COMPUTER GRAPHICS AND IMAGE PROCESSING 18, 109–144 (1982)

Ray Casting for Modeling Solids*

SCOTT D. ROTH

Unimation Inc., 2211 S. Hacienda Blvd., Suite 112, Hacienda Heights, California 91745

Received December 8, 1980

This paper presents ray casting as the methodological basis for a CAD/CAM solid modeling system. Solid objects are modeled by combining primitive solids, such as blocks and cylinders, using the set operators union, intersection, and difference. To visualize and analyze the composite solids modeled, virtual light rays are cast as probes. By virtue of its simplicity, ray casting is reliable and extensible. The most difficult mathematical problem is finding line–surface intersection points. So surfaces such as planes, quadrics, tori, and probably even parametric surface patches may bound the primitive solids. The adequacy and efficiency of ray casting are issues addressed here. A fast picture generation capability for interactive modeling is the biggest challenge. New methods are presented, accompanied by sample pictures and CPU times, to meet the challenge.

1. INTRODUCTION

1.1. Modeling via Combinatorial Geometry

Using the constructive solid geometry method, solid objects are modeled as compositions of primitive solids, combined using the boolean set operators union (+), intersection (&), and difference (−). Figure 1 illustrates the three operators. Examples of primitive solids are blocks, spheres, cones, cylinders, and tori.

A solid composition is represented by a binary tree as Fig. 2 shows. The leaf nodes of the tree are primitives. The root node at the top represents the entire composition, but every node in the tree represents a complete solid.

Numerous modeling systems based on constructive solid geometry have been developed or are under development [1–7]. (Only Ref. [4], however, is based on ray casting.) The benefits of this approach are threefold:

- —the model represents a true solid with volume;
- —curved as well as planar surfaces bound the solids;
- —the combined operators are remarkably effective for modeling solid artifacts, particularly mechanical parts.

For information about constructive solid geometry in general, see the numerous papers by Voelcker and his group [2, 3, 8].

The GM solid modeling project led by Boyse [1], in which I participated, built a solid modeling system employing blocks and cylinders as primitives. Using many of the geometric conventions established there, I developed the ray casting methods described in this paper and implemented them in PL1 on an IBM 3033 computer. This system produced, at least in part, Figs. 1, 2, 6, 7, 8, 10, 11, 12, 13, 15, 16, 17, 18, 20, and 21.

*Research reported herein was conducted at the Computer Science Department of the GM Research Laboratories.

0146-664X/82/020109-36$02.00/0

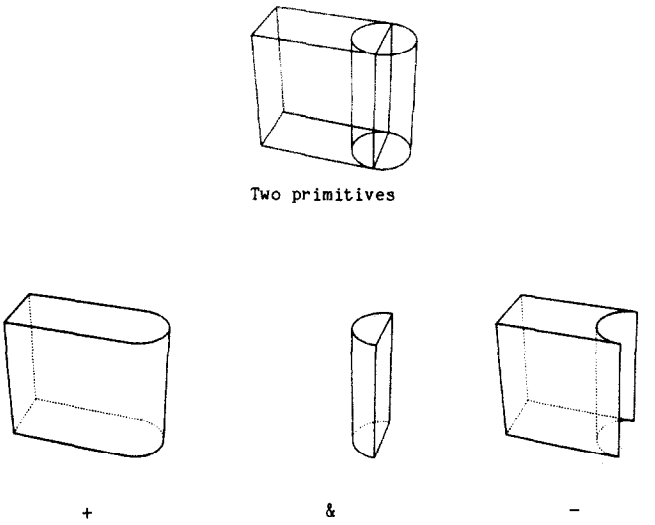


FIG. 1. Example combines of two primitives.

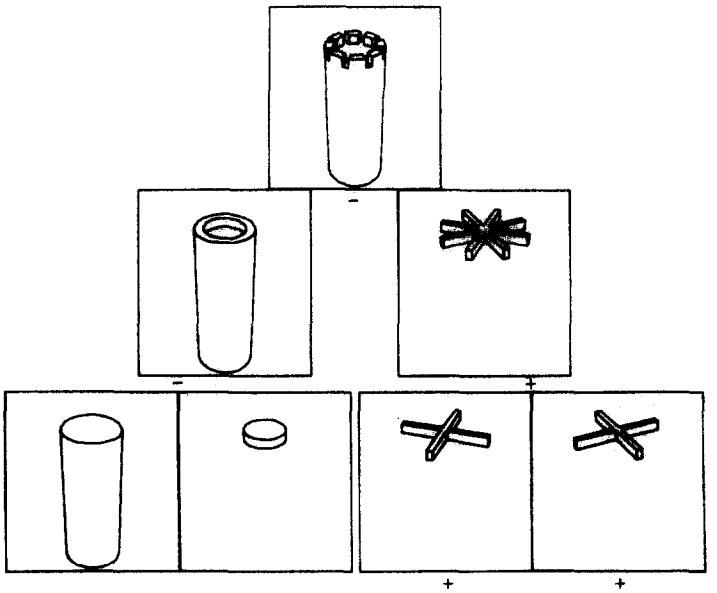


FIG. 2. Example of composition tree.

A basic command system for a solid modeling system, through which the user interacts, is:

CREATE/DELETE
MOVE
COMBINE/UNDO
COPY
PLOT
SAVE/RESTORE

The CREATE command is for creating new instances of primitive solids. Accompanying CREATE the user specifies a primitive type such as block or cone and a name for future reference. DELETE, the counterpart of CREATE, simply deletes a specified primitive or composite solid.

With MOVE the user specifies a solid and the information necessary to scale, translate, or rotate it. This command is the most complicated for the user and has various possible implementations.

With the COMBINE command the user specifies two existing solids, a combine operator (+, &, or −), and the name of the new composite solid. UNDO, given a composite solid, undoes the combine operation leaving two subcompositions in its place.

COPY makes a copy of a solid. With this command, previously modeled solids can be used like primitives or solid macros. That is, a user can copy an existing part, scale it, and move it to a new position. The real set of primitives only determines the types of surfaces available.

PLOT displays a solid. Accompanying this command, a user may be able to specify one or multiple views, to zoom, to see hidden lines removed, etc.

Last of all, SAVE is for storing the current set of solids on disk for later retrieval via RESTORE.

1.2. Ray Casting Origins: MAGI

The idea of ray casting apparently originated with Goldstein and Nagel [4] of MAGI (Mathematics Applications Group, Inc.). They give credit to Walter Guber of MAGI for originally formulating the combinatorial geometry method. In Goldstein's and Nagel's paper, "3-D Visual Simulation," ray casting is called ray tracing and the application is shaded picture generation. To make shaded pictures of solids, they simulated the photographic process in reverse. For each picture element (pixel) in the screen, they cast a light ray through it into the scene to identify the visible surface. The first surface intersected by the ray, found by "tracing" along it, was the visible one. At the ray–surface intersection point found, they computed the surface normal and, knowing the position of the light source, computed the brightness of the pixel in the screen.

Extending this method further, MAGI developed a commercial CAD/CAM system called SynthaVision [9] that is capable of making shaded pictures and line drawings, computing mass properties, and verifying noninterference in N/C machining operations. Unfortunately, it is an expensive batch system.

Many experts in the CAD/CAM field have doubts about the sufficiency of ray casting and consider it to be an impractical, brute force method. In solid modeling as in all CAD/CAM, good man–machine interaction is essential. Efficiency for interactive use and sufficiency for various applications are ray casting issues directly addressed in this paper.

1.3. Overview

Following this introduction, I present ray casting in five sections: 2. Basis: Light Rays and Cameras; 3. Ray Casting Algorithm; 4. Enclosures and Efficiency; 5. Applications; 6. Extensions and Conclusions.

Section 2 defines rays and the camera model. Light rays and the camera or eye geometry form an effective framework for solid modeling. The camera is a model of

the user's viewpoint and rays are the user's probes, lines of sight into a scene of solids. Section 2 defines the information that probing rays return and briefly explains how to use it to make line drawings, shaded pictures, and volume calculations.

Section 3 describes the actual algorithm that "classifies" a ray with respect to a solid, a two-part process: (1) intersect rays with primitives to find the enter–exit points, thereby providing ray–primitive classifications, and (2) combine ray classifications according to the +, &, and − operators. Also in Section 3, the data structure for solids is specified and the subject of coordinate systems is discussed. Various coordinate systems are used, necessitating the transforming of rays between them.

Section 4 discusses efficiency. The simplest implementation of a solid modeling system based on ray casting requires little memory for data and code, but it uses much CPU time. Since the ray casting algorithm is simple and used reiteratively, special computing hardware will probably be available someday. But this Section analyzes the cost factors for the software implementation and describes numerous means, such as box enclosures around solids, to make it faster.

Section 5 describes various applications of ray casting. The first is making line drawings, with emphasis on speed for interactive use. Combining and editing solid compositions are very fast operations, entailing only additions and modifications to the solid data structure. The problem is providing visual feedback to the user, quickly. Ways to quicken line drawing generation are presented and sample pictures with actual run times are shown. The speed is a function of the display resolution and the complexity of the solid. A problem remains, however, that is common to computer graphics: aliasing. Jagged edges can be smoothed, but the faces of isolated, needle thin solids may be overlooked in drawings.

The second application described in Section 5 is a natural one for ray casting: shaded picture generation. The realistic effects of specular reflections, transparencies, shadows, and multiple light sources can all be modeled.

The last application discussed in Section 5 is solid analysis, via conversion to polyhedra. Building a polyhedral representation of a solid, originally defined by a composition tree to have curved surfaces, is fairly easy with ray casting. The user can ask for it when needed and specify the accuracy of approximation. Polyhedra, because of their simplicity, yield to engineering analyses. And because of their wide use, they are a medium to other CAD/CAM or FEA (finite element analysis) systems.

Section 6 describes two unrelated ways for extending ray casting: "on" classification and adding new primitives. The first, "on" classification, is not actually recommended because of its limited utility and inherent complexity. More positively, Section 6 explains that extending the geometric coverage of a ray casting based system is relatively easy. Tori and probably even primitives bounded by parametric surface patches can be modeled.

2. BASIS: LIGHT RAYS AND CAMERAS

Light rays and the camera geometry form the basis for all geometric reasoning here. This includes nonpictorial applications such as mass analysis. The following text defines the camera model, describes the information ray casting provides, and then explains how to apply the information.

