggested a simple technique to reduce the number of shadow rays that must be cast to determine if a particular surface point is in shadow.

Aside from the aforementioned shadow optimization, further major speed improvements to the basic algorithm used by *UgRay* appear unlikely. The major disadvantage of the algorithm appears in scenes where an uneven distribution of polygons causes a non-uniform loading of the cells in the spatial subdivision. This can be only slightly alleviated by allocation of more cells in the subdivision. Such scenes would require a subdivision technique that readily (and optimally) adapts to the distribution of primitives. Although the octree data structure exhibits some of these properties, it does not adapt quickly enough to changes in scene density to offset the added expense incurred during propagation of a ray from one cell to another.

Two techniques come to mind that might improve performance of non-uniformly distributed scenes. First, multiple uniform subdivisions might be allocated (perhaps hierarchically) around each part of a scene that exceeds a certain complexity. If this data structure is organized carefully (and not nested too deeply), the adaptive nature of the octree could be approximated while retaining a very fast ray propagation procedure.

Secondly, adaptive "slabs" could be used to subdivide space instead of uniform cubes. Ray propagation techniques such as DDA should be adapted so that rays could be propagated from cell to cell quickly. Integer arithmetic may not be applicable here, however, since widely varying cell sizes may require floating-point computation to maintain accuracy. The goals would be much closer adaptation to the scene than is available in an octree, and a fast ray propagation procedure, of course. The main difficulty is finding heuristics that would guide the adaptation towards a good solution.

One of our first goals in the development of *UgRay* was a fast procedure for inserting polygons in the uniform spatial subdivision. This pays off when the preprocessing time requires a significant portion of the total rendering time (during simple ray-casting, for example). But during full ray-tracing, the preprocessing time dwindles into insignificance compared to the computational expense of ray intersection calculations and ray propagation. Therefore, it would be beneficial to spend a longer time producing a good adaptive subdivision of space if this would alleviate computation during the rendering phase.

One major practical feature that is almost always required when producing images with ray-tracing is currently missing from *UgRay*. The program should include a restart capability so that it can pick up where it left off when a machine crashes or is shut down. Some of our most expensive images (such as figure C.5, appendix C) required longer to render than the mean running time of the particular computer we were using.

We have suggested many areas where the realism of images produced by *UgRay* could be enhanced using texture mapping, bump mapping, better light models and indirect lighting effects, better surface models, and a higher-quality anti-aliasing routine. Almost without exception, these will require more computation. Many of them may have to wait for more powerful hardware.

This brings us to a final observation that is almost universally accepted by the ray-tracing community. Hardware will have a large impact on the feasibility of ray-tracing for purposes such as commmercial animation in the future. Many of the lower-level intersection calculations will probably migrate to hardware, leaving software to address the more complicated issues of realistic lighting models. But in the absence of special-purpose hardware, networks of workstations working on images in parallel have been proven to be an effective technique for reducing the time necessary to produce ray-traced images. Because rays can be traced independently, the ray-tracing algorithm is relatively easy to distribute among large numbers of parallel processors. An implementation of *UgRay* that can distribute its calculations among several processors holds the most hope for future performance gains without major algorithmic redesign.

Appendix A

References

Bibliography

[Arv86] J. ARVO, Backward Ray Tracing, *ACM SIGGRAPH Developments in Ray Tracing Course Notes*, Dallas, TX, August 1986.

[Ber86] P. BERGERON, A General Version of Crow's Shadow Volumes, *Computer Graphics and Applications 6*, 9 (September 1986), 17-28.

[BiW86] G. BISHOP and D. M. WEIMER, Fast Phong Shading, *Computer Graphics 20*, 4 (August 1986), 103-106.

[BlN76] J. F. BLINN and M. E. NEWELL, Texture and Reflection in Computer Generated Images, *Comm. of the ACM 19*, 10 (October 1976), 456-461.

[Bli77] J. F. BLINN, Models of Light Reflection for Computer Synthesized Pictures, *Computer Graphics 11*, 2 (1977), 192-198.

[Bun82] W. M. BUNKER, Filtering Simulated Visual Scenes - Spatial and Temporal Effects, *Proceedings of Fourth Interservice/Industry Training Equipment Conference*, 1982, 531-540.

[Cat78] E. CATMULL, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, PhD thesis, University of Utah, 1978.

[CoG85] M. F. COHEN and D. P. GREENBERG, The Hemi-Cube: A Radiosity Solution for Complex Environments, *Computer Graphics 19*, 3 (July 1985), 31-40.

[CoT81] R. L. COOK and K. E. TORRANCE, A Reflectance Model for Computer Graphics, *Computer Graphics 15*, 3 (August 1981).

[Coo84] R. L. COOK, Distributed Ray Tracing, *Computer Graphics 18*, 3 (July 1984), 137-145.

[Coo86] R. L. COOK, Stochastic Sampling in Computer Graphics, *ACM Transactions on Graphics 5*, 1 (January 1986), 51-72.

[DiS84] M. DIPPE and J. SWENSEN, An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis, *Computer Graphics 18*, 3 (July 1984), 149-158.

[DiW85] M. A. Z. DIPPE and E. H. WOLD, Antialiasing Through Stochastic Sampling, *Computer Graphics 19*, 3 (July 1985), 69-78.

[FTI86] A. FUJIMOTO, T. TANAKA and K. IWATA, ARTS: Accelerated Ray Tracing System, *Computer Graphics and Applications 6*, 4 (April 1986), 16-26.

[Gla84] A. S. GLASSNER, Space Subdivision for Fast Ray Tracing, *Computer Graphics and Applications 4*, 10 (October 1984), 15-22.

[Gou71] H. GOURAUD, Computer Display of Curved Surfaces, *IEEE Trans.*, June 1971, 623-629.

[HaG86] E. A. HAINES and D. P. GREENBERG, The Light Buffer: A Shadow-Testing Accelerator, *Computer Graphics and Applications 6*, 9 (September 1986), 6-16.

[Han86] P. HANRAHAN, Using Caching and Breadth-First Search to Speed Up Ray-Tracing, *Proceedings of Graphics Interface '86*, Vancouver, 1986, 56-61.

[HeH84] P. S. HECKBERT and P. HANRAHAN, Beam Tracing Polygonal Objects, *Computer Graphics 18*, 3 (July 1984), 119-127.

[Hec86] P. S. HECKBERT, Survey of Texture Mapping, *Computer Graphics and Applications*, November 1986, 56-67.

[ICG86] D. S. IMMEL, M. F. COHEN and D. P. GREENBERG, A Radiosity Method for Non-Diffuse Environments, *Computer Graphics 20*, 4 (August 1986), 133-142.

[Ina86] M. INAKAGE, Reflection and Refraction Model for Ray Tracing, *ACM SIGGRAPH Developments in Ray Tracing Course Notes*, Dallas, TX, August 1986.

[Kaj86] J. T. KAJIYA, The Rendering Equation, *Computer Graphics 20*, 4 (August 1986), 143-150.

[KaK86] T. L. KAY and J. T. KAJIYA, Ray Tracing Complex Scenes, *Computer Graphics 20*, 4 (August 1986), 269-278.

[Max86] N. L. MAX, Atmospheric Illumination and Shadows, *Computer Graphics 20*, 4 (August 1986), 117-124.

[NeO86] K. NEMOTO and T. OMACHI, An Adaptive Subdivision by Sliding Boundary Surfaces, *Proceedings of Graphics Interface '86*, Vancouver, 1986, 43-48.

[NeS79] W. M. NEWMAN and R. F. SPROUL, *Principles of Interactive Computer Graphics, Second Edition*, McGraw-Hill, 1979.

[Pho75] B. T. PHONG, Illumination for Computer Generated Pictures, *Comm. of the ACM 18*, 6 (June 1975), 311-317.

[Shi86] L. A. SHIRMAN, Symmetric Interpolation of Triangular and Quadrilateral Patches between Cubic Boundaries, Tech. Report, U.C. Berkeley, Dec. 1986.

[SDB85] L. R. SPEER, T. D. DEROSE and B. A. BARSKY, A Theoretical and Empirical Analysis of Coherent Ray-Tracing, *Proceedings of Graphics Interface '85*, Montreal, 1985, 1-8.

[SSS74] I. E. SUTHERLAND, R. F. SPROULL and R. A. SCHUMACKER, A Characterization of Ten Hidden-Surface Algorithms, *Computing Surveys 6*, 1 (March 1974).

[Ull83] M. K. ULLNER, *Parallel Machines for Computer Graphics*, PhD Thesis, California Institute of Technology, 1983.

[Ups85] S. UPSTILL, *Realistic Presentation of Synthetic Images*, PhD thesis, University of California, Berkeley, 1985.

[Whi80] T. WHITTED, An Improved Illumination Model for Shaded Display, *Comm. of the ACM 23*, 6 (June 1980), 343-349.

Appendix B

Manual Pages

NAME

ugray – a ray-tracing renderer for UNIGRAFIX

SYNOPSIS

ugray [options]

DESCRIPTION

Ugray is a ray-tracing renderer for viewing UNIGRAFIX scenes with shadows, reflective surfaces, transparent (and refractive) objects, color, and anti-aliasing. These effects enhance the realism of scenes that can be displayed faster with the various scan-line UNIGRAFIX renderers such as *ugshow*, *ugplot*, or *ugdisp*.

INPUT

Ugray accepts standard UNIGRAFIX scene descriptions (with some restrictions detailed below). In order to specify surface and volume properties (such as reflectivity and transparency) and colored light sources, *ugray* accepts some extensions to the UNIGRAFIX language.

Surface Properties

The UNIGRAFIX "color" statement is used to specify various surface properties and lighting effects that are applied to polygons painted with the color. The syntax is:

c *id value* [*hue* [*saturation* [*translucency*]]]
[**kd** *n*] [**ks** *n1* [*n2*]] [**kt** *n1* [*n2*]]
[**highlight** *n*] [**li** *n*]
[**reflect** *b*] [**transmit** *b*] [**round** *b*] [**shadowed** *b*] [**castshadow** *b*] ;

The meaning of the first four parameters (*value*, *hue*, *saturation*, and *translucency*) is defined by the order in which these parameters appear. The rest of the optional parameters are introduced by keywords, and may appear in any order. *n* denotes a floating-point parameter, and *b* denotes a boolean parameter where 0 turns the associated keyword off, and 1 turns it on.

Note that the *translucency* parameter is defined in the UNIGRAFIX language specification but is not used by any of the scan-line renderers that pre-date *ugray*. This value controls the manner in which two colors are mixed when a definition incorporating colored faces is instantiated with a new color. The new instance color overwrites the old color of the faces if it is completely opaque (*translucency* of 0.0). The old color of the faces is preserved if the new color is completely transparent (*translucency* of 1.0). For intermediate values, the new color is mixed into the old color with a weight based on the value of the *translucency* parameter. When mixing is performed, only the *hue*, *saturation*, and *value* parameters are mixed; all other parameters are taken from the old color. Faces that have no color specification are painted with the new color regardless of its translucency. If no value is specified, *translucency* is assumed to be 0.0.