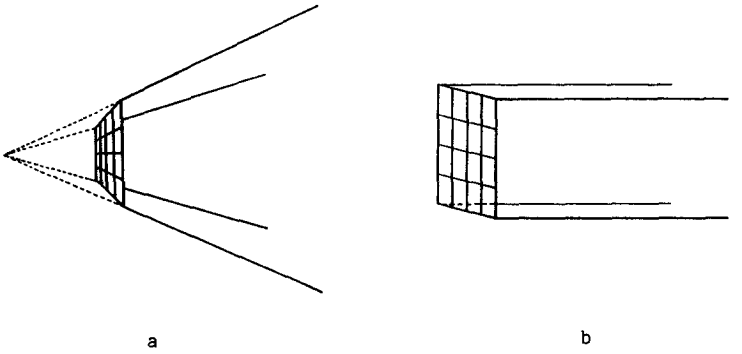


FIG. 3. Camera models: (a) perspective and (b) parallel views.

2.1. *Simple Camera Model*

The pinhole camera model is used, the standard in image processing [10]. This camera model consists of a focal point (or eye point) and a square pixel array (or screen). The visible solids, called the "scene," are in front of the camera—enclosed by the truncated, four-sided pyramid shown in Fig. 3a. Straight light rays pass through the pixel array to connect the focal point with the scene, one ray per pixel. To shade pictures, the rays' intensities are measured and stored as pixels. The reflecting surface responsible for a pixel's value intersects the pixel's ray.

When the focal length, distance between focal point and screen, is infinite, then the view is called "parallel" because all light rays are parallel to each other, perpendicular to the screen (Fig. 3b). Although the perspective view is natural for making pictures, many applications need rays that can be uniformly distributed in space.

For convenience, the standard coordinate system for the camera has the screen in the X–Y plane, the scene in the $+Z$ half space, and the focal point on the $-Z$ axis (Fig. 4).

A ray is simply a straight line in the 3-D space of the camera model. It is best defined in parameterized form as a point ($X0$, $Y0$, $Z0$) and a direction vector (Dx, Dy, Dz). In this form, points on the line are ordered and accessed via a single parameter t. For every value of t, a corresponding point (X, Y, Z) on the line is

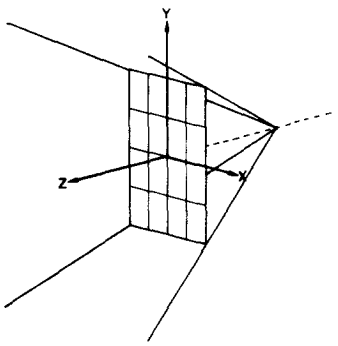


FIG. 4. Camera's local coordinate system.

defined:

$$X = X0 + t * Dx,$$
$$Y = Y0 + t * Dy,$$
$$Z = Z0 + t * Dz.$$

So, a ray in a parallel view that passes through pixel (X, Y) in the screen is simply defined as $(X, Y, 0)(0, 0, 1)$. In a perspective view, the ray is defined as $(X, Y, 0)(X, Y, Ze)$ given the screen center $(0, 0, 0)$ and eye point $(0, 0, -Ze)$.

2.2. *Information from Ray Casting*

RAYCAST is the ray–solid evaluator, the heart of the solid modeling system. Input to RAYCAST is a ray; output from RAYCAST is information about how the ray intersects the scene. Knowing the camera model and the solid in the scene, RAYCAST finds where the given ray enters and exits the solid (Fig. 5). This information is returned in the following two lists:

Ray parameters:	$t[1], t[2], \ldots, t[n]$
Surface pointers:	$S[1], S[2], \ldots, S[n]$

where n is the number of ray–solid intersections. The ordered list of ray parameters, $t[i]$, denote the enter–exit points. The ray enters the solid at point $t[1]$, exits at $t[2]$, enters at $t[3]$, etc., and finally exits at $t[n]$. Point $t[1]$ is closest to the camera and point $t[n]$ is furthest. In association with the ray parameters, RAYCAST also returns a list of pointers to the surfaces through which the ray passes.

2.3. *Applying the Information*

Three algorithms using ray casting—to make line drawings, to make shaded pictures, and to compute volumes—are outlined here. Details are in Section 5. "Applications." Each algorithm, given a camera model, casts one ray per pixel in the

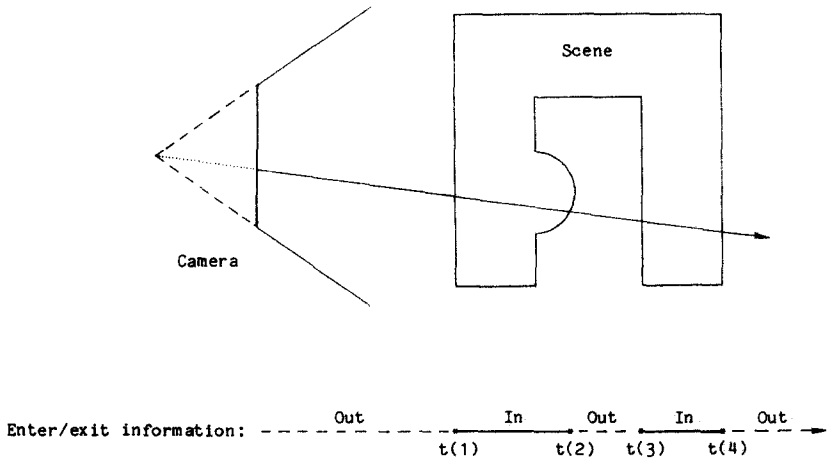


FIG. 5. Example of a cast ray.

screen. For computing volume, the density of pixels in the screen to use depends on the desired accuracy of the solution. For line drawings and picture shading, the density depends on the quality of the image desired and on the resolution of the display device.

LINE DRAWINGS. To draw the visible edges of a solid, generate one ray per pixel moving top–down, left–right in the screen. Evaluate each ray via RAYCAST in order to identify the visible surface $S[1]$, the first surface pointer in the list that RAYCAST returns. If the visible surface at pixel (X, Y) is different than the visible surface at pixel $(X - 1, Y)$, then display a vertical line one pixel long centered at $(X - 0.5, Y)$. Similarly, if the visible surface at pixel (X, Y) is different than the visible surface at pixel $(X, Y - 1)$, then display a horizontal line one pixel long centered at $(X, Y - 0.5)$. The resulting drawing will consist of horizontal and vertical edges only.

SHADED PICTURES. To make a shaded picture, again cast one ray per pixel in the screen. This time, however, use the visible surface pointer $S[1]$ at each pixel to access the description of the surface. From this, compute the surface normal at the visible point $t[1]$. The pixel's value, the displayable light intensity, is proportional to the cosine of the angle formed by the surface normal and light-source-to-surface vector. Processing all pixels this way produces a raster type picture of the scene.

COMPUTING VOLUME. The volume of a solid bounded by curved surfaces is easily computed by the "approximating sums" integration method, by approximating the solid with a set of rectangular parallelepipeds. This is accomplished by taking an "in-depth" picture of the solid in a parallel view. Casting rays through the screen into the solid partitions the solid into volume elements. Two dimensions of the parallelepipeds are constant, defined by the 2-D spacing of rays in the screen. The third dimension is variable, defined by the enter–exit points returned by RAYCAST. Specifically, if the horizontal and vertical distance between rays in the screen is S, then the volume "detected" by each ray is

$$S * S * (t[2] - t[1] + t[4] - t[3] + \cdots + t[n] - t[n-1]) * L,$$

where $L = (Dx * Dx + Dy * Dy + Dz * Dz)^{1/2}$, the length of the ray's direction vector. Each $(t[i] - t[i-1]) * L$ is a length of a ray segment that is inside of the solid.

3. RAY CASTING ALGORITHM

RAYCAST, the ray–solid evaluator, finds where a given ray enters and exits a given solid, accounting for the combine operators +, &, and −. Before presenting the mechanics of RAYCAST, I first describe the data structures for solid compositions. Then I describe the coordinate systems that RAYCAST uses and explain how rays are transformed from one coordinate system to another. Finally, the RAYCAST algorithm is presented in abstract procedural form, followed by details.

3.1. Composition Data Structure

The data structure representing solid compositions is an inverted, binary tree (Fig. 2). Leaf nodes of the tree are PRIMITIVEs and the internal nodes are COMPOSITEs, solids formed by the combine operators +, &, and −. If a composition has N PRIMITIVE solids, then there are $N - 1$ COMPOSITE solids

TABLE 1
Solid Data Structures

```
COMPOSITE:
  NAME
  OPERATION
  L_SOLID_PTR
  R_SOLID_PTR

PRIMITIVE:
  NAME
  TYPE
  LOCAL_TO_SCENE_TRANSFORM
  SCENE_TO_LOCAL_TRANSFORM
  SURFACE_PTRS
```

for a total of $2 * N - 1$ solids. This relation holds regardless of whether the tree is balanced. The minimum COMPOSITE and PRIMITIVE data structures are shown in Table 1.

All solids in the tree, both COMPOSITEs and PRIMITIVEs, have a NAME attribute. It is simply a string identifier that the user provides when he creates the solid.

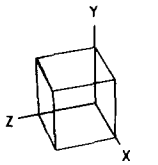
The OPERATION attribute of a COMPOSITE specifies +, &, or −. L_SOLID_PTR and R_SOLID_PTR are pointers to solid nodes, PRIMITIVEs or COMPOSITEs, designating the left and right subtrees.

The TYPE attribute of a PRIMITIVE specifies block, cylinder, cone, sphere, torus, etc. LOCAL_TO_SCENE_TRANSFORM is a 4×4 linear transformation matrix and SCENE_TO_LOCAL_TRANSFORM is its inverse. The LOCAL_TO_SCENE_TRANSFORM transforms points from the local coordinate system of the primitive solid to the scene coordinate system. These transformations are explained next in Section 3.2. Last of all, SURFACE_PTRS point to surface data structures. They are not defined here because no surface attributes are required minimally.

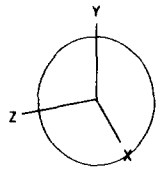
3.2. Coordinate Systems

RAYCAST uses three kinds of coordinate systems, all cartesian: scene (or global) coordinate system; primitives' local coordinate systems; screen coordinate system. The user should be aware of only the global coordinate system. In it he moves solids around, positions the camera, and sees the X, Y, Z axes displayed with the solids. If only one coordinate system could be used, this would be it. The others are used in order to simplify geometric calculations with the camera model or with the primitives.

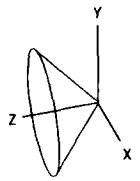
Each primitive type has a local coordinate system. When a user creates a new primitive solid, its position and scale in the global coordinate system are the same as those in the local coordinate system. When the user moves or scales it, however, its position or size changes in the global coordinate system. The scene-to-local transform records the transformation needed to get it back to the local coordinate system. The local coordinate systems of primitives are shown in Fig. 6.



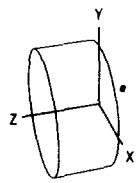
The block is a unit cube in the positive octant with one vertex at the origin.



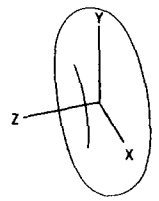
The sphere is centered at the origin with a unit radius.



The cone's axis lies along the +Z axis with its vertex at the origin. Radius and length are 1.



The cylinder's axis lies along the +Z axis with one end at the origin. Radius and length are 1.



The torus is centered at the origin with its major circle in the X-Y plane. Its major radius is 1 and its minor radius is variable, between 0 and 1.

FIG. 6. Local coordinate systems of primitives.

In the screen coordinate system, the screen is in the X–Y plane. That is, the screen's plane equation is $Z = 0$. The scene is in the $+Z$ half space and the eye point is in the $-Z$ half space, displaced from the center of the screen (Fig. 4). The coordinate systems of the screen and scene are linked via a "screen-to-scene" transform and its inverse "scene-to-screen" transform, as defined by the camera model. Each primitive has its scene-to-local transform and inverse, but there is only one screen-to-scene transform and inverse. Given a ray originating in the screen coordinate system, RAYCAST transforms it into the local coordinate systems of primitives via the scene coordinate system in order to find the ray–solid intersection points.

Using the homogeneous coordinate techniques common to computer graphics and geometric modeling, linear transformations between coordinate systems are well defined [11]. Only point and vector transformations are important for ray casting. The syntax I use for a point transformation is

$$P' = P*M$$

or

$$(X', Y', Z', 1) = (X, Y, Z, 1) * M,$$

where $P = (X, Y, Z)$ is the given point, M is a 4×4 linear transformation matrix, and $P' = (X', Y', Z')$ is the transformed point that results.

Similarly, the transformation of a vector $V = (Dx, Dy, Dz)$ is

$$V' = V * M$$

or

$$(Dx', Dy', Dz', 0) = (Dx, Dy, Dz, 0) * M,$$

where $V' = (Dx', Dy', Dz')$ is the transformed vector that results.

A line (i.e., ray) is transformed by simply transforming its fixed point and direction vector:

$$\begin{aligned}(X0, Y0, Z0)(Dx, Dy, Dz) * M &= ((X0, Y0, Z0, 1) * M)((Dx, Dy, Dz, 0) * M) \\ &= (X0', Y0', Z0')(Dx', Dy', Dz')\end{aligned}$$

When transformed this way, the line parameterizations are the same in both coordinate systems. That is, $P'(t) = P[t] * M$ for all t, where $P[t]$ is a point on the original line corresponding to parameter t and $P'(t)$ is a point corresponding to the same t on the transformed line. What practical significance does parameter independence have? RAYCAST does not actually find the points at which rays enter and exit solids. Rather, it finds the ray parameters that designate those points. So, only rays need to be transformed between coordinate systems, not parameters or points.

Solids or surface equations are never explicitly transformed. Primitives' scene-to-local transforms are only used to transform rays. The effect of scaling, rotating, and translating a solid is achieved by scaling, rotating, and translating the rays. Consequently, finding ray–solid intersections is very easy. Arbitrary elliptic cylinders in the scene are all the same solid in their local coordinate system, a right circular cylinder at the origin with unit radius and length. Similarly, ellipsoids are all the same unit sphere, elliptic cones are all the same right circular cone with unit radius and unit length, and parallelepipeds are all the same unit block.

3.3. In–Out Classification

Given a ray and a solid composition tree, RAYCAST classifies the ray with respect to the solid and returns the classification to the caller. By definition, the classification of a ray with respect to a solid is the information describing the ray-solid intersection. It designates what parts of the ray are in the solid versus out of the solid. RAYCAST starts at the top of the solid composition tree, recursively descends to the bottom, classifies the ray with respect to the primitive solids, and then returns up the tree combining the classifications of the left and right subtrees (Fig. 7).

The procedural form of RAYCAST, written in pseudo-PL1, is presented below. Assume that the application program that calls RAYCAST has already transformed the screen ray into the scene coordinate system and it is globally available to RAYCAST. Then the procedure call, RAYCAST(GIVEN_SOLID_PTR), returns to

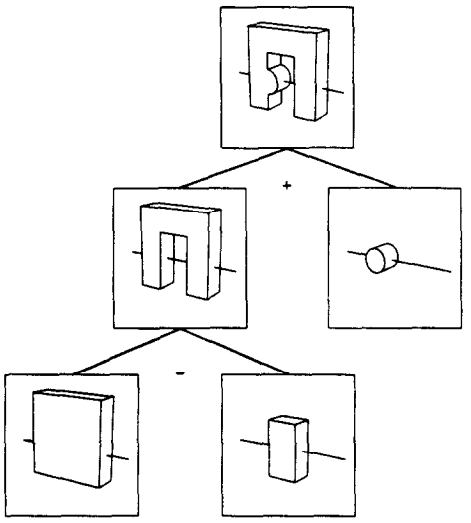


FIG. 7. Sample ray and composition tree.

the application program the classification of the ray with respect to the root solid that GIVEN_SOLID_PTR points to. Upper case text is either programming language syntax or references to previously defined data structures. Comments are quoted.

```
RAYCAST: PROCEDURE (SOLID_PTR) RETURNS (Classification);
IF (SOLID_PTR points to a COMPOSITE solid)
  THEN
    DO "Combine";
    L_classification = RAYCAST(SOLID_PTR → COMPOSITE.L_SOLID_PTR);
    R_classification = RAYCAST(SOLID_PTR → COMPOSITE.R_SOLID_PTR);
    RETURN (Combine (L_classification, R_classification));
    END "Combine";
  ELSE
    DO "Primitive";
    Transform the scene ray to the primitive's local coordinate system
    via the primitive's SCENE_TO_LOCAL_TRANSFORM;
    DO CASE (SOLID_EP → PRIMITIVE.TYPE);
      BLOCK:     Do 6 ray-plane intersection tests;
      SPHERE:    Do 1 ray-quadric intersection test;
      CYLINDER:  Do 2 ray-plane and 1 ray-quadric intersection tests;
      CONE:      Do 1 ray-plane and 1 ray-quadric intersection tests;
      TORUS:     Do 1 ray-quartic intersection test;
      END "Do by case";
    RETURN (Primitive_classification);
    END "Primitive";
END "Classify ray";
```

Input to RAYCAST are a ray equation and a solid pointer; output from RAYCAST is the ray–solid classification: an integer N specifying the number of ray–solid intersections points, a real array PARMS[1:N] of ray parameters that specify the intersection points, an associated pointer array THRU_SURFS[1:N] specifying the surfaces through which the ray passes, and a binary array CLASSIFS[1:N + 1] specifying the in–out classifications of the N + 1 ray segments defined by the N intersection points.

The following text describes in greater detail the two essential parts of the algorithm: (1) intersecting rays with primitives and (2) combining left and right classifications.

INTERSECTING RAYS WITH PRIMITIVES. The general ray–solid intersection problem reduces to ray–primitive intersection problems because rays enter and exit solids via the solids' surfaces and all surfaces originate with primitives.

For convex primitives such as blocks, cylinders, cones, and spheres, the ray–primitive intersection test has four possible outcomes:

1. The ray missing the primitive;
2. The ray touches (i.e., is tangent to) the primitive at one point;
3. The ray enters and exits the primitive at two different points;
4. The ray lies on a face of the primitive.

In cases 1 and 2, the ray is classified as completely out and so no intersection points are noted. In cases 3 and 4, two intersection points are noted and the ray is divided into three segments: out–in–out. Treating "on" as "in," case 4, has subtle implications as explained in Section 6.1.

Tori are different. A ray may be tangent to a torus at one or two points, and may intersect it at as many as four points. So, with respect to a torus, a ray classifies as out, out–in–out, or out–in–out–in–out.

The computation for finding ray–primitive intersections, given that the ray has been transformed into the primitive's local coordinate system, is:

```
FOR EACH Surface of primitive
  DO;
    Simultaneously solve the ray and surface equations;
    IF (Intersection points found are within the primitive's bounds)
      THEN Note the ray parameters, disregarding duplicates;
  END;
```

Table 2 lists the surface equations and bounds for various primitives in their canonical form, the form in their local coordinate system.

The ray–plane intersection calculation is simple. For instance, to intersect the parameterized ray ($X0$, $Y0$, $Z0$)(Dx, Dy, Dz) with the Y–Z plane, simultaneously solve $X = 0$ and $X = X0 + t * Dx$ for t. The resulting parameter $t = -X0/Dx$ defines the point of intersection $(0, Y0 + t * Dy, Z0 + t * Dz)$. If this point lies within the bounds of the primitive then it is a good ray–solid intersection point. The bounds test for this point on the Y–Z plane of a block is

$$(0 \leq (Y0 + t * Dy) \leq 1) \quad \text{AND} \quad (0 \leq (Z0 + t * Dz) \leq 1).$$

Finding ray–quadric intersection points is similar. Simultaneously solving a ray's line equation with the quadric surface equation yields a 2nd order polynomial in t,

TABLE 2
Surface Equations and Bounds per Primitive

Primitive	Surface equations	Bounds test
Block	$X = 0$	$0 \le Y, Z \le 1$
	$X = 1$	$0 \le Y, Z \le 1$
	$Y = 0$	$0 \le X, Z \le 1$
	$Y = 1$	$0 \le X, Z \le 1$
	$Z = 0$	$0 \le X, Y \le 1$
	$Z = 1$	$0 \le X, Y \le 1$
Cylinder	$Z = 0$	$X*X + Y*Y \le 1$
	$Z = 1$	$X*X + Y*Y \le 1$
	$X*X + Y*Y = 1$	$0 \le Z \le 1$
Cone	$Z = 0$	$X*X + Y*Y \le 1$
	$X*X + Y*Y - Z*Z = 0$	$0 \le Z \le 1$
Sphere	$X*X + Y*Y + Z*Z = 1$	none
Torus	$(X*X + Y*Y + Z*Z + 1 - r*r)^2$ $-4*(X*X + Y*Y) = 0$	none

which is appropriate for the quadratic formula. Finding ray–torus (i.e., ray–quartic) intersection points is more complicated. Simultaneously solving the ray and torus equations yields a 4th order polynomial in t. The problem of finding the roots of such polynomials has a closed form solution [12].