The **kd**, **ks**, and **kt** keywords allow the user to control the coefficients of diffuse reflection, specular reflection, and transmitted light, respectively. The range of these coefficients is between 0.0 and 1.0, inclusive, but the sum of all three cannot exceed 1.0 for any incident angle (all three will be rescaled and a warning message will be issued if this situation occurs). If only a subset of these keywords appear in a color statement, *ugray* attempts to pick reasonable values for the unspecified coefficients. If **ks** and **kt** are unspecified and the surface is used in the definition of a transparent solid, *ugray* will calculate these coefficients as functions of the incident angle and the index of refraction, giving realistic-looking results. See the tables at the end of this document to determine the default values of coefficients that are left unspecified.

The **ks** and **kt** keywords take an optional second value. If this value is present, the corresponding coefficient will vary depending upon the angle at which the surface is being viewed. *n1* is the value of the coefficient when the surface is perpendicular to the direction of view, and *n2* is the value when the surface is parallel to the direction of view. Values at intermediate angles are computed by fitting a parabola through *n1* and *n2*. For **ks**, *n1* is usually less than *n2* (specularity increases at grazing angles). For **kt**, *n1* is usually greater than *n2* (transmission decreases at grazing angles).

Following is a brief description of each of the new keywords in the color statement:

kd *n* Coefficient of light diffusely reflected from surface.

ks *n1* [*n2*]
Coefficient of light specularly reflected from surface. If *n2* is present, the coefficient varies as described above. This coefficient affects the intensity of specular highlights and reflections of other surfaces in the scene.

kt *n1* [*n2*]
Coefficient of light transmitted through a transparent surface. If *n2* is present, the coefficient varies as described above.

highlight *n*
n specifies the sharpness of specular highlights displayed on a surface by modifying the exponent of the cosine term used in Phong shading. Typical values are between 5 and 100. Higher values create the appearance of shiny surfaces. If the **highlight** keyword is not present, or if the value of *n* is 0, no specular highlights are calculated.

li *n* *Li*ght *I*nteraction coefficient. *n* specifies the degree to which specularly-reflected light interacts with the surface. Range is 0.0 to 1.0, inclusive. If *n* is 0.0, light does not interact with the surface at all, and therefore a specular highlight has the color of the light source and a reflection has the color of the reflected object. This behavior is typical of certain plastics. If *n* is 1.0, light interacts with the surface in such a way that specular highlights and reflections are the color of the surface. This behavior is more typical of metals.

round *b*
This is a hack to produce smooth shading on a polygonal surface. If no shadows, specular highlights, reflections, or transparency is used, Gouraud shading is employed. If any of the above effects are also desired, *ugray* attempts to interpolate a normal vector for the surface. This is slow and produces numerous undesirable defects due to the underlying polygonal representation. If *b* is 1, smooth shading is turned on, otherwise it is turned off.

reflect *b*
If *b* is 1, *ugray* will cast a "reflected" ray to determine which surfaces in the scene (if any) should be reflected in the current surface. The intensity of reflections is governed by the **ks** coefficient. If *b* is 0, no reflection calculation will be performed. Since reflections require much additional time to calculate, you can shorten the rendering time of your scene significantly by reducing the number of reflective objects.

transmit *b*
If *b* is 1, *ugray* will cast a "transmitted" ray to determine which surfaces in the scene (if any) should be visible through the current transparent surface. If the current surface bounds a transparent solid, the direction of the transmitted ray will be determined using the index of refraction (see "Volume Properties" below). If *b* is 0, *ugray* will not perform these calculations (the surface will therefore appear opaque). The default is 1 (cast a transmitted ray if the surface has a non-zero transmission coefficient **kt**). Since a single refracted ray can spawn dozens of other refracted and reflected rays, inclusion of transparent solids in your scene will increase rendering times dramatically.

shadowed *b*
If *b* is 1, the surface will display shadows cast by other objects (if the **-sh** option is on – see "Rendering Options"). If *b* is 0, no shadow computation will be performed on this surface, significantly reducing computation time (especially if

there are several light sources). The default is to compute shadows.

castshadow *b*

If *n* is 1, the surface will cast a shadow. It is sometimes convenient to turn off the shadow of a given object, especially if it is transparent. By default, all surfaces cast shadows. Turning this parameter off will not improve rendering performance.

EXAMPLES:

```
c shinygold .9 42 1 ks .4 .7 kd .3 li .8 highlight 13 reflect 1;
c dullgray .3 0 0 1;
c smoothorange .7 20 1 round 1 shadowed 0;
c glass 1 li 0 highlight 50 kd .05 reflect 1 transmit 1;
```

Volume Properites

In order to introduce the notion of a volume in UNIGRAFIX, *ugray* accepts an expanded "definition" statement with the optional **solid** keyword. It is up to the user to make sure that the definition contains a group of faces that form an enclosed space, and that the normal of each face points toward the outside of the solid (using the standard UNIGRAFIX rules to determine the orientation of the face normal):

def *id* [**solid** [*volpropid*]] ;

A solid defined with this extended "definition" statement can be assigned volume properties in a manner similar to the assignment of colors to faces. The volume properties are defined with a new statement:

vol *volpropid value hue saturation* [*index*] ;

The attenuation of light passing through the solid is specified by the color resulting when white light passes through one unit (in world units) of the material. This color is specified in the same manner as in the "color" statement, using *value*, *hue*, and *saturation*. Note that none of these fields are optional as they are in the "color" statement. An optional fourth parameter allows the user to specify an index of refraction. Typical values are 1.0 for air, 1.5 for glass, and 1.33 for water. If no value is specified, the default value 1.0 is used (no bending of light will occur when entering the solid from open space).

A volume property defined in the above manner can be assigned to a solid within a "definition", "instance", or "array" statement. As seen above, a volume property identifier can be given after the **solid** keyword in a "definition" statement, thereby immediately assigning the indicated volume properties to the solid being defined. Assignment of volume properties can also be delayed (or overridden) at the time of instantiation. The extended "instance" and "array" statements accept a volume property identifier after the color identifier:

i [*id*] (*defid* [*colorid* [*volpropid*]] ...*transforms*...) ;
a [*id*] (*defid* [*colorid* [*volpropid*]] ...*transforms*...) ...*transforms*... ;

A volume property specified within an "instance" or "array" statement overrides all volume properties specified within the definition or its sub-definitions. Volume properties have no effect upon definitions that aren't defined using the **solid** keyword.

EXAMPLES:

```
vol greenishglass .8 150 .2 1.5;
vol redglass .6 0 .5 1.5;
def tintedpane solid greenishglass;
  ...face and vertex statements with face normals pointing outwards...
end;
i stainedglass (tintedpane glass redglass -rx 40);
```

Colored Light Sources

Ugray allows colors to be assigned to directional or ambient light sources using an extended light source statement:

l [*id*] *intensity* [*x y z*] [*colorid*] ;

The *colorid* refers to a color defined with the "color" statement. Although the color definition may contain surface property and translucency specifications, only the *hue*, *saturation*, and *value* fields are used by the light statement. Note that the color will affect the perceived intensity of the light source (i.e., a "black" light will not illuminate any surfaces in the scene no matter what its intensity is).

If shadows are being calculated, each additional light source will dramatically increase execution time. Therefore, it is best to limit the number of light sources in a scene to one or two.

OUTPUT

Ugray makes a significant departure from older UNIGRAFIX renderers in that device drivers are no longer incorporated into the renderer. Instead of rendering an image directly on a device, *ugray* generates a file containing the image. This image file can be displayed on a variety of devices using the program *ugrim* (*Ugray Image Manipulator*).

Rendering with a ray-tracer usually requires enough time so that it is desirable to save the results in a file instead of recomputing the image each time it is viewed. Also, exclusive control of the display device is needed only during display of the final image, instead of during several hours that might be required to build an image on the device while it is being rendered. A further advantage is that new device drivers can be written quickly and easily without having to modify or recompile the renderer. See the *ugrim* manual pages for further details.

OPTIONS

All options have the same format, whether they are specified on the command line or in a command file. Options begin with a dash '–' followed by one or two lower-case letters. Options which require values are followed by the appropriate list of numbers, separated by spaces:

–ep 10 -12 15.5

Some options are on/off toggles. In this case, 0 means off, and 1 means on. The following option turns shadow computation off, for example:

–sh 0

File Options

filename

Each filename that appears without a preceding option anywhere on a command line or within a command file is interpreted as a scene file. Multiple scene files can be specified (although naming conflicts across files may cause syntax errors). If no scene file appears on the command line or within a command file, or if the keyword "STDIN" is given, the scene is read from standard input.

Ugray produces two different image files and a statistics file as output. These files can be selectively supressed or directed to different filenames. To supress output to a given file, specify "NULL" as the filename. To direct the output to standard output, specify "STDOUT", and to direct to standard error, specify "STDERR".

–of *filename*

The image is produced in the output file *filename*. Default is STDOUT (standard output).

–nf *filename*

The image is produced in the intermediate file *filename*. The intermediate file differs from the output file specified by the -of option in that no run-length encoding or buffering is used. This is occasionally useful if you want to see *ugray*'s output at the same time the renderer is producing it. Since the -of output file is run-length encoded and buffered, it is sometimes one or two scan-lines behind the current pixel being calculated. The default

destination for the intermediate file is NULL. It is legal to produce both -of and -nf image files simultaneously. See the *ugrim* manual pages for further information on file formats and methods for displaying an image at the same time *ugray* is computing it.

–sf *filename*
Write scene and performance statistics to *filename*. These statistics are valuable in determining how well your scene fits in the spatial subdivision (how many faces per cell, for example). Default destination is STDERR (standard error).

–cf *filename*
Read option values from the command file *filename*. Options are specified within a command file as if they were typed on the command line. For readability, *ugray* ignores newlines and text inside comment characters { }. Multiple command files are read in the order they are specified on the command line. Command files may not be nested.

Viewing Options

–ep *x y z*
Eye point for perspective view from this point.

–ed *x y z*
Eye direction for parallel projection.

–vc *x y z*
View center for a perspective view; i.e., this point is mapped to the center of the display.

–va *angle*
View angle for a perspective view; must be between 0 and 180 degrees, exclusively. The view angle defines the maximum angle along the larger axis of a rectangular-based viewing pyramid, anchored at the eye point. (The base of this pyramid depends on the number of pixels rendered in each direction and the aspect ratio of the pixels – see "Device Options" below.) The default view angle is automatically adjusted so that the entire scene fall within it, unless this angle would be greater than 90 degrees. In this case, the view angle is set to 90 degrees and the scene will be clipped.

–vr *angle*
View rotation. By default the y-axis points up; the displayed scene is rotated counterclockwise around the viewing direction by *angle* degrees.

Ugray centers the scene and scales it to the maximal size that will fit in the rectangle of the screen or plot. Specifying a view center or view angle for a perspective view overrides this auto scaling, and the picture may occupy only part of the screen or plot. If no eye direction or eye point is specified, the default view is **–ed** 0 0 -1, i.e., an orthogonal projection from the negative z-axis. Clipping is correctly performed for all scenes, unlike older UNIGRAFIX renderers. Actually, the ray-tracing algorithm automatically displays the correct image without a special clipping step. Therefore, objects which are outside the viewing pyramid will still be correctly reflected in shiny surfaces inside the viewing pyramid. It is also legal to place the eyepoint within a scene.

Device Options

Ugray needs to know only three device-specific pieces of information in order to render an image for a particular device: the number of pixels to render in the horizontal direction, the number of pixels to render in the vertical direction, and the aspect ratio of each pixel (the height of a pixel divided by its width, usually 1.0 for devices with square pixels). This information is provided by the following options.

–px *n* Render *n* pixels in the horizontal (X) direction. Default is 64. (Make this small for faster rendering.)

–py *n* Render *n* pixels in the vertical (Y) direction. Default is 64. (Make this small for faster rendering.)

–pa *n* Render *n* pixels in both the horizontal (X) and vertical (Y) directions. (Make this small

for faster rendering.)

-pr *n* Specify the aspect ratio of a pixel (pixel height divided by pixel width). Default is 1.0.

Rendering Options

-cn *n* Specify approximate number of cells in spatial subdivision. If *n* is zero, *ugray* will allocate (50 * *the number of faces in the scene*) cells. If *n* is -1, the renderer will not use the spatial subdivision technique (this is *incredibly* slow).

-aa *n* Specify the anti-aliasing technique to use. For increasing values of *n*, better and more costly anti-aliasing techniques are used:

0	no anti-aliasing (one ray per pixel)
1	average samples taken at four corners of each pixel
2	adaptively sample edges of pixel
3	five rays per pixel (not implemented yet)

The default anti-aliasing technique is method 2, which produces reasonably good images without too much overhead.

-sh *b* Turn shadow computation on (*b* = 1) or off (*b* = 0). Default is off.

-dt *n* Limit the depth of the ray tree to *n*. By default *n* = 5, meaning that *ugray* will follow up to 5 reflections of an initial ray. A scene containing many shiny, transparent objects causes each ray to split into a reflected ray and a transmitted ray. This can result in a binary tree of rays containing 2^5 = 32 rays from a single primary ray. Low values of *n* will reduce rendering time significantly for scenes with many reflective and transparent objects, but realism may suffer. To suppress reflected and refracted rays entirely, specify *n* = 0.

-bg *value hue saturation*
Specify a background color for rays that don't hit any object in the scene. Default is black.

-fg *n* *n* is the distance (in world units) before a ray is completely attenuated by fog. If *n* is 0, no fog is generated. Fog is the color of the background (see -bg) and attenuates light exponentially (instead of linearly as in previous renderers).

Miscellaneous Options

-h Help mode. Interactively display and prompt for values for all options.

-s Silent mode. Produce no warning messages or diagnostic output to standard error unless a fatal error occurs.

-t *b* Turn rendering trace on (*b* = 1) or off (*b* = 0). If tracing is turned on, a progress report is printed to the statistics file (see -sf option). A period is printed for each scan-line completed. This is helpful in determining how long a job has to go before completion.

SPECIFYING OPTIONS

Ugray takes a large number of command-line options. It would be a tedious task to type each of these options every time *ugray* is executed. Therefore, *ugray* provides a number of different levels at which option values can be specified.

At the lowest level, the *ugray* source code defines default values for all options. These values can be overridden at any of the higher levels.

The next level is the *.ugrayrc* file. This file contains options as if they were typed on the command line. For readability, newlines and comments within { } characters are ignored. When *ugray* starts execution, it looks for the *.ugrayrc* file in the current directory. If no such file exists in the current directory, it looks in the user's home directory. If no *.ugrayrc* file exists there, a warning message is issued and execution continues.

The *.ugrayrc* file is useful for overriding default values. It is also convenient for defining default device characteristics if you usually display *ugray* images on one type of device. (If you don't wish to use the *.ugrayrc* file, you may want to create an empty file with that name so that *ugray* won't complain every time it looks for it.)

It is common to have a set of options that apply to a particular scene. It is handy to have some place to store these options so that you don't have to retype them every time you execute *ugray*, and you don't have to rediscover a particular eyepoint when you want to regenerate an old image, for example. *Ugray* accepts a "command file" in order to specify a set of options. A command file is a file containing option specifications as they would be typed on the command line (newlines and comments are ignored as in the *.ugrayrc* file). Command files can be easily edited using a text editor in order to change one or two option values. If more substantive changes are needed, a better method will be described below.

Here is an example of a typical execution of *ugray* using a command file:

```
ugray –cf goodview.cf
```

Any options that are specified in the *.ugrayrc* file and are respecified in the command file are overridden by the values in the command file.

After *ugray* parses both the *.ugrayrc* and command files, it parses options given on the command line. Options typed on the command line override all other values. This allows you to override a few options in a command file without having to edit it. In the following command line, *ugray* is instructed to compute shadows and render at a resolution of 100 pixels, regardless of the options specified in the

```
ugray –sh 1 –cf goodview.cf –pa 100
```

Note that the order in which the options and the command file are specified makes no difference: command files are always parsed first, then the command-line options override values given at lower levels.

To make it easier to specify the numerous option values, *ugray* also has a prompted, interactive mode for specifying option values. This mode can be requested using the **-h** ("help") flag:

```
ugray –h
```

The prompted mode displays the current value for each option, and allows the user to change the value at that time. When the user is finished, *ugray* offers to save the final option values in a command file so that the renderer may be run later with the same values.

The prompted mode is entered *after* all other option specifications have been processed. This is especially useful when you simply want to verify what the renderer is about to do, or if you want to change a couple of values that were specified in a command file:

```
ugray –cf goodview.cf –h
```

Once again, the order of specification on the above command line is not important. Prompted mode is entered after all other option specification methods have been performed, regardless of where the **-h** option appears on the command line.

SHADING COEFFICIENT DEFAULTS

As noted earlier, *ugray* attempts to choose reasonable default values for the shading coefficients **kd**, **ks**, and **kt** if the specification is omitted from a color statement. The values it chooses are different depending upon whether a polygon painted with the color bounds a transparent solid or not. The following tables show the defaults chosen in each case. Note that it is sometimes preferable to omit specification of **ks** and **kt** in surfaces bounding a transparent solid so that these coefficients will be calculated as functions of the incident angle and the index of refraction.

Values supplied by the user (using the **kd**, **ks**, or **kt** keywords in the color statement) are marked with asterisks.

Coefficients for simple polygon			
kd	**ks**	**kt**	Comments
1.0	0.0	0.0	Completely diffuse surface.
*	0.0	0.0	Diffuse surface.
0.0	*	0.0	Smooth mirror.
*	*	0.0	Dusty or scratched mirror.
0.0	0.0	*	Dull transparent surface.
*	0.0	*	Dull diffuse transparent surface.
0.0	*	*	Shiny transparent surface.
*	*	*	Dusty or scratched transparent surface.

Coefficients for polygon bounding a solid			
kd	**ks**	**kt**	Comments
0.0	*f*(angle,index)	*f*(angle,index)	Realistic glass.
*	*f*(angle,index)	*f*(angle,index)	Realistic dusty glass.
0.0	*	1.0 – ks	Smooth transparent solid.
*	*	1.0 – ks – kd	Dusty transparent solid.
0.0	1.0 – kt	*	Smooth transparent solid.
*	1.0 – kt – kd	*	Dusty transparent solid.
0.0	*	*	Smooth transparent solid.
*	*	*	Dusty transparent solid.

RESTRICTIONS

For the sake of simplicity and speed, *ugray* currently accepts only planar, convex polygons with a single contour. Scenes containing concave polygons, polygons with holes, or polygons with multiple contours should be submitted to the UNIGRAFIX utility program *ugtess* before rendering with *ugray*. Polygons that do not conform to the above restriction will cause erroneous output and error messages that may not diagnose the source of the problem.

UNIGRAFIX "wire" statements are accepted but ignored, since it is infinitely improbable that an infinitely thin ray will ever intersect an infinitely thin wire.

BUGS

Light passing from a light source to a surface through a transparent solid is not refracted. This is a very difficult computational problem to solve in the general case. For practical reasons, such light rays are not attenuated by fog either.

Ambient light is not attenuated inside a transparent solid. This is also a difficult problem to solve in a general case.

Solids may be nested inside each other, but if a ray leaves the solids in some order other than the reverse of the order it entered them, it will get refraction indexes confused. This means that solids with different refractive indexes must be completely nested inside each other in order to get correct results.

A non-existent scene file specified within a command file causes the program to immediately abort. It is sometimes desirable to read a command file for the other option values it contains, regardless of the existence of the scene files mentioned.

FILES

~ug/bin/ugray	The *ugray* program.
~ug/src/ug2/ray/ugray	Source code.
~/.ugrayrc or ./.ugrayrc	Initialization file.

SEE ALSO

ugrim(UG)	To display *ugray* images.
ugtess(UG)	To tesselate concave and multi-contoured polygons.

AUTHOR

Don Marsh
dmarsh@degas.Berkeley.EDU

NAME

ugrim – *Ugray Image Manipulator*

SYNOPSIS

ugrim [options]

DESCRIPTION

Ugrim performs a number of useful functions on image files produced by the UNIGRAFIX ray-tracing renderer *ugray*. *Ugrim* can display image files on various output devices, choose a reasonable color map for devices that require it, expand or compress an image using run-length encoding, print information and documentation accompanying an image, and change the documentation.

OPTIONS

filename

If a filename is specified on the command line, *ugrim* reads it and, if necessary, modifies it. If no filename is specified, *ugrim* expects to receive valid image data on standard input, and will write modifications (if required) to standard output.

–D *device*

Display the image on the specified *device*. *device* is the name of a device information file residing in the directory name contained in the environment variable **UGRIMINFO** (you should normally set this environment variable to the standard directory ~ug/bin/ugriminfo). If an appropriate device information file is not available for your device in this standard directory, it is not a difficult task to create a new driver if you have nominal experience with the C programming language and you know how to write pixel values to your device – see below.

–c Don't clear the device display before transferring this image.

–g Display the image in shades of gray instead of color. This is especially useful when you want to view a color image on a device that requires a color map. Since color-mapping is a time-consuming process (see **-cm** option) and can only be performed after the entire image has been completed, the **-g** option is useful for previewing an image on a color-mapped device. The color map is simply initialized to shades of gray and the image is displayed in the appropriate shades. The **-g** option also works on devices that don't require color maps.

–cm *n* Modify the image so that it uses a color map with *n* entries. Color mapping will be invoked automatically if you attempt to display a non-color-mapped image (which *ugray* produces) on a device that requires a color map. Since color mapping is a time-consuming process, you may want to calculate and save a color-mapped version of your image which can then be displayed immediately. The process of color mapping begins by histogramming the entire image and then attempting to choose a set of *n* colors that best represents the range of colors in the original image. To reduce contouring effects, colors are randomly dithered between transitions.

–nm This option defeats automatic color-mapping when a non-color-mapped image is displayed on a device that requires a color map. *Ugrim* attempts to partition the color map in a way to cover as much of RGB space as possible (regardless of what colors are actually needed to display the given image). The result is nearly always gross contouring and coloring errors, but this provides another method of previewing an incomplete image on a color-mapped device (see **-g** option).

–i Display image information (horizontal and vertical resolution, run-length encoded or not, color-mapped or not, and aspect ratio of pixels).

–d Display the documentation associated with this image file.

–cd *filename*

Change the documentation associated with the image file to the text contained in

filename.