COMBINING LEFT AND RIGHT CLASSIFICATIONS. Figure 8 illustrates how to combine ray classifications from the left and right subtrees for each of the

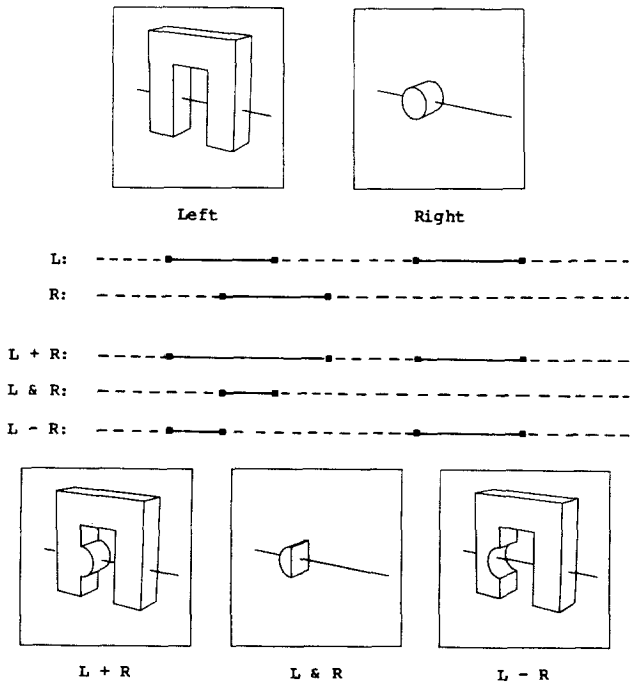


FIG. 8. Combining ray classifications.

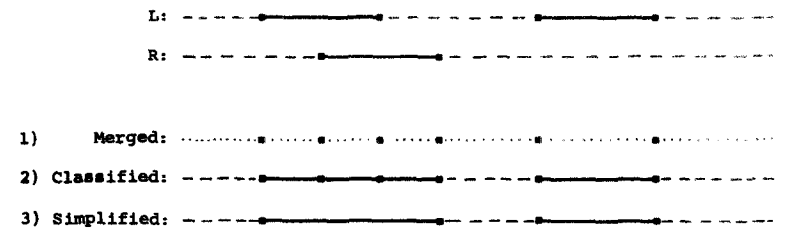


FIG. 9. Example of the Three Step Combine Process.

operators +, &, and −. Solid lines represent ray segments inside of the solid and dashed lines represent ray segments outside of the solid.

Consider the combine operation to be a three step process. (1) The intersection points from the left and right rays are merged in sorted order, forming a segmented composite ray. (2) The segments of the composite ray are classified in or out depending on the combine operator and the classifications of the left and right rays along those segments. (3) The composite ray is simplified by merging contiguous segments with the same classification. Figure 9 illustrates the three steps for the + operator shown in Fig. 8.

Combining classifications is simply boolean algebra. But instead of the usual complement operator, difference is used. See Table 3 for the rules.

4. ENCLOSURES AND EFFICIENCY

Ray casting qualifies as a brute force method for solving problems. The minimal RAYCAST algorithm is very simple, particularly in consideration of its many applications and ease of use. But most applications call RAYCAST in their inner loop—an inner loop that is executed thousands or hundreds of thousands of times.

By introducing additional logic and refinements to the minimal RAYCAST algorithm, it will run much faster. In this section I first explain that the complexity of the scene, measured by the number of primitives in the solid composition, is related to RAYCAST's CPU usage. Then I explain how the use of simple box enclosures around solids speeds up the search for ray–solid intersections. Last of all,

TABLE 3
Boolean Operations

Operator	Left	Right	Composite
+	IN	IN	IN
	IN	OUT	IN
	OUT	IN	IN
	OUT	OUT	OUT
&	IN	IN	IN
	IN	OUT	OUT
	OUT	IN	OUT
	OUT	OUT	OUT
−	IN	IN	OUT
	IN	OUT	IN
	OUT	IN	OUT
	OUT	OUT	OUT

I describe various refinements to the algorithm, short of implementing it in assembly language, which improve execution time by about a factor of 2.

4.1. CPU Usage vs Scene Complexity

For the minimal RAYCAST algorithm presented in Section 3, memory and CPU usage is directly proportional to the scene complexity—that is, to the number of primitive solids in the composition. Memory usage is relatively small, so it is not a concern. The important question is "How fast is it?"

To appreciate the cost of using RAYCAST, consider the following scenario. A user has built a model of a complex solid using 300 primitives and asks his picture shading program to display the solid on a raster display. Assume the shading program uses ray casting in a simple, straightforward way. Since the raster display has 500 × 500 pixels, the shading program generates 250 000 rays and passes them one at time to RAYCAST for in–out classification. Since the solid is composed of 300 primitives, the binary composition tree also has 300 (actually 299) composite solids making a total of 600 solids.

Each time RAYCAST is called, it visits each node in the tree via recursion—that is, via about 600 procedure calls to itself:

$$\text{Cost \#1: } 600 * 250\,000 = 150\,000\,000 \text{ recursive procedure calls.}$$

At each composite solid in the tree, RAYCAST combines the ray classifications from the left and right subtrees:

$$\text{Cost \#2: } 300 * 250\,000 = 75\,000\,000 \text{ classification combines.}$$

Actually, there is a hidden combinatoric factor in the combine operation. The cost grows faster than linear, but it will probably never dominate in practice—at least not with compositions of 1000 primitives or less. When two ray classifications are combined, the two ray–solid intersection lists are merged in sorted order. The computation is simple, but it is performed at every composite node in the tree, so the worst case cost is N^2 for a composition with N primitives. To be N^2, the ray must intersect every primitive, however, and the tree must be completely unbalanced (i.e., be a list). The following discussion assumes the combine cost is linear, like the other costs.

For each primitive solid, RAYCAST transforms the ray into the local coordinate system of the primitive for the ray–solid intersection test:

$$\text{Cost \#3: } 300 * 250\,000 = 75\,000\,000 \text{ ray transformations.}$$

Last of all, each ray–solid intersection test involves one or more ray–surface intersection tests. Assuming that the solid's primitives have an average of four surfaces each, every call to RAYCAST involves 300*4 ray–surface intersection tests:

$$\text{Cost \#4: } 1200 * 250\,000 = 300\,000\,000 \text{ ray–surface intersection tests.}$$

So the total cost for generating the shaded picture of the user's solid is the sum of the costs:

#1: 150 000 000 recursive procedure calls.
#2: 75 000 000 classification combines.
#3: 75 000 000 ray transformations.
#4: 300 000 000 ray–surface intersection tests.

Although this might be a good place to hope for tomorrow's faster computers or to seek special hardware for this simple, highly repetitive task, there are easy ways to significantly reduce the cost. The next section on enclosures shows how the multiplier 300 in the cost figures can be significantly reduced for almost all compositions. And, of course, not all applications need to cast as many rays as 250 000.

4.2. Box Enclosures

By using minimum bounding boxes around the solids in the composition tree, the exhaustive search for a ray–solid intersection resembles an efficient binary search. The minimal RAYCAST algorithm presented does an exhaustive search because it always visits all of the nodes in the tree—transforming the ray into primitives' local coordinate systems, testing for ray-surface intersections, and combining classifications—even when the ray clearly misses the solid. In order to detect a "clear miss," RAYCAST must use the binary composition tree as a hierarchical representation of the space that the solid composition occupies. But all position, shape, and size information is stored at the leaves of the tree where the primitive solids are. The top and intermediate nodes in the tree only specify combine operators. Characterizing with enclosures the space that all solids fill gives all nodes in the tree an abstract summary of position and size information. Then, quick "ray intersects enclosure" tests guide the search in the hierarchy. When the test fails at an intermediate node in the tree, the ray is guaranteed to classify as out of the composite, so recursing down its subtrees to further investigate is unnecessary.

The following text defines the box enclosure, the "ray intersects box" text, and the algorithms for creating the box enclosures. Then, how enclosure directed ray classification is affected by the spatial distributions of solid's primitives and the organization of the composition tree is analyzed. Last of all, spheres are presented as an enclosure for special applications.

DEFINITION OF BOX ENCLOSURES. Box enclosures are simply orthogonal rectangular parallelepipeds in the screen coordinate system (Fig. 10). They are in the screen coordinate system because rays originate there and they are constrained by the camera model to pass through the screen. A box is defined by six numbers, the minimum and maximum X, Y, Z's: MIN_POINT[1:3] and MAX_POINT[1:3]. This enclosure information is stored in each solid, both primitives and composites.

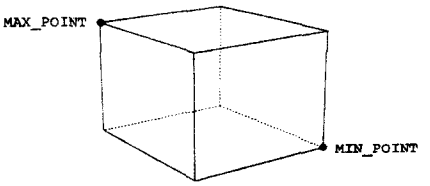


FIG. 10. Box enclosure.

The "ray intersects box" test is basically a "point in rectangle test." Given a ray passing through pixel (X, Y) in the screen, the test is simply

$$\text{MIN\_POINT[1]} \leq \text{X} \leq \text{MAX\_POINT[1]},$$

$$\text{and MIN\_POINT[2]} \leq \text{Y} \leq \text{MAX\_POINT[2]}.$$

The test is basically 2-D because rays usually start at the screen and infinitely extend into the scene. When rays are bounded in depth, as they are for certain applications, the condition

$$\text{MIN\_POINT[3]} \leq \text{RAY\_DEPTH}$$

is added to the test. "RAY_DEPTH" would then be an additional parameter to RAYCAST calls.

To create box enclosures around solids, RAYCAST needs an initialization procedure. The enclosures are view dependent, of course, so must be recreated whenever the camera model changes or the composition changes. The control structure for the algorithm that creates enclosures is recursive like RAYCAST (see Section 3.3). Starting at the composition tree's root node, the algorithm recursively descends to the bottom, encloses the primitives, and then returns up the tree combining the enclosures of the left and right subtrees.

The following four steps compute a box enclosure around a primitive in the screen coordinate system:

1. List the vertices of a polyhedron that tightly encloses the primitive in its local coordinate system. For a block primitive, simply use the block's vertices: $(0, 0, 0)$, $(0, 0, 1)$, $(0, 1, 0)$, etc. For a cone, use its apex $(0, 0, 0)$ and the vertices of the square that encloses its one circle: $(1, 1, 1)$, $(1, -1, 1)$, $(-1, 1, 1)$, and $(-1, -1, 1)$. Etc. for the other primitives.
2. Transform the polyhedron's vertices from the primitive's local coordinate system to the screen coordinate system.
3. Project the transformed vertices to the screen. If the view is parallel, (X, Y, Z) simply projects to $(X, Y, 0)$. Otherwise, (X, Y, Z) projects in perspective to $(X*Ze/(Ze + Z), Y*Ze/(Ze + Z), 0)$ given the eye point $(0, 0, -Ze)$.
4. Find the minimum and maximum values of the projected X, Y's and of the unprojected Z's. These define the enclosing box.

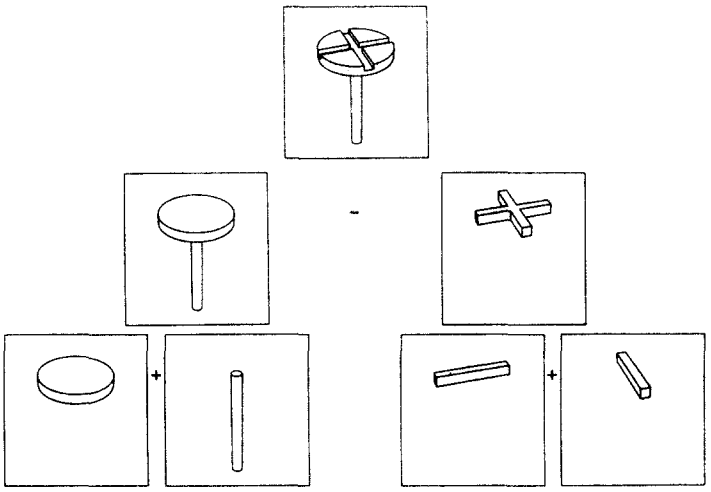
The following procedure combines enclosures at composite nodes in the tree. Note that a composite's enclosure may be smaller than its subcompositions.

```
For I = 1,2,3;
  DO CASE (Operator);
    '+': MIN_POINT[I] = MIN(L_MIN_POINT[I], R_MIN_POINT[I]);
         MAX_POINT[I] = MAX(L_MAX_POINT[I], R_MAX_POINT[I]);
    '&': MIN_POINT[I] = MAX(L_MIN_POINT[I], R_MIN_POINT[I]);
         MAX_POINT[I] = MIN(L_MAX_POINT[I], R_MAX_POINT[I]);
    '-': MIN_POINT[I] = L_MIN_POINT[I];
         MAX_POINT[I] = L_MAX_POINT[I];
  END "Do by case";
END;
```

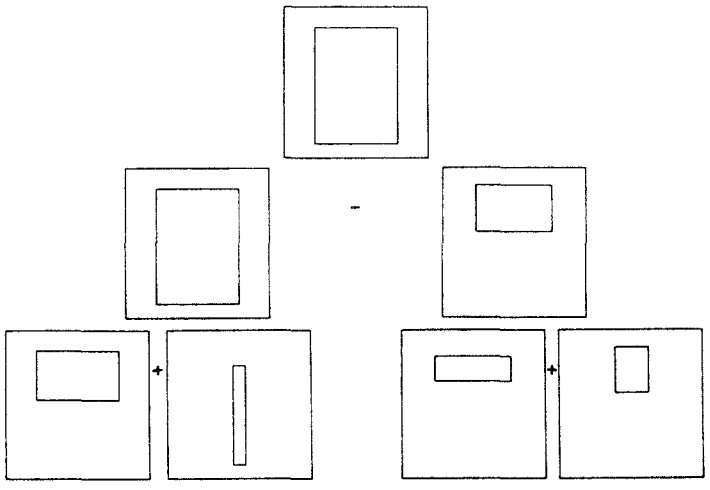
Unlike the + and & operators, − does not obey the usual rules of algebra. The enclosure of the composite $A - B$ is equal to the enclosure of A, regardless of B. To do better, the actual space occupied by the solids would have to be analyzed—a complication not likely to pay for itself.

Figures 11 and 12 illustrate box enclosures. Figure 11 shows how the tree of enclosures corresponds to the composition tree. Figure 12 shows the enclosures for a more complicated solid, a crankshaft. Myrl Thompson of GM's Oldsmobile Division used over 100 primitives to model it. Try to imagine the tree organization for the composite enclosures.

SPATIAL DISTRIBUTIONS AND TREE ORGANIZATION. Accurately assessing the cost savings for using enclosures is difficult because it depends on the

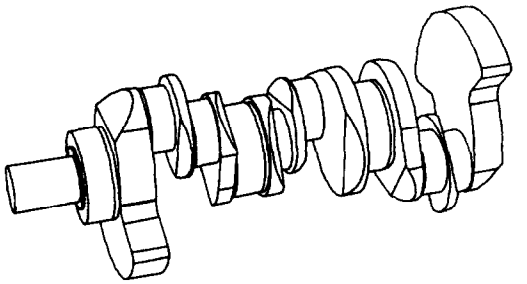


Solid composition tree

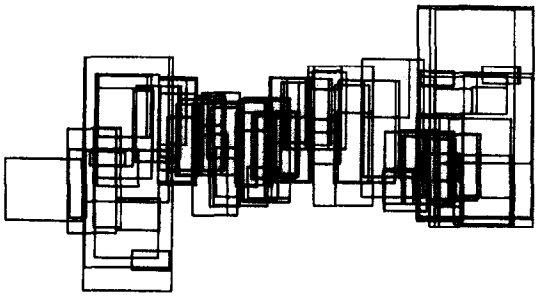


Tree of box enclosures

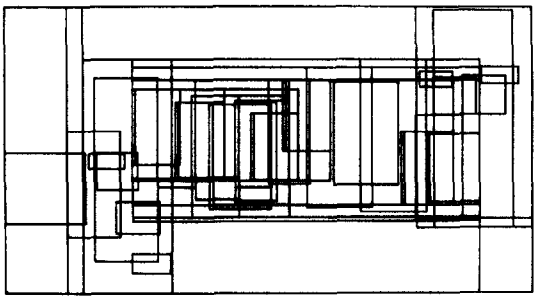
FIG. 11. Box enclosures for solid composition.



Crankshaft model



Primitive enclosures



Composite enclosures

FIG. 12. Enclosures for a crankshaft model.

spatial distribution of the primitives (the complexity distribution) and on the organization of the composition tree. The optimal conditions are

—no primitive enclosures overlap in space;
—composition tree is balanced and organized so that subsolids near in space are also nearby in the tree.

In contrast, the worst condition is

—all primitive enclosures mutually overlap.

In this worst case, the use of enclosures and the tree organization are both irrelevant to efficiency. As an example, consider Fig. 13 where multiple instances of a cylinder

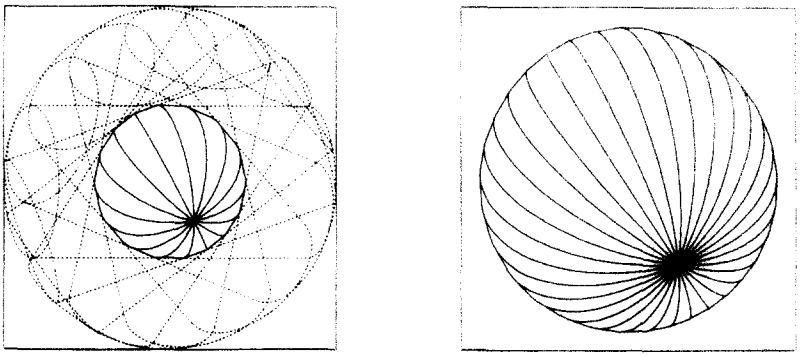


FIG. 13. Intersecting cylinders approximate a sphere.

are rotated around a fixed axis perpendicular to the cylinder's axis and then intersected to approximate a sphere. The uselessness of enclosures in this case should be obvious. Every ray which intersects the composition will intersect every primitive in the tree.

The optimal conditions are more realistic than the worst for two reasons. (1) Why would hundreds of primitives in a complex solid mutually overlap, except to approximate a different surface type? Adding a new primitive to the system with the different surface type is much more practical. Enclosures may largely overlap, however, even though the primitives do not. The primitives would have to be needle thin and diagonally oriented in length, e.g., from $(0,0,0)$ to $(1,1,1)$, and closely packed together. Arrangements like this are possible, but definitely unusual. (2) Users will often build a complex solid in parts or subassemblies and then union them together. The resulting composition tree will be an efficient and natural decomposition of the solid in space. Even then, I think automatically reorganizing the tree of enclosures to optimize the search space is possible.

Given that RAYCAST uses enclosures, how does the spatial distribution of primitives and the tree organization affect cost? Look back to the cost figures in Section 4.1. Costs #1 and #2 depend on the number of nodes in the tree that RAYCAST must visit in order to classify a ray. Costs #3 and #4 depend on the number of primitives that pass the "ray intersects box" test. If all primitives pass the test, as all the primitives would in the sphere approximation shown above, then costs #3 and #4 are maximum and all the nodes in the tree must be visited to reach them, so costs #1 and #2 are also maximum. Consequently, all four costs depend on the spatial distribution of primitives. The only costs dependent on tree organization, however, are #1 and #2. Given a reasonable distribution of primitives in space, RAYCAST will visit fewer nodes in a well-organized tree than in a badly organized tree. A badly organized tree is deep and narrow, like a list, linking the primitives in random order. Again, however, such trees do not affect the number of ray–primitive intersection tests. So, in summary, the spatial distribution of primitives is a more important efficiency factor than tree organization.

The cost factor for ray casting ranges from 0 to N, where N is the number of primitives in the composition, as explained in Section 4.1 and above. The average

cost factor cannot be derived, because it depends on the average complexity distribution of all objects to be modeled, which is unknown. Consider a simple case, however, that can be analyzed. If the enclosures of N primitives are disjoint and uniformly distributed in a sphere, then the average ray intersects less than $C * N^{1/3}$ enclosures where C is the constant $(6/\pi)^{1/3}$. When rays are bounded in depth, then the number of ray–primitive intersection tests is less because RAYCAST will avoid the deeper solids in the scene. And when the depth bound is defined to be the depth of a nearby (visible) solid, then the number of intersection tests is 1. Many rays are bounded this way by the algorithms described in Section 5.1.

SPHERES. A few applications generate rays that are unconstrained by a camera model. For example, specularly reflected rays originate anywhere in the scene, not on the screen, and their orientation depends on arbitrary surface normals (see Section 5.2. "Realistic Shaded Pictures"). For unconstrained rays, spheres instead of boxes must be used as solid enclosures. The "ray intersects sphere" test reduces to a point-to-line distance computation. By comparing the square of this distance with the square of the sphere's radius, no square root is necessary. Spheres are more general than orthogonal boxes, but boxes are tighter enclosures than spheres in almost all cases and the "ray intersects box" test is faster; so spheres should only be used when the rays are not constrained by the camera geometry. For a detailed analysis of the use of spheres as enclosures, see Levine's thesis [13].