-f "Follow" an incomplete image. If a premature end-of-file is detected, the program will wait for more data to be added to the image file instead of aborting execution. This is especially useful in conjunction with the **-D** option to monitor the progress of an image as it is being computed. *Ugrim* will periodically update the device display as new data is added to the end of the specified image file. *Ugrim* will exit normally when the image is completed (or you can kill the process if you get bored without affecting the image being computed). Note that the **-g** or **-nm** should be used if you are displaying on a device that requires a color map. Also, slightly better response can sometimes be achieved by "following" the intermediate file produced by *ugray* (with *ugray*'s **-nf** option), because this file is not run-length encoded or buffered.

-en Run-length encode an image. This effectively reduces the size of most images. No effect if the image is already run-length encoded.

-ex Expand an image by removing all run-length encoding. This is usually only useful if you want to look at individual pixel values with the Unix dumper *od*.

-s Silent mode. Suppress all unnecessary messages (the color mapper is especially verbose, but the information it prints is usually interesting).

Multiple options can be specified simultaneously if they make sense (it is possible to change an image's documentation and run-length encode it during the same execution, for example). It is not possible to run-length encode, expand, or change the documentation of an image at the same time you display it using the **-D** option.

EXAMPLES:

```
ugrim -D iris -d -i ball.im
ugrim -D vx -cm 128 thing.im
ugrim -cd doc -cm 256 -ex <old.im >new.im
```

Creating a New Device Driver

Three things are needed in order to display *ugrim* images on a new device. The first thing is a "device information file" that gives *ugrim* information about the resolution and aspect ratio of the pixels on your device. The device information file is a short text file similar to a *ugray* command file (this isn't coincidence – *ugray* may look at this information file someday in order to get default device information). Information is specified using options with command-line syntax. Comments can be included within brace characters.

It is easiest to see the structure of a device information file by looking at an example:

```
{*** Ikonas (low-resolution) device information file ***}
-px 480                 {pixels horizontal resolution}
-py 512                 {pixels vertical resolution}
-pr 0.69                {pixel aspect ratio (vertical / horizontal)}
-cm 0                   {device color map size (0 for no color map)}
-dd ~ug/bin/ugrimdev/ik {device driver}
```

The meaning and usage of each of these parameters is as follows:

-px *nx*

-py *ny* *Ugrim* uses the number of pixels your device displays in the horizontal direction *nx* and the number of pixels in the vertical direction *ny* to center an image that is smaller than your display, or crop an image that is too large.

-pr *n* *n* defines the aspect ratio of the pixels on your display (the vertical height of one pixel divided by its horizontal width). The value you provide here will be compared with the pixel aspect ratio of the image you wish to display. If these values differ, *ugrim* warns you that the image was calculated for a device with a different pixel aspect ratio, and the

image may be distorted on your current display device.

–cm *n* *n* is the number of color-map entries your device has (0 if your device does not require a color map). This parameter allows *ugrim* to decide when color-mapping is necessary. If you display a non-color-mapped image on a device requiring a color map, *ugrim* will first generate a color map with the number of entries specified here. *Ugrim* can display both color-mapped and non-color-mapped images on a device that doesn't require color maps.

–dd *command*

command is a command that executes a device driver for your device. *command* is expanded using C-shell conventions, so it is legal to use the tilde character for home directory specification. The device driver will be discussed in greater detail below. If the command includes spaces, place double quote marks around it, like this:
–dd "rsh degas –l dmarsh ugrimdev/ik" {device driver}

Rather than typing a device information file from scratch, you may want to copy the file ~ug/bin/ugriminfo/proto into your directory and edit it. Name your device information file with the name you wish to use in *ugrim*'s **-D** option.

The second thing you must do is set the environment variable **UGRIMINFO** to the name of the directory containing your new device information file. *Ugrim* looks for device information files in the directory named by this environment variable. You may want to link or copy the device information files into your directory from ~ug/bin/ugriminfo so that you can continue to use previously-defined devices without continually changing the **UGRIMINFO** variable.

The final thing needed to complete the definition of your device is the code that actually writes pixel values to your device. This is a small program that is named in the **-dd** parameter in the device information file. This program receives simple move and draw commands on standard input from *ugrim*. *Ugrim* takes care of all the difficult tasks such as interpreting run-length encoding, color-mapping, centering and cropping images, etc.

Copy the file ~ug/bin/ugrimdev/proto.c to your directory, and edit it according to the directions in the comments. You should modify the #define statements at the beginning (XPIXELS, YPIXELS, and COLORMAP) to match the capabilities of your device. The main procedure interprets device-independent codes sent by *ugrim*, and rarely needs to be changed for individual devices. You will need to modify the remaining procedures: *initdevice()* (do whatever initialization your device requires), *cleardevice()* (clear your device's screen, if necessary), *setdevicecmap()* (send a color map to a device that requires it), *closedevice()* (close the device and release control), *movexy()* (move the current drawing location to new coordinates), and *writepixels()* (write a run of pixel values). For most devices, these are fairly easy functions to perform, and the device driver can be suitably modified in half an hour or less.

When you can compile your device driver without errors, make sure that the **-dd** parameter in the device information file names it correctly (you may have to specify a full path name), and that the **UGRIMINFO** environment variable points to the directory containing the device information file. You are ready to go!

IMAGE FILE FORMAT

The following section is provided for users who wish to perform other operations on image files or convert them to alternate formats for use with other programs. It may be beneficial to read and perhaps use the functions defined in the file ~ug/src/ug2/ray/ugrim/imagefile.c. This module contains the interface for reading and writing *ugrim*-format image files.

All *ugrim* image files begin with a header section that contains identification and version information, the number of pixels in the horizontal and vertical directions, and other things. The contents of this header are summarized in the table below.

Ugrim image file header		
Byte	Contents	Comments
0 1 2 3 4	'U' 'G' 'R' 'I' 'M'	"UGRIM" identification string
5	*version*	Format version number
6	*runcoded*	Run-length encoding flag
7 8	*xpixels*, MSB *xpixels*, LSB	Number of pixels horizontal
9 10	*ypixels*, MSB *ypixels*, LSB	Number of pixels vertical
11 12	*aspectratio*, MSB *aspectratio*, LSB	Aspect ratio of pixels * 16384
13 14	*cmapsize*, MSB *cmapsize*, LSB	Number of entries in color map
15–*n*	*documentation*	Null-terminated documentation string (maximum length 4096 characters).

If the given image uses a color map, *cmapsize* (indicated in the header) color specifications follow the header. Each color specification contains 3 bytes, indicating the red, green, and blue intensities of the color. The range of intensity values is between 0 and 255. If the image has a color map with 64 entries, 64*3 = 192 bytes of color map information would follow the header. The red, green, and blue components of color number 0 would be given in the first three bytes of this block. The red, green, and blue components of color number 1 would occupy the next three bytes, and so on. Image data begins after the color map. If *cmapsize* is zero, no color map information is contained in the image file, and image data begins immediately following the header.

The format of image data depends on whether the image has been run-length encoded, and whether it uses a color map. If the image has been run-length encoded, the value of *runcoded* (see header) will be 1, otherwise it will be 0. If the image is not run-length encoded, image data is simply sequential pixel colors (starting at the upper left corner of the image, and proceeding right to left and top to bottom). A pixel color can be specified with 3 bytes indicating red, green, and blue intensities in the range 0 to 255, if the image does not have a color map, or the appropriate color number if the image has a color map. The number of bytes necessary to indicate the color number depends on the size of the color map (*cmapsize*). If the value of *cmapsize* is between 1 and 256, one byte is used. If the value is between 257 and 65536, two bytes are used. Otherwise, three bytes are used. Note that no indication is given at the end of a scanline or the end of the image data: these can be calculated using *xpixels* and *ypixels* (from the header information).

If the image is run-length encoded, access is a little more complicated. A *run* of pixels is defined to be one or more pixels that share the same color or have all different colors. Each run is preceded by a variable-length count that indicates how many pixels are in the run. The count may span any number of bytes, depending on how large it is. The high bit of each byte determines whether it is the last byte in the count string. If the high bit of a particular byte is 0, the high bit is discarded, and the remaining bits are appended to bits that may have been specified in previous bytes. If the high bit is 1, the high bit is discarded, the remaining bits are appended to bits from previous bytes, and the process is repeated on the following byte. To specify a count of 257, for example, two bytes would be used: 0x82 0x01. The lower seven bits of the second byte are appended to the lower seven bits of the first byte to obtain the correct value. The high bit of the

first byte is 1 to indicate that the count continues in the following byte, and the high bit of the second byte is 0 to indicate that it is the last byte in the count string.

The type of run is indicated by the next-most-significant bit (bit 6) in the first byte of the run count. If this bit is a 1, a run of identical pixel colors is produced. The color (either red, green, and blue or a color number) following the run count is duplicated the number of times indicated by the run count. If the bit is 0, a run of different pixel colors is produced. In this case, *n* pixel colors follow the run count, where *n* is the number indicated by the run count. This allows runs of different pixels to be produced without requiring separate run counts for each. Note that only 6 bits of the first byte in the run count are used to determine the run count number, since the type of run consumes an extra bit in this first byte. Also note that a run of pixels is allowed to span many scan-lines if possible. It is the user's responsibility to break long runs into several scan-lines if nececssary.

BUGS

Yet to be reported.

FILES

~ug/bin/ugrim	The *ugrim* program.
~ug/src/ug2/ray/ugrim	Source code.
~ug/bin/ugriminfo	Standard directory containing device information files.
~ug/bin/ugrimdev	Standard directory containing device drivers.
UGRIMINFO	Environment variable containing pathname of the directory containing device information files.

SEE ALSO

ugray(UG)	Ray-tracing renderer that produces *ugrim*-format image files.