4.3. Notes on Optimizing

EARLY OUTS. If the operator at a composite node in the tree is $-$ or & and the ray classifies as out of the composite's left subsolid, then the ray will classify as out of the composite regardless of the ray's classification with respect to the right subsolid. So, classifying the ray with respect to the right subsolid is unnecessary and should be avoided for efficiency. This amounts to a single PL1 statement:

```
IF (operator ≠ +) AND (L_classification = out)
  THEN RETURN(out);
```

It should be inserted in the RAYCAST procedure outlined in Section 3.3. between the two statements:

```
L_classification = RAYCAST (SOLID_PTR → COMPOSITE.L_SOLID_PTR);
R_classification = RAYCAST (SOLID_PTR → COMPOSITE.R_SOLID_PTR);
```

TRANSFORMATIONS. By initially combining the screen-to-scene transform with the primitives' scene-to-local transforms and storing the resultant screen-to-local transforms in the primitives' data structures, one ray transform per call to RAYCAST is eliminated. Like enclosures, the screen-to-local transforms must be reinitialized whenever the camera model or composition changes.

Fast ray transformations are easy to implement. For every 4×4 transformation matrix M, $M[1,4] = M[2,4] = M[3,4] = 0$ and $M[4,4] = 1$. Multiplication by 0 and 1 and additions of 0 are needless. Also, rays constrained by the camera model originate at $Z = 0$, further simplifying the computation. In fact, transforming a general ray (regardless of view) requires only 15 multiplications and 12 additions.

RECURSION. Given a deep composition tree, allocating the arrays PARMS, THRU_SURFS, and CLASSIFS with each recursion can be expensive and should be avoided. These arrays were introduced in Section 3.3 as parameters to RAYCAST calls, output receptacles describing the ray classification with respect to the root solid. But they can also be used as working space for classifying the composition's subsolids. To be used this way, the arrays must be allocated by the application program that calls RAYCAST to have lengths equal to three times the number of primitives in the composition. When RAYCAST recurses to classify an intermediate node, it must pass an index into the arrays designating the starting point for storing the classification.

In my final implementation, I eliminated all internal procedures by making RAYCAST one large procedure. Macros helped. Further, I simulated recursion by using static arrays as stacks. The stack size must be at least as large as the number of primitives in the composition.

5. APPLICATIONS

The central geometric modeling activities are build and edit, made possible by a command system and pictorial display. Since these are interactive activities, speed is a critical issue. Consequently, this section begins by examining the problem of generating line drawings using ray casting.

The second part of this section explains how to make shaded pictures on a raster display, a natural application of ray casting. Many realistic effects such as specular reflections, transparencies, shadows, and multiple light sources can be easily modeled.

The last application discussed is solid analysis. Mass properties of a solid are easy to compute via numerical integration based on ray casting. The volume elements in the integration are parallelepipeds. To extend this analysis capability, the general problem of conversion to polyhedra is examined because they are simpler to analyze and many CAD/CAM and FEA systems accept polyhedra or polygonal surface models as input.

In summary, this section describes the following ray casting applications:

—Line Drawings for Interactive Modeling: speed; antialiasing; poke-at-it interaction; detecting occluded edges for display; detecting "nondisplayable" edges for removal.
—Realistic Shaded Pictures: specular reflections; transparencies; shadows; multiple light sources; area light sources.
—Solid Analysis: mass properties; conversion to polyhedra.

5.1. Line Drawings for Interactive Modeling

The simple line drawing algorithm presented in Section 2.3 indeed works, but it has some practical deficiencies. First, consider the advantages of using ray casting to make line drawings of solid compositions:

—Formulating and then parameterizing and classifying the curved edges created by the intersections of surfaces—a very complicated task—is unnecessary!
—Hidden lines, i.e., occluded edges, are effortlessly removed.
—Silhouettes of curved surfaces are a by-product, so need not be explicitly created whenever the view changes.

Two drawbacks of the algorithm are:

—Speed: the algorithm is CPU bound;
—Aliasing: edges are jagged and surface "slivers" may be overlooked.

Speed is particularly important; pictorial display provides essential feedback to the user when interactively composing solid models. Aliasing, a less serious problem, is common to raster type computer graphics. After addressing the speed and aliasing problems below, I discuss some general issues regarding line drawings and display interaction.

SPEED. Quickly drawing pictures is very important for interactive use. So additional complexity here should be acceptable. The minimal line drawing algorithm is slow, even using the improved version of RAYCAST with box enclosures. Below I present three separate ideas for speeding up the task, which can be used independently or together: dynamic bounding, coherence, and updating. Dynamic bounding is a refinement to the RAYCAST routine for more quickly identifying visible surfaces. Via coherence and updating, rays are selectively used, so fewer are cast.

Dynamic Bounding. If only the visible edges of the solid are to be displayed, RAYCAST can dynamically bound the ray to cut off the search. That is, once RAYCAST finds that the ray intersects a subsolid, RAYCAST can use the intersection point closest to the screen to tighten the depth bound for the "ray intersects box" test. To implement this, the following statement should be executed by RAYCAST on exit from all recursive calls:

IF ($N > 0$)
  THEN RAY_DEPTH = MIN(RAY_DEPTH, PARMS[1]$*Dz$),

where N is the number of ray–solid intersection points, PARMS[1] is the first ray parameter, and the ray is in general form $(X, Y, 0)(Dx, Dy, Dz)$.

Unfortunately, this only works for the + part of the tree, at the top. With − and &, nearby "in" parts of the ray may later become "out." Figure 14 illustrates which nodes in the tree are valid (V) for recomputing RAY_DEPTH on exit. These solids should be flagged by the initialization process which forms enclosures. Obviously, the tree organization will affect the cost savings for using dynamic bounding, and there will be savings only for scenes with two or more solids in depth from the screen. For such scenes, the savings will be significant even if the tree is randomly

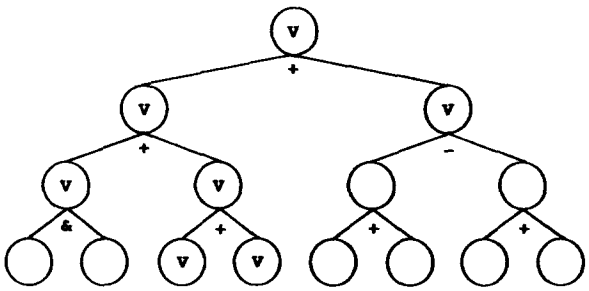


FIG. 14. Valid tree nodes for tightening ray bound.

organized. To improve tree organization: the − operators should be pushed down the tree as far as possible; at + nodes, the solid closest to the screen should be chosen for the left.

Coherence. The principal of coherence is that the surfaces visible at two neighboring pixels are more likely to be the same than different. Developers of computer graphics and vision systems have applied this empirical truth for efficiency and performance. But I do not know if it has been applied to a ray casting based system. If in fact the image area containing edges is much less than the total image area and only the edges are sought, then ray casting should be concentrated around the edges and not in the open regions. This can be effectively implemented by sparsely sampling the screen with rays and then locating, when neighboring rays identify different visible surfaces, the edges via binary searches. The rays used in each binary search can be bounded in depth, the depth being the minimum of the maximum depths of the two surfaces' primitives.

This procedure will significantly improve the speed of drawing all pictures, depending of course on the sampling rate. This should be under user control. As the sampling becomes sparser, the maximum width of isolated solid faces (slivers) that may be overlooked becomes larger. When building and editing compositions, the user can tolerate the occasional loss of slivers. If in doubt about scene content, he can always turn up the sampling density.

Updating. In build or edit mode, only update the part of the screen involved in change. If a subsolid is modified or newly created and added to the composition, only the subsolid's bounding box in the screen needs to be rescanned to update the picture. To be effective, the screen should show multiple views of the composition simultaneously, such as three orthographic views and a perspective view in a preferred direction, to minimize the user's need for view changes. This update technique particularly applies to raster displays.

Figure 15 shows six sample pictures made by casting rays. Box enclosures, dynamic bounding, and coherence were used. For each picture the screen was sampled with a density of about 100 × 100 (i.e., 10 000) rays and new edges were located via binary searches. Then all edges were followed by casting additional rays at one pixel increments on the two sides of the edges. Each picture was drawn on a Tektronix tube at 780 × 780 resolution. CPU seconds used by this PL1 program on an IBM 3033 processor were (a) 13, (b) 17, (c) 9, (d) 15, (e) 10, (f) 95. The scene in the last picture is composed of over 1000 primitives (over 3000 surfaces). I modeled it quickly by simply replicating an initial solid composed of four primitives: a block, spheroid, elliptical cone, and elliptical cylinder (i.e., 4 + 4 = 8, 8 + 8 = 16, ..., 512 + 512 = 1024).

ANTIALIASING. Aliasing is an undesirable effect of point sampling techniques and is a common problem with raster display algorithms. Details in the scene smaller than the spacing between sample points may be lost. Specifically, linear or smoothly curved edges will appear jagged and the edges bounding surface slivers, smaller than a pixel in width, may be overlooked. These effects are particularly objectionable in animations because movement of the image makes the jagged edges look like moving escalators as they repeatedly appear and disappear in the image. Jagged edges can be smoothed, however, and slivers occur very infrequently.

The jagged edges in a line drawing can be smoothed by edge following. The purpose of such an algorithm is to minimize the number of lines needed to draw the