AUTHOR

Don Marsh
dmarsh@degas.Berkeley.EDU

Appendix C

Sample Images

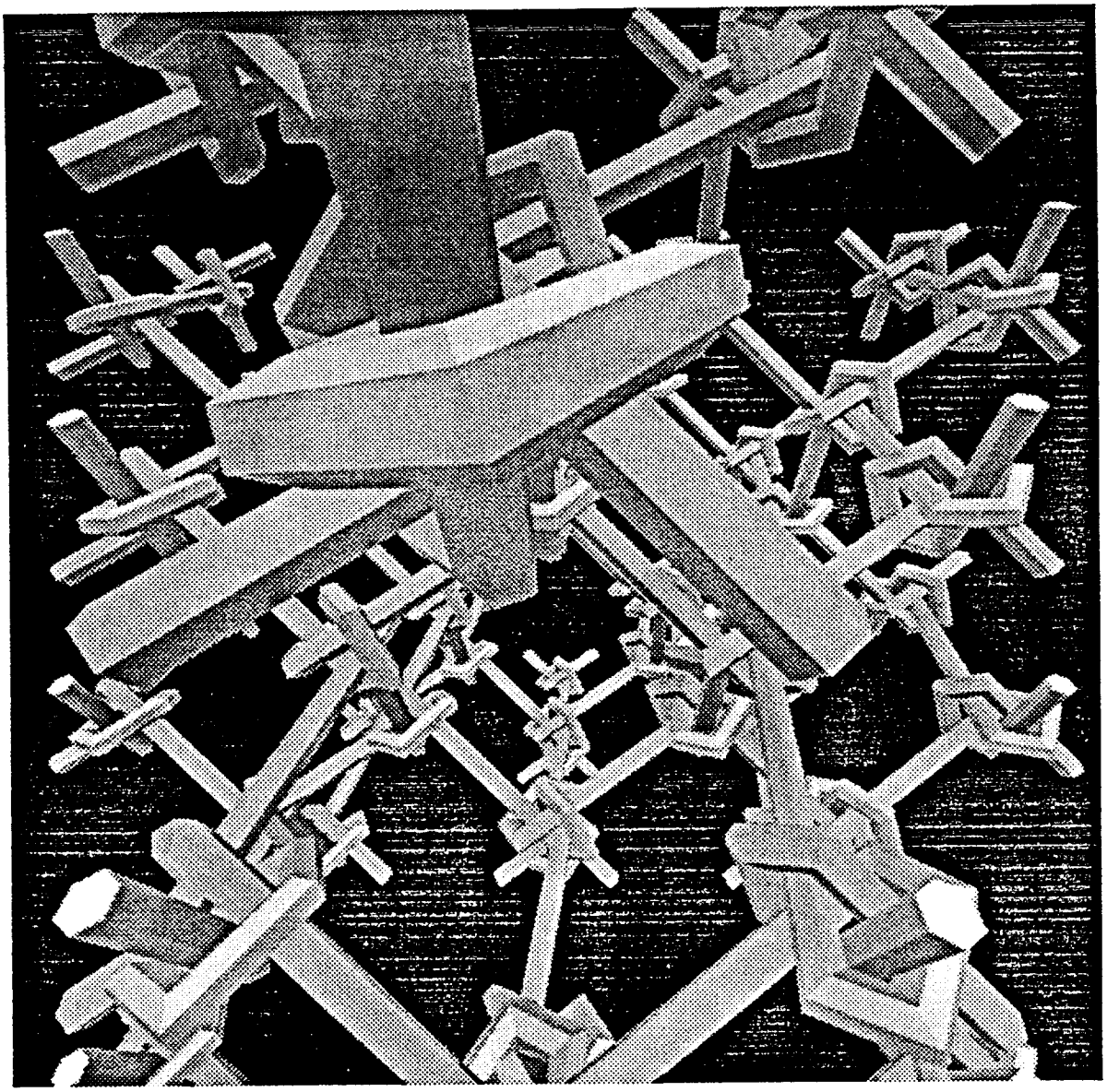


Figure C.1 – Granny-Knot Lattice

This image was rendered at a resolution of 768 by 768 pixels on a Vax 8650. No shadows, reflections, or other sophisticated effects were requested. Number of polygons: 3864. Number of cells: 305,256. Average intersection tests per ray: 6.7. Total rendering time: 11.5 minutes.

Figure C.2 – UGRAY Logo

This image was rendered at a resolution of 1024 by 768 pixels on a Vax 8650. It demonstrates shadows from two light sources (near the base of the letters), reflections, specular highlights, and smooth shading (on the planet). The star field was generated as a series of randomly-placed, randomly-sized octagons with fixed illumination values. Number of polygons: 2799. Number of cells: 153,340. Average intersection tests per ray: 10.3. Total rendering time: 39.5 minutes.

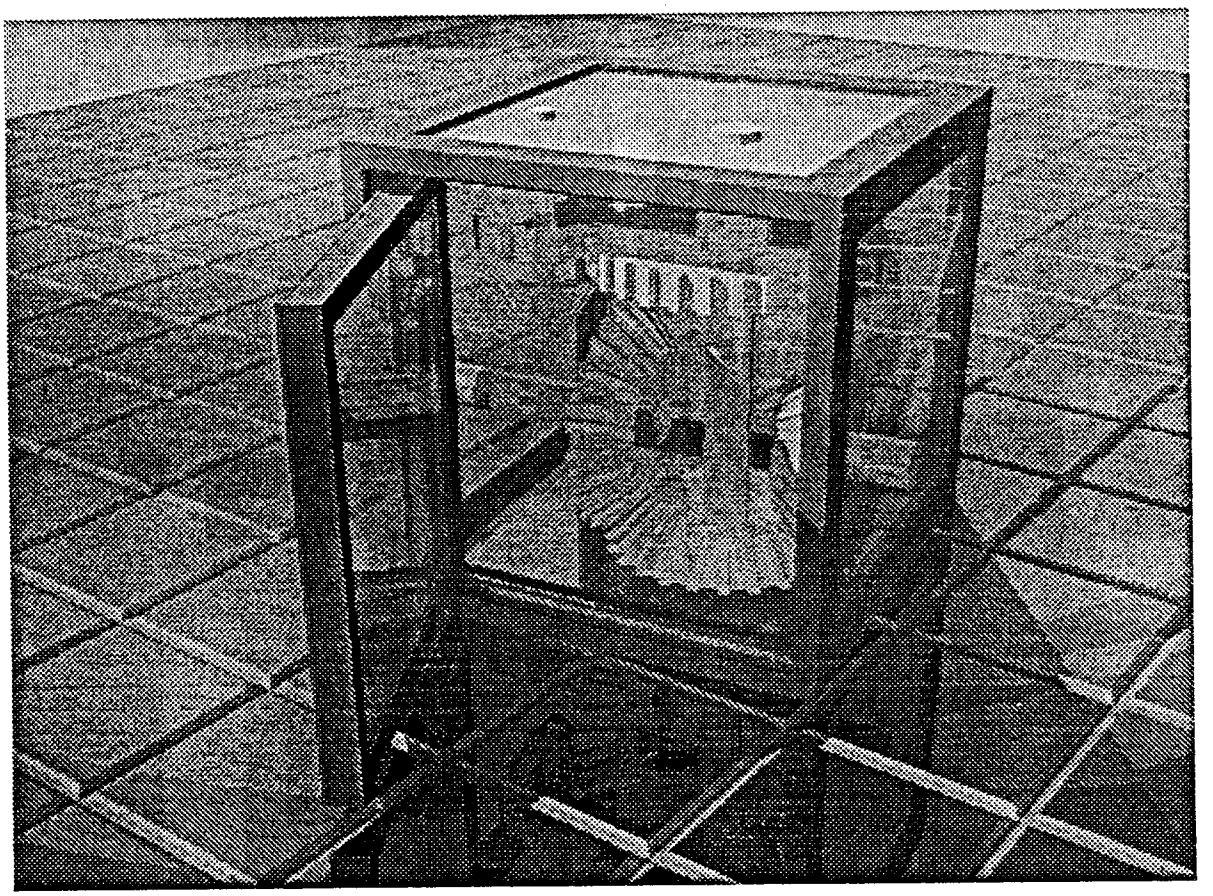


Figure C.3 – Engineering Showcase

This image was rendered at a resolution of 1024 by 768 pixels on a Vax 8650. It demonstrates shadows from three light sources, reflections, refraction and attenuation of light passing through glass, and fog. The description of the gears was initially generated using the *mkgear* program by Tom Laidig. The gears were then tesselated into convex polygons using *UgTess* by Leon Shirman. To generate realistic-looking glass, the maximum ray depth was set to 10, and some rays reached this limit. When the glass panels were omitted from this scene, the image was rendered over 3.4 times faster. Number of polygons: 3576. Number of cells: 253,440. Average intersection tests per ray: 26.6. Total rendering time: 7 hours, 57 minutes.

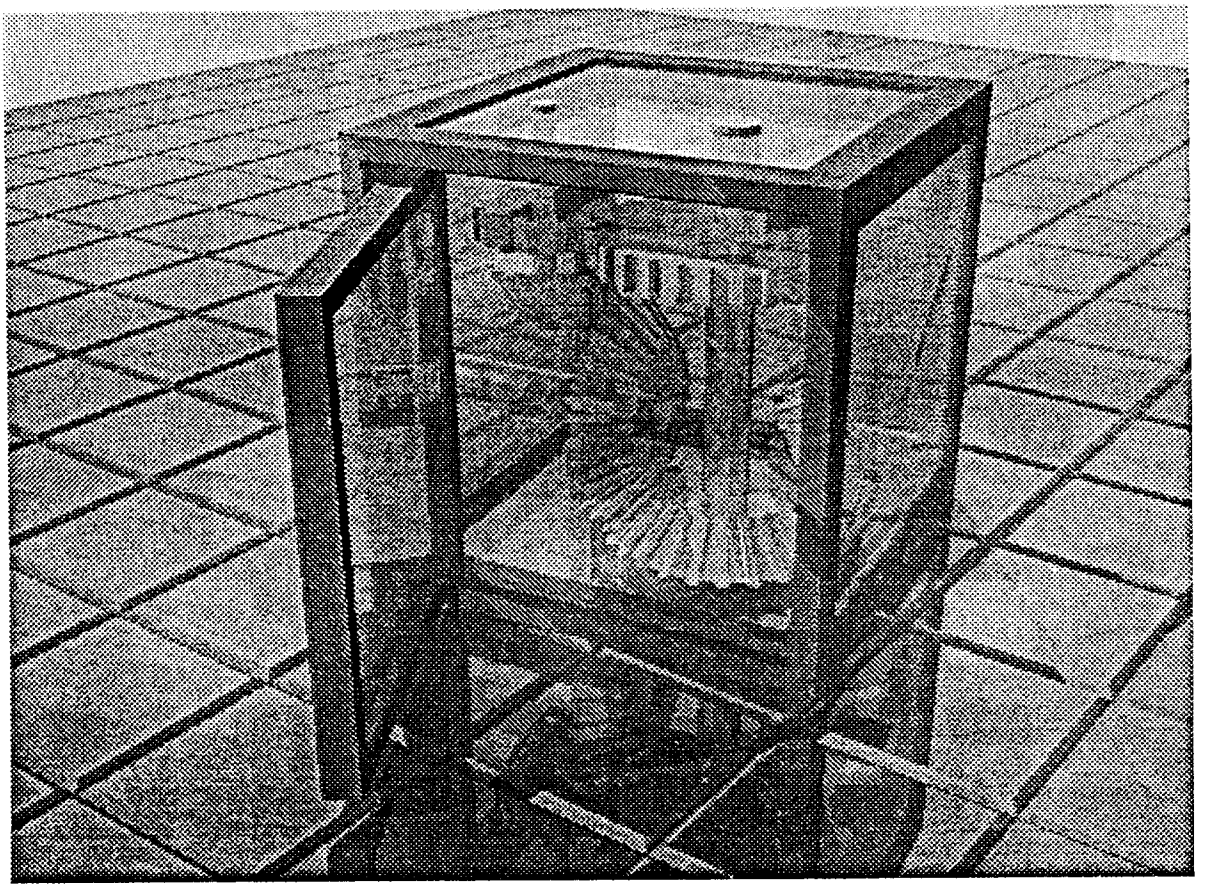


Figure C.4 – Engineering Showcase (cellophane panels)

This image is the same as figure C.3, but the transparent panels have been modeled as sets of transparent polygons instead of transparent volumes. Therefore, no refraction or attenuation effects are calculated when rays strike the transparent panels. Interestingly enough, this image required much more time than figure C.3 because fewer rays could be eliminated as a result of attenuation. Number of polygons: 3576. Number of cells: 253,440. Average intersection tests per ray: 27.7. Total rendering time: 9 hours, 48 minutes.