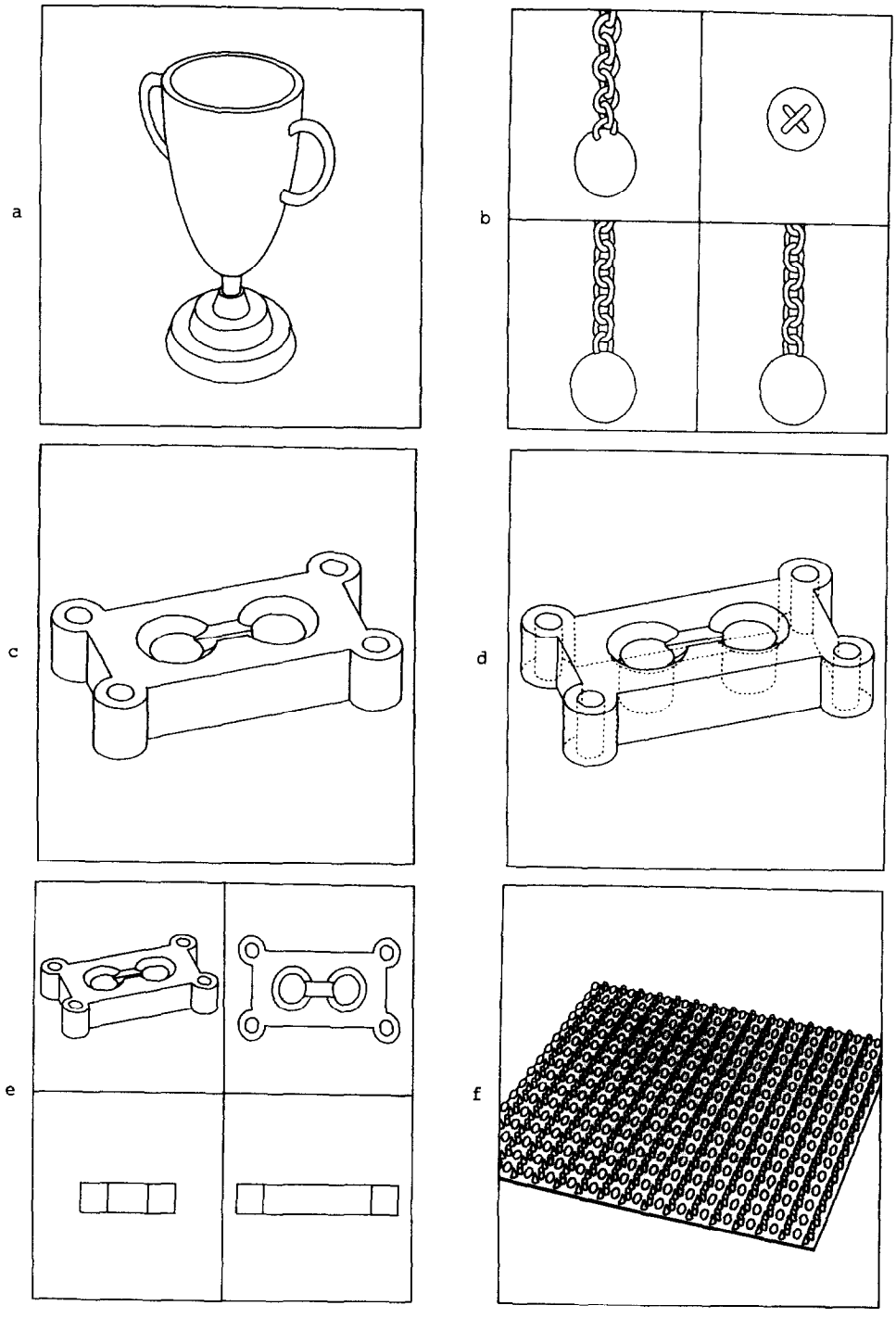


FIG. 15. Sample pictures made by casting rays.

picture within one pixel accuracy. Smooth edges result. The pictures in Fig. 15 were drawn this way. The algorithm quickly converts a "raster edge" (i.e., chain encoded edge) into a vector line sequence. CPU time used is proportional to the length of the given raster edge.

To smooth the jagged edges in a shaded picture with subpixel accuracy, additional rays must be cast for more information about the edges. Via binary search, if the visible surface at pixel (X, Y) is different than the visible surface at pixel $(X + 1, Y)$, then a ray would be generated midway between them at $(X + .5, Y)$ and the visible surface there identified. The distance between sample points could be further subdivided, but the search need not be deep. The binary search depth to smooth jagged edges is a function of the intensity gradient across the edge. Since (1) the area of the image that contains edges is usually a small percentage of the total area and (2) the extra rays cast in binary searches can be bounded in depth—that of the visible primitives forming the edges—the cost for smoothing jagged edges is affordable.

Slivers are solid faces (bounded surfaces) that are thinner than a pixel in width when displayed. Most slivers need not be detected because they lie along the edges of larger faces, so the slivers' edges coincide with others that are displayed. If the sliver is completely isolated, however, it may be overlooked. Whitted [14] partially solves this problem by measuring the diameters of enclosing spheres in the image when initializing the picture making process. Enclosures smaller than a pixel are enlarged to cover a pixel. Then, when a ray intersects the enlarged enclosure, additional rays are generated in search of the sliver. This does not always work, however. If the sliver is long and thin, lying between two rows of pixels, then the enclosing sphere will cover many pixels which are not even neighbors to the sliver. In general, intersecting surfaces may leave a very small volume that is not practical to look for.

HOW TO USE RAY CASTING TO

—interact by poking-at-it;
—detect occluded edges for display;
—detect "non-displayable" edges for removal

Ray casting intrinsically supports poke-at-it interaction. Given a graphics display with an attached pointing device such as a light pen, tablet, or cross hairs, users can send probing rays into the scene by picking points on the screen. The picked point on the screen and the camera model define a ray. Given the ray, RAYCAST finds the surfaces through which it passes. So by probing the scene with rays, the user can specify surfaces or solids for editing, assigning attributes, etc.

Detecting occluded edges for display is more expensive than removing them, an unusual situation in computer graphics. The simple line drawing algorithm presented only finds and displays visible edges. To do so, it uses an array of surface pointers:

$$\text{SURF\_ID}[1{:}W].$$

It contains pointers to the surfaces visible at each of the W pixels on a row of the screen. When pixel i in a row is being processed, SURF_ID$[1{:}i-1]$ points to the visible surfaces in pixels 1 through $i-1$ on the current row and SURF_ID$[i{:}W]$ points to the visible surfaces in pixels i through W on the previous row. This is

adequate because all surface pointer comparisons are to the adjacent pixels on the left or above. To detect occluded edges, the same algorithm works but more memory is needed. All of the surfaces through which a ray passes must be stored, ordered by depth. If an array is used, it must have enough space for the worst case. Since a ray can intersect each primitive at two points, the array must be declared

$$\text{SURF\_ID}[1{:}W, 1{:}2 * N],$$

where N is the number of primitives in the composition. For a display width of 500 pixels (W) and a composition of 300 primitives (N), the array would have length

$$500 * 2 * 300 = 300\,000 \text{ words.}$$

Almost all of this space will never be used, however. Even if it were half filled, the edges formed by 150 transparent surfaces would just blacken the display. For clarity, the number of transparent surfaces in depth should be limited. Seven or so depths would be a practical upper limit and one would be a good default. A maximum of seven surface depths in this example would require only $500 * 7$ words of memory. A depth of four transparent surfaces is shown in Fig. 16.

"Nondisplayable" edges are not visible in shaded pictures, so may not be wanted in line drawings. A nondisplayable edge, like a displayable edge, is defined by the intersection of two surfaces. If the two surfaces are mutually tangent along the edge, then it is nondisplayable (Fig. 17). That is, the normals change continuously moving from one surface to the other across the edge. Just checking for the equality of surface normals along the edge suffices, however.

The simple line drawing algorithm presented displays nondisplayable edges; surface normals are not computed and analyzed. Unfortunately, the surface normals along edges cannot be directly computed because the exact locations of edges are not known. Only known are the surfaces visible at the midpoints of pixels, the points sampled by rays. Edges are inferred to be visible between the midpoints of adjacent pixels. Points on or very close to the edge can be found, however, via binary search. For efficiency, note:

—If both surfaces are planar, a case worth checking, then their normals are invariant and so binary search is unnecessary. This test alone will produce good results.

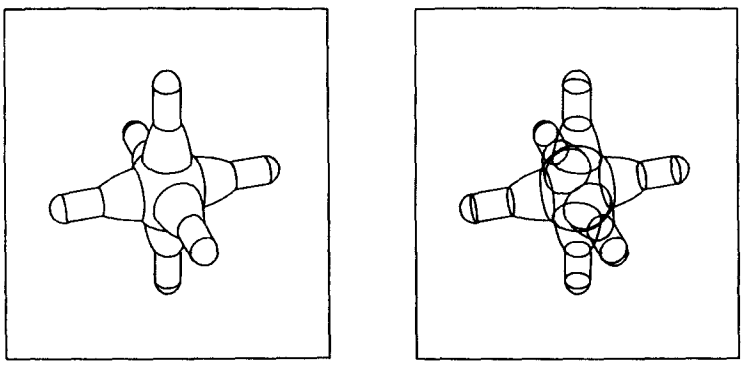


FIG. 16. Display of occluded edges.

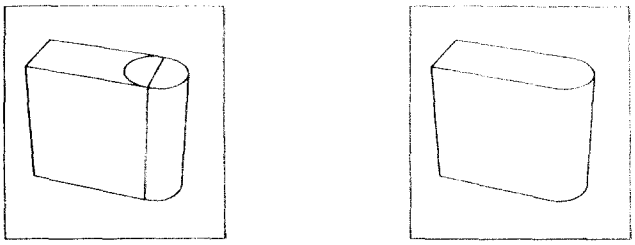


FIG. 17. Example of nondisplayable edges.

—The edges on primitives will probably always be displayable. Exceptions can be marked a priori. Then, if the two visible surfaces forming the edge in question belong to the same primitive, their displayability is already known.

—If the surface normals are different at one point on the edge, they will be different along the entire edge. The converse, however, is not true because the surfaces may be tangent at points only. For example, consider the two points where the ellipses, displayable edges, intersect in Fig. 13. What is the maximum number of tangent points possible per edge, given a pair of surface types?

—The rays cast in the binary search can be bounded in depth, the depth being the minimum of the maximum depths of the two surfaces' primitives.

5.2. *Realistic Shaded Pictures*

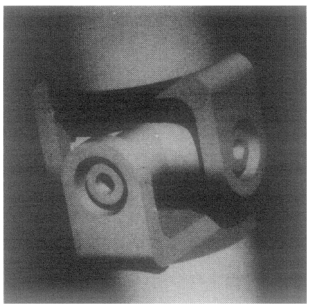
Ray casting is a natural modeling tool for making shaded pictures. MAGI [4] used it for this purpose, producing basic continuous tone pictures that modeled diffuse reflections. Whitted [14] extended the use of ray casting to depict shadows from multiple light sources, transparencies with refraction, and specular reflections ("mirroring"). Ray casting was also used to antialias the pictures.

At the GM Research Labs, Daniel Bass and I implemented a system coincidentally similar to Whitted's. Our system produced the pictures in Fig. 18 on a Ramtek color raster display. To compose pictures, the system provided the user with the following controls:

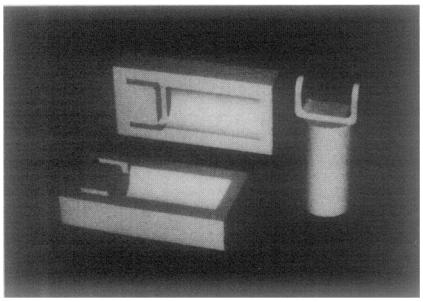
View
- —viewing direction and position
- —focal length: wide angle perspective to parallel
- —zoom factor

Illumination
- —number of light sources
- —directions and intensities of lights
- —optionally shadow
- —intensities of ambient light and background

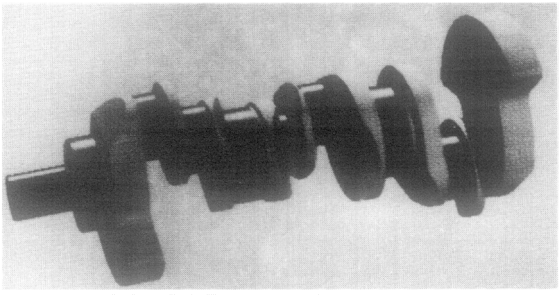
Surface Reflectance
- —% reflected diffusely
- —% reflected specularly
- —% transmitted



U-joint



Mold for u-joint arm



Crankshaft

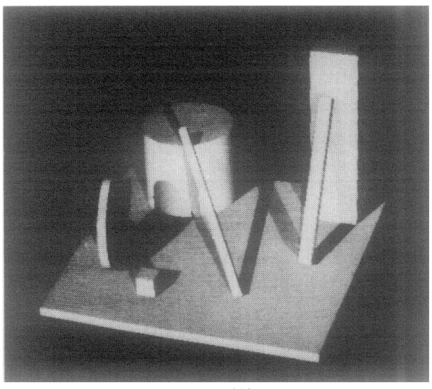
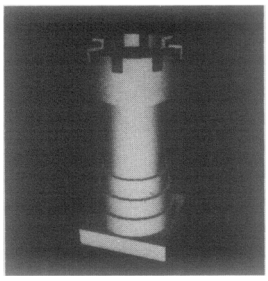


Table scene with shadows



Chess rook

FIG. 18. Shaded pictures via ray casting.