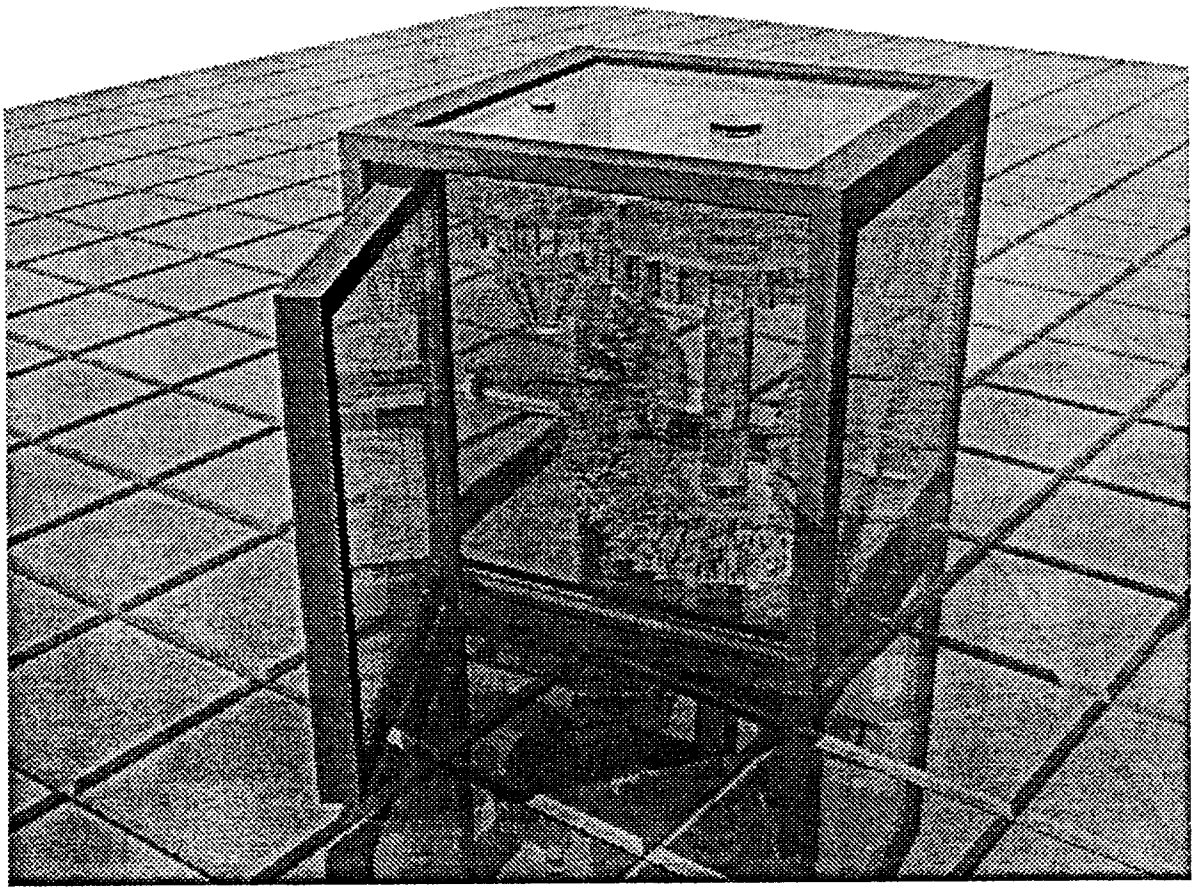


Figure C.5 – Engineering Showcase (glass gears)

This image is the same as figure C.3, but the description of the gears has been modified from reflective metal to green glass. Because glass is so expensive to render, we attempted to attenuate rays entering the glass gears as quickly as possible. Even with this consideration, this image required an incredible amount of time to render. Nearly 32 million rays were generated, and over 1 billion intersection calculations were required. Number of polygons: 3576. Number of cells: 253,440. Average intersection tests per ray: 33.8. Total rendering time: 20 hours, 19 minutes.

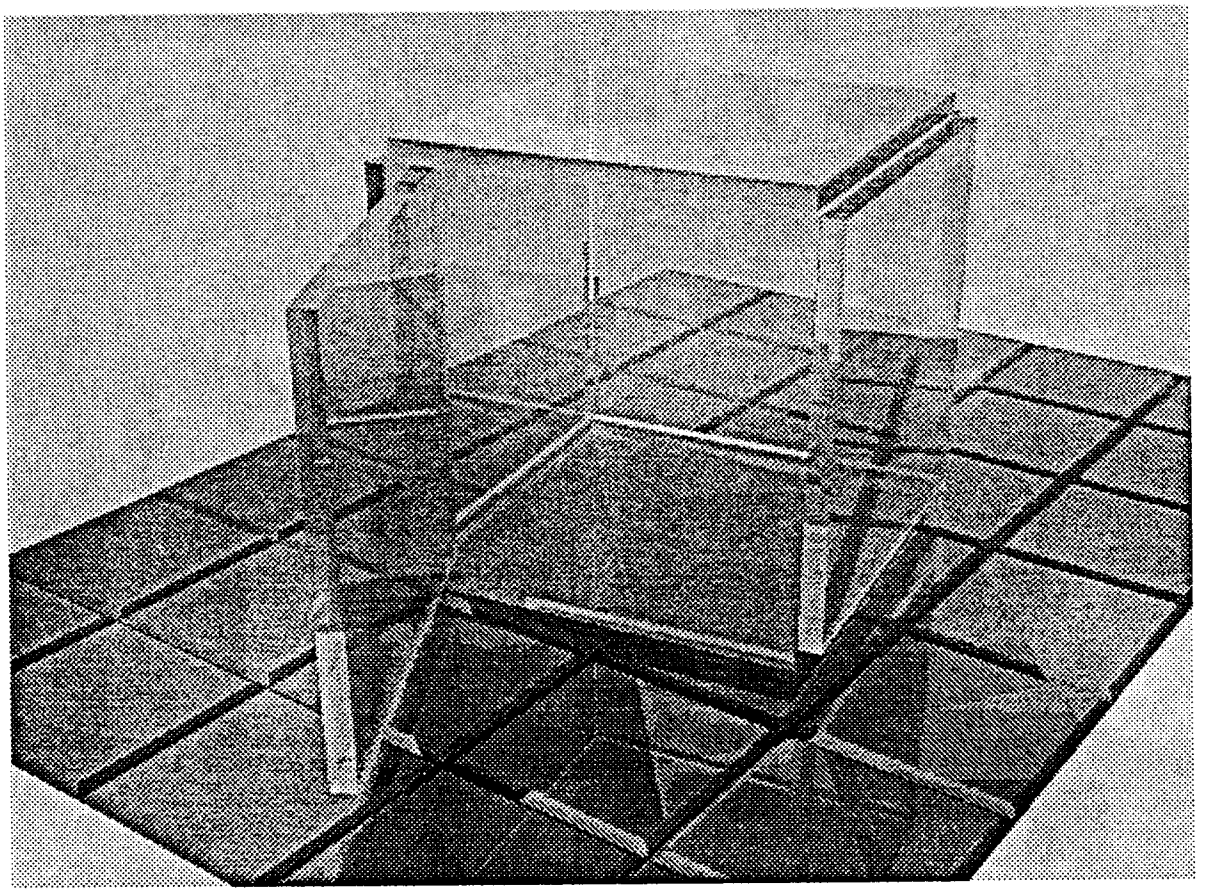


Figure C.6 – Glass Panels

This image was rendered at a resolution of 1024 by 768 pixels on a Vax 8650. We generated this image to test the realism of glass. We simply deleted the frame and the gears from the scene description of figure C.3. We also deleted some of the tiles in order to achieve a more uniform distribution of polygons, and therefore better performance using the uniform subdivision. The color version of this image shows greenish attenuation of light passing through the glass along the edges of the panels, and it looks very realistic. Number of polygons: 211. Number of cells: 81,620. Average intersection tests per ray: 2.76. Total rendering time: 7 hours, 16 minutes.

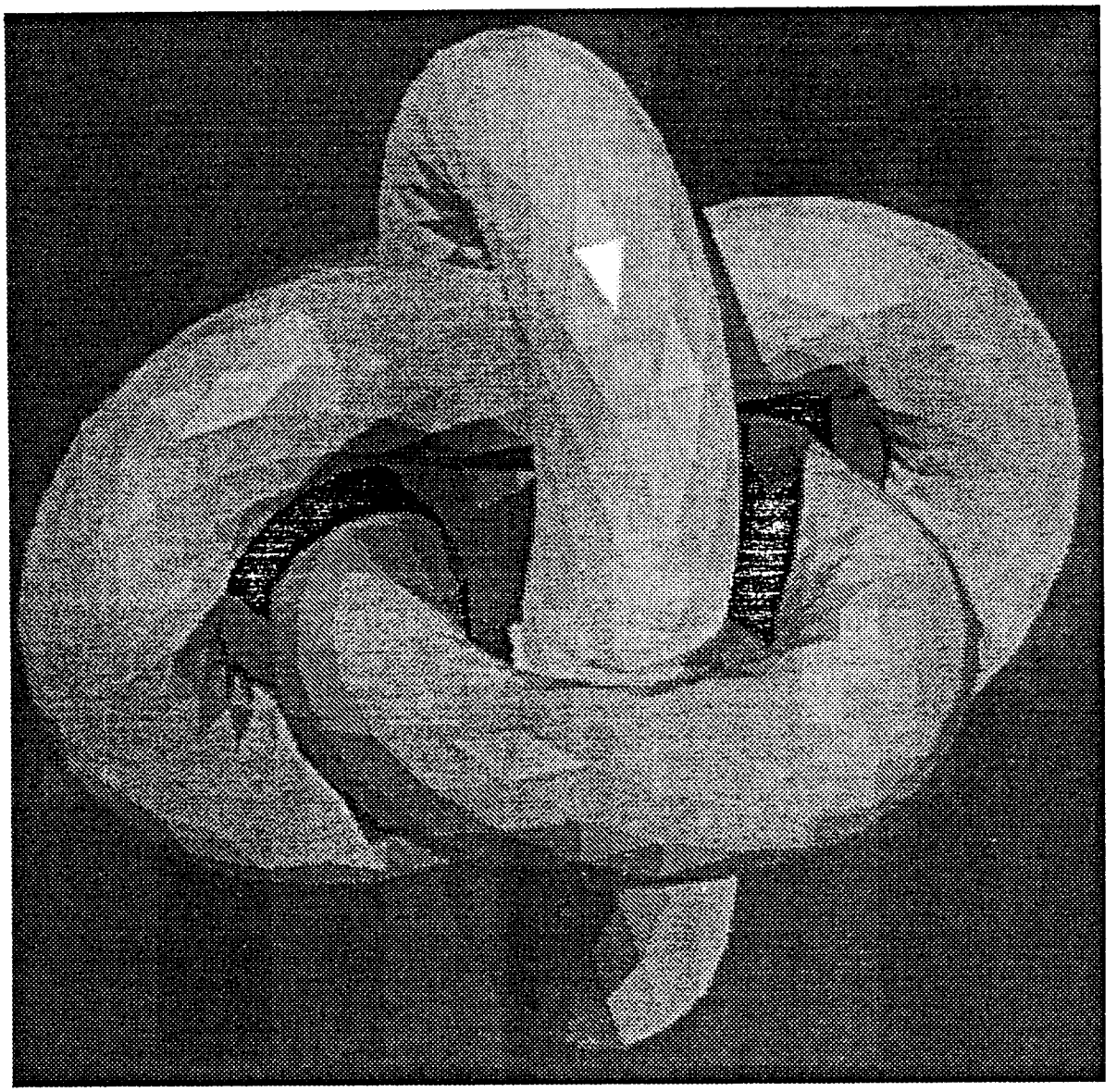


Figure C.7 – Shiny Loops

This image was rendered at a resolution of 400 by 400 pixels on a Vax 8650. It exhibits specular highlights and shadows from 3 light sources, but no reflections. Number of polygons: 3168. Number of cells: 62,160. Average intersection tests per ray: 4.5. Total rendering time: 5.5 minutes.

Figure C.8 – Reflective Loops

This image was rendered at a resolution of 400 by 400 pixels on a Vax 8650. It is the same as image C.7, but reflections are calculated to a depth of 5. Number of polygons: 3168. Number of cells: 40,320. Average intersection tests per ray: 6.7. Total rendering time: 10 minutes.

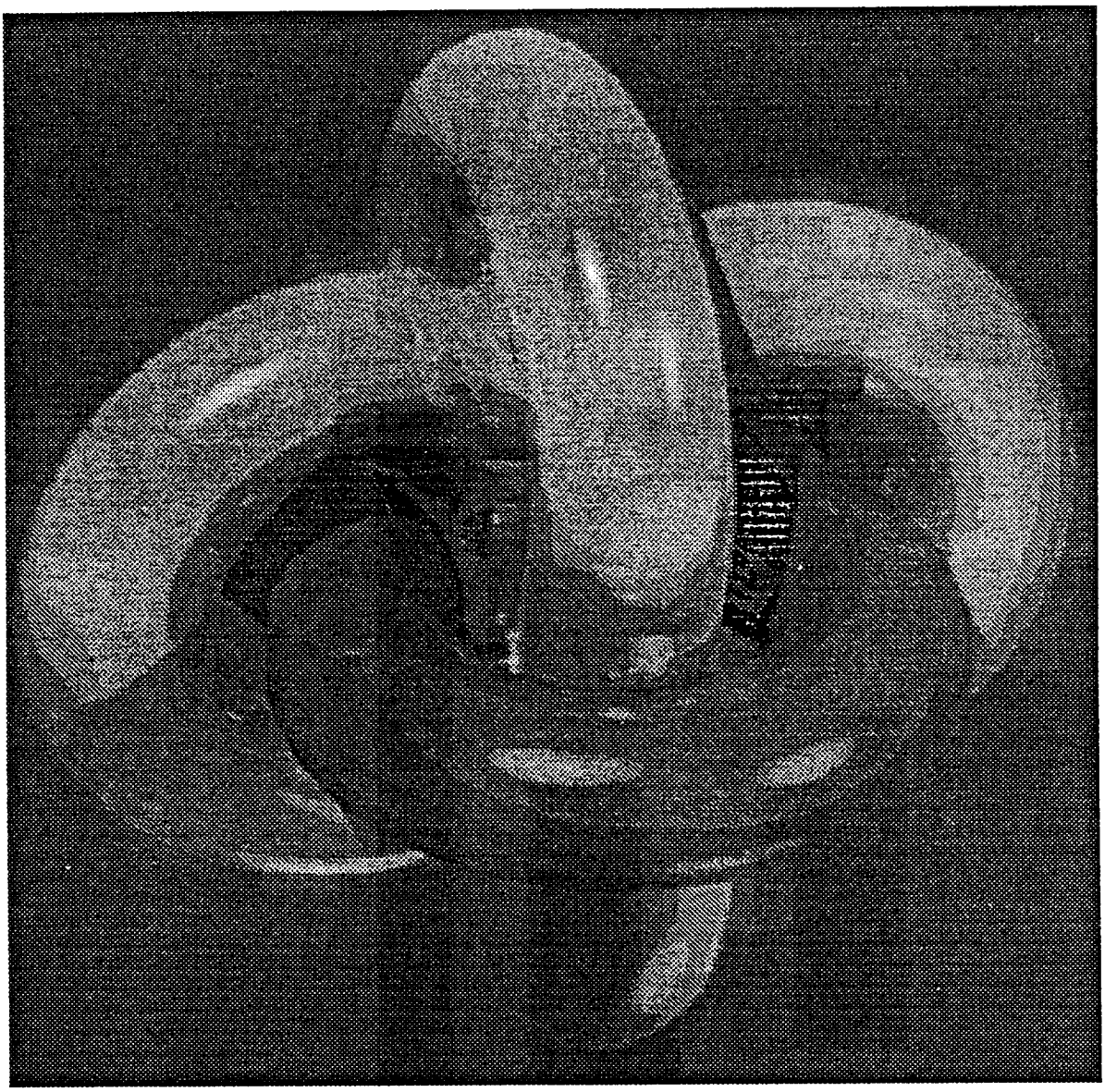


Figure C.9 – Rounded Loops

This image was rendered at a resolution of 400 by 400 pixels on a Vax 8650. It is the same as figure C.8, except normal interpolation was used to simulate a smooth surface. Although this produces some defects resulting from the underlying polygonal representation, it is a very easy way to generate smooth surfaces. Number of polygons: 3168. Number of cells: 40,320. Average intersection tests per ray: 6.58. Total rendering time: 13 minutes.

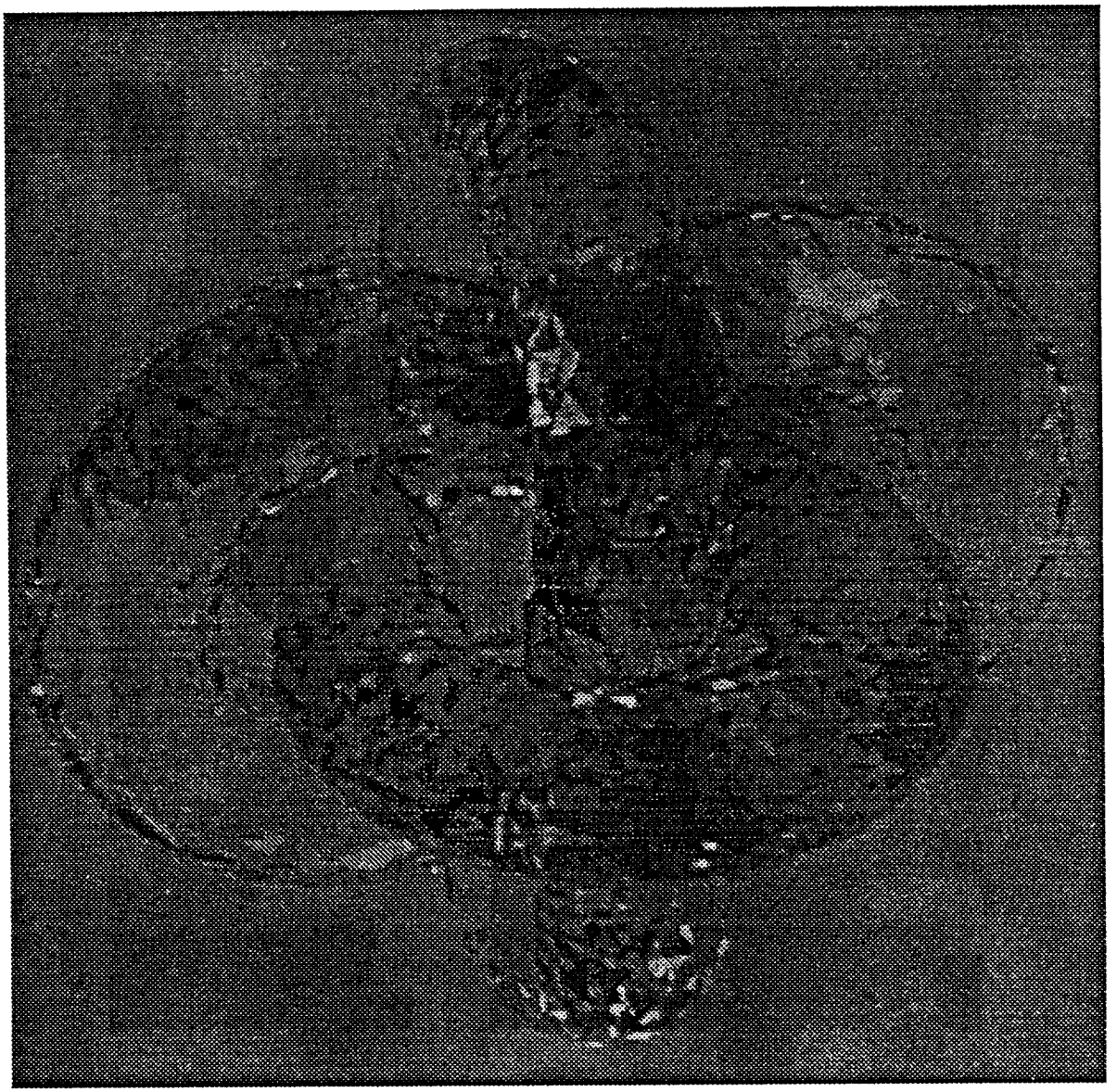


Figure C.10 – Glass Loops

This image was rendered at a resolution of 400 by 400 pixels on a Vax 8650. The model is the same as figure C.8 (no normal interpolation), but the loops have been modeled as glass instead of shiny plastic. Reflected and refracted rays were generated to a maximum depth of 5. Number of polygons: 3168. Number of cells: 40,320. Average intersection tests per ray: 8.2. Total rendering time: 1 hour, 23 minutes.

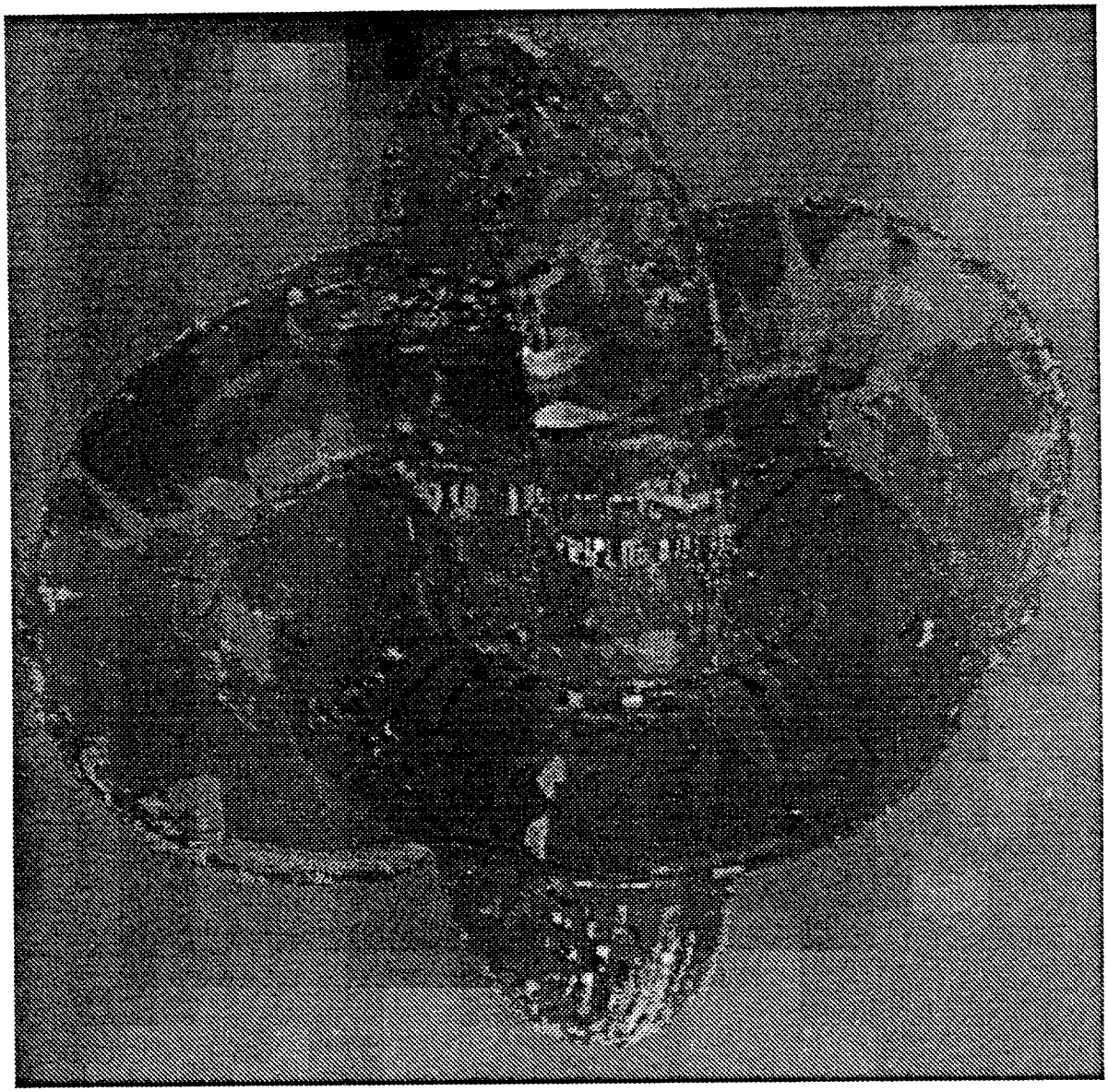


Figure C.11 – Rounded Glass Loops

This image was rendered at a resolution of 400 by 400 pixels on a Vax 8650. It is the same as figure C.10, except normal interpolation was used to simulate a smooth surface. Because the direction of refracted rays is affected by the underlying polygonal representation, many interesting defects are visible in this image. The maximum depth of reflected and refracted rays was 10 (twice that of figure C.10) in this image. Number of polygons: 3168. Number of cells: 40,320. Average intersection tests per ray: 8.45. Total rendering time: 3 hours, 35 minutes.

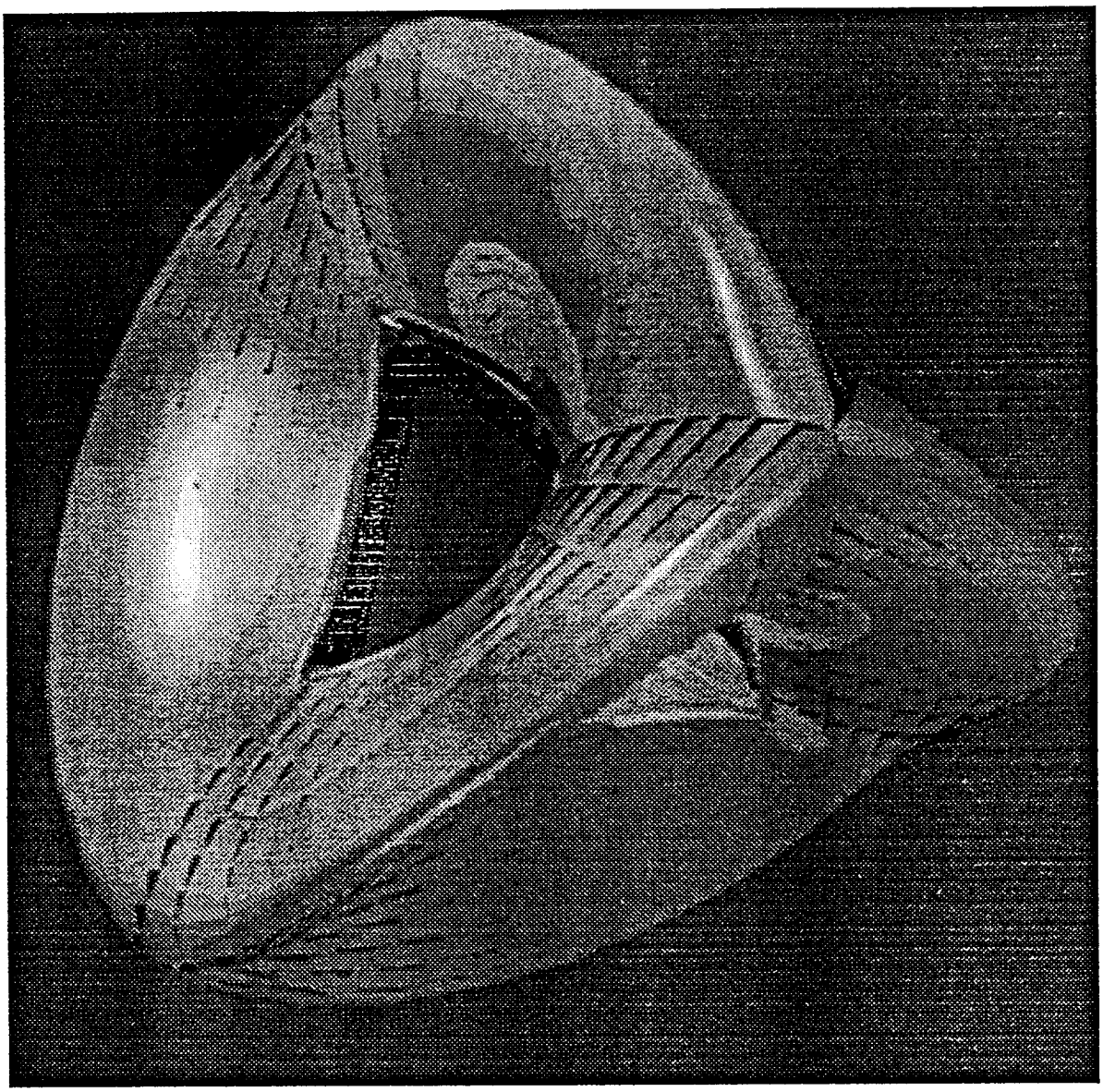


Figure C.12 – Unigrafix Flying Object

This image was rendered at a resolution of 400 by 400 pixels on a Vax 8650. A smooth polygonal mesh was generated by submitting a rough polygonal mesh in the shape of a tetrahedron to *UniCubix*. In this particular case, some non-planar quadrilaterals resulted. This produced the interesting "vent-holes" in the object. The image was rendered with highlights, reflections, and shadows from several colored light sources. Number of polygons: 2200. Number of cells: 52,022. Average intersection tests per ray: 5.4. Total rendering time: 13 minutes.

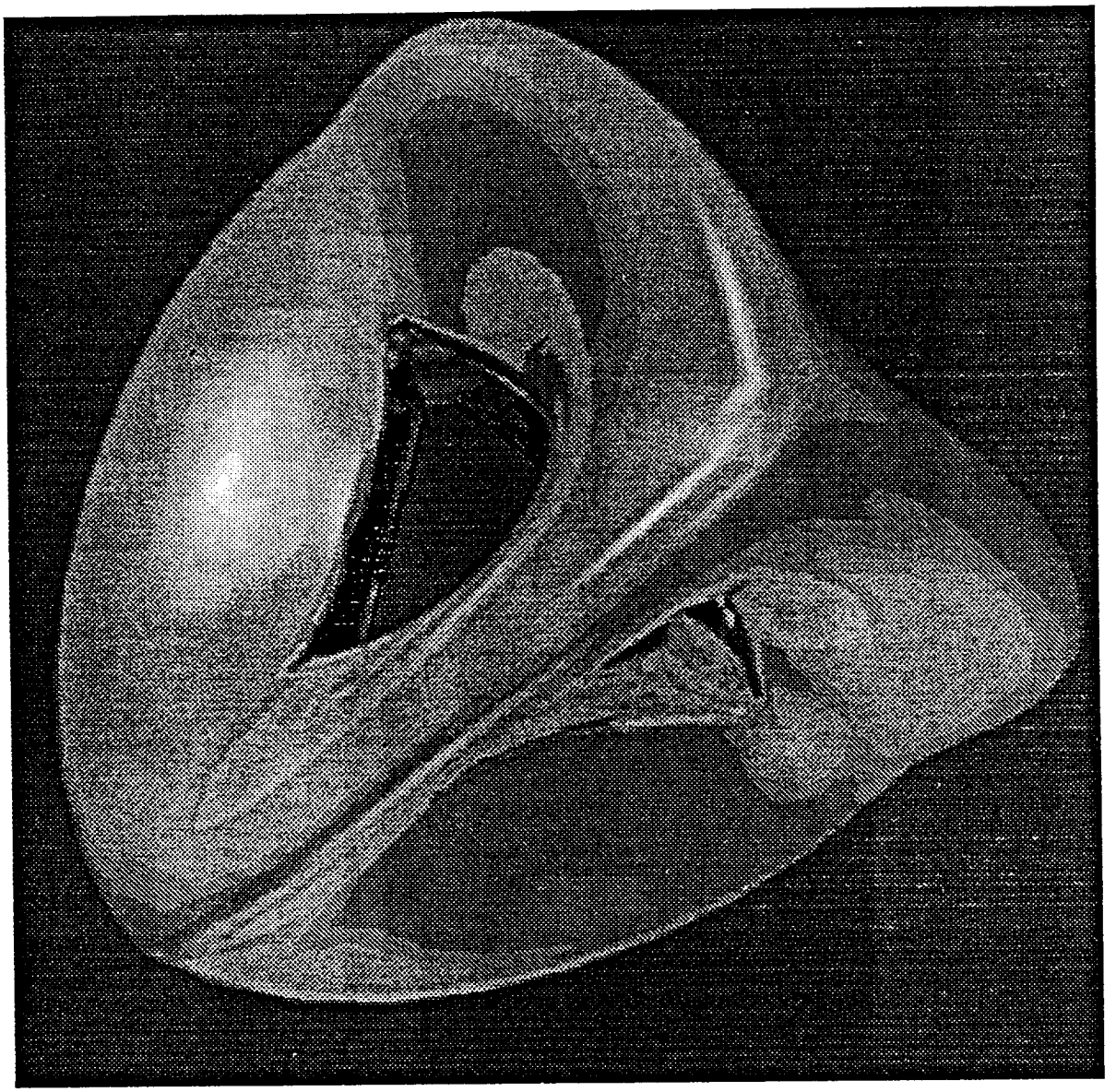


Figure C.13 – Tessellated Unigrafix Flying Object

This image was rendered at a resolution of 400 by 400 pixels on a Vax 8650. It is the same as figure C.12, except the scene description was tessellated with triangles using *UgTess* to remove non-planar polygons. Number of polygons: 4000. Number of cells: 52,022.