The view items define the camera model. The illumination items define the lighting for the scene, with shadows optional. All light sources are point light sources. The three surface reflectance items are attributes of surfaces. The user can assign them to solids or individual surfaces by poking at the screen. This information as well as the surfaces' equations for computing surface normals are stored in SURFACE data structures, linked to primitives in the composition tree.

Shading algorithms that implement all of these realistic effects are computationally expensive, but relatively simple. The basic algorithm in Section 2.3 suffices for modeling opaque objects reflecting diffusely with highlights—that is, with specular reflections of light sources. For each pixel a ray is cast into the scene, the visible surface identified, the surface normal at the visible point computed, and the

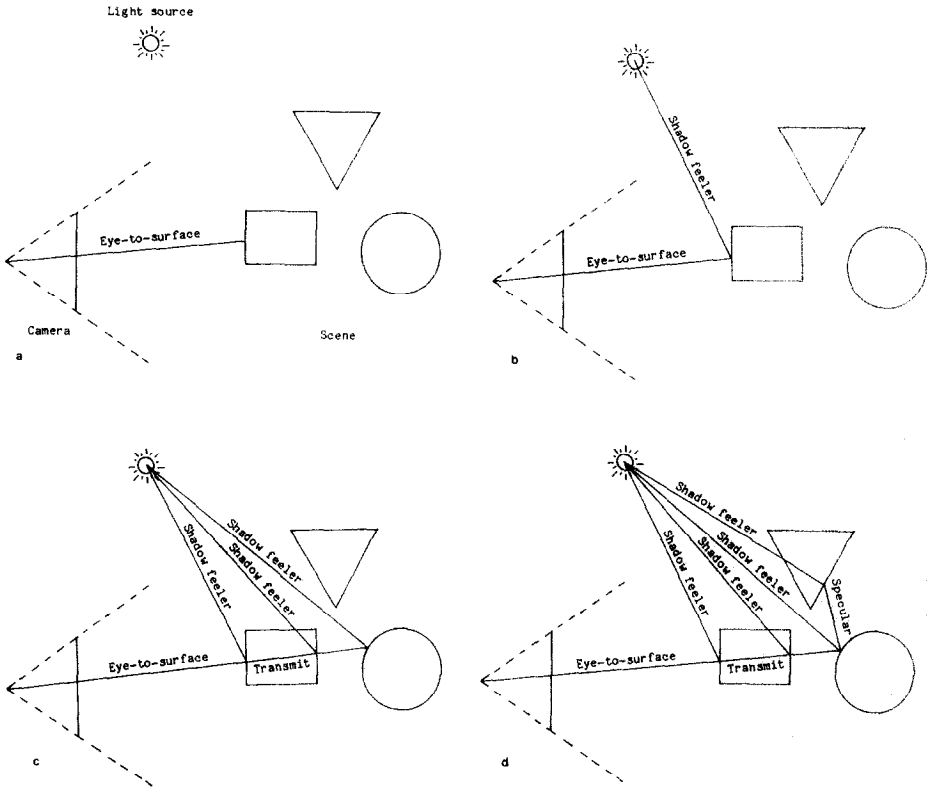


FIG. 19 (a) Diffuse reflection only. (b) With shadow. (c) With transparent rectangle. (d) With specular circle.

visible light intensity computed. To model shadows, refractive transparencies, and general specularity, additional rays must be cast. The sequence in Figs. 19a–d illustrate the rays cast to model the various special effects, just to compute the intensity of a single pixel.

SHADOWS. Given a visible point on a surface, the following procedure is executed for each light source:

—Cast a secondary ray connecting the visible point with the light source
—If the ray intersects any surface between the visible point and the light source, then the point is in shadow
—If a surface that shadows is transparent, then appropriately attenuate the intensity of the light that passes through

The shadow rays are not constrained by the camera model, so the box enclosure tests of RAYCAST cannot be used. For light sources in the scene, the shadow rays must be classified with a version of RAYCAST that uses sphere enclosures. However, light sources outside of the scene can be modeled individually as virtual cameras in order to use box enclosures for greater efficiency.

TRANSPARENCIES. Transparent, actually semitransparent, surfaces may be modeled with or without refraction. If nonrefractive, the task is easy. RAYCAST returns all of the intersection points given an unbounded ray. If the first surface

intersected transmits light, then the second surface is also processed, etc. The light reflected toward the viewer along the ray must be attenuated by the transmittance percentages attributed to the surfaces.

To model refraction, a secondary ray from the visible surface point must be cast at an angle determined by the solid's index of refraction. The secondary ray is then processed as a specular ray. For the refraction formula and pictorial examples, see Whitted's paper [14].

SPECULAR REFLECTIONS. Glass, liquid, and smooth metallic surfaces reflect specularly. To model specular reflections, a secondary ray must be cast to detect the visible mirror image. The secondary ray originates on the specular surface, lies in the plane defined by the surface normal and eye-to-surface ray, and makes the same angle with respect to the surface normal as the eye-to-surface ray does. If the specular ray intersects another specular surface, then another specular ray could be cast, etc. For most applications, however, one bounce is enough. Specular rays as well as refractive rays are not constrained by the camera model, so a version of RAYCAST that uses sphere enclosures must be used.

EFFICIENCY. Many of the ideas presented in the previous section, "Line Drawings for Interactive Modeling", apply. Updating in build or edit mode, in particular, is easy to implement given a raster display.

Refracting and specular rays are bounded at one end; shadow rays are bounded at both ends. Ray bounds always make ray classification faster. Dynamic bounding can be used, but if some surfaces are transparent and nonrefractive, then RAYCAST must check this surface attribute before tightening the bound.

Antialiasing and applying coherence are more complicated when modeling shadows, transparencies, and specularities because secondary rays are cast. Binary searches for edges in the image require neighbor comparisons. For a line drawing of visible surfaces, this simply amounts to comparing visible surface pointers. But for pictures with the special effects mentioned, all of the surfaces and lights involved in the computation of a pixel's intensity have to be compared with that of neighboring pixels. This is possible, but a little awkward. Bass and I only antialiased edges formed by the primary surfaces. Whitted [14] simply antialiased gradients in the image that exceeded a threshold.

AREA LIGHT SOURCES. Area light sources are not practical to model by casting rays. A natural example is a bright hemisphere of sky. Subtler, surfaces illuminate neighboring surfaces—which is why shadows are rarely pure black. Most graphics systems approximate the overall effect via "ambient light," an omnipresent light source of low intensity. A bounded area light source can be approximated by using an array of point light sources. Note, though, that the cost of computing shadows is proportional to the number of lights.

5.3. Solid Analysis

Since solid analysis is a broad subject, where problems vary as do the functions of the objects modeled, a geometric modeling system should provide the means for easily:

—developing new analysis programs;
—transferring the models to other systems.

As new design issues arise and the domain of objects modeled widens, new programs will have to be developed. Of course, a quick way to broaden capability is to interface with other CAD/CAM and FEA systems and use theirs.

Below, "Mass Properties" describes some analysis routines that have been implemented using ray casting. The mass properties are computed via polyhedral approximation. Then "Conversion to Polyhedra" directly addresses this representation because it is simple to analyze and is most commonly used by other systems.

MASS PROPERTIES. Boyse at the GM Research Laboratories, employing the basic algorithm for computing volume presented in Section 2.3, wrote a program that also calculates the centroid, moments of inertia, and products of inertia given a solid model. The program partitions a solid into volume elements by casting rays through it and enclosing each "inside" segment of each ray with a rectangular parallelepiped. Figure 20 shows how one parallelepiped fits around a ray segment and Fig. 21 shows this applied to a 2-D array of rays cast through a simple solid. (Hidden lines are not removed from the pictures of rays and parallelepipeds.) The density of rays in this example is sparse, for illustration. Of course, as the density of rays increases, the volume approximation increases in precision.

Boyse used a random number generator to position the rays in pixels. The user specifies the number of rays to cast, which in turn determines the virtual screen density—e.g., 160,000 rays means 400×400 pixels. Random numbers then decide exactly where in each pixel the ray originates. In the mass calculations, however, the rays are processed as if they were uniformly distributed in the screen. The random positioning has two advantages over center positioning: (1) it eliminates consistent over/under estimates along edges in the image and (2) a second execution, using different random numbers, can be compared and averaged with the first one.

If different parts of the solid have different densities for modeling purposes, density can be introduced as an attribute of primitives. When two solids with different densities occupy the same space via the + or & operators, precedence must be specified by the user. Alternatively, a rule could be enforced that requires solids of different densities to be assembled only—i.e., to be the union of nonoverlapping solids.

CONVERSION TO POLYHEDRA. Polyhedra are a "canonical" representation for solids because so many FEA, graphics, and general modeling systems use them. If not polyhedra, they accept exterior polygons as a surface model. These systems use polygons and polyhedra for simplicity; the only surface type is the plane.

The advantage of starting with an "analytical" model of a solid, using planar and curved surfaces, and then converting it as needed to a polyhedron is twofold:

—Arbitrary precision of the approximation is available on demand;
—The analytical model is much more succinct.

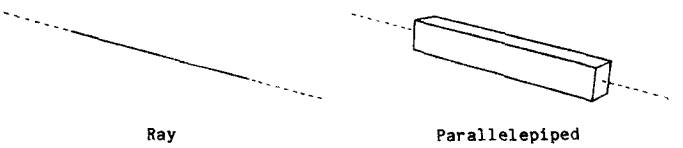


FIG. 20. Enclosing "inside" ray segment with a parallelepiped.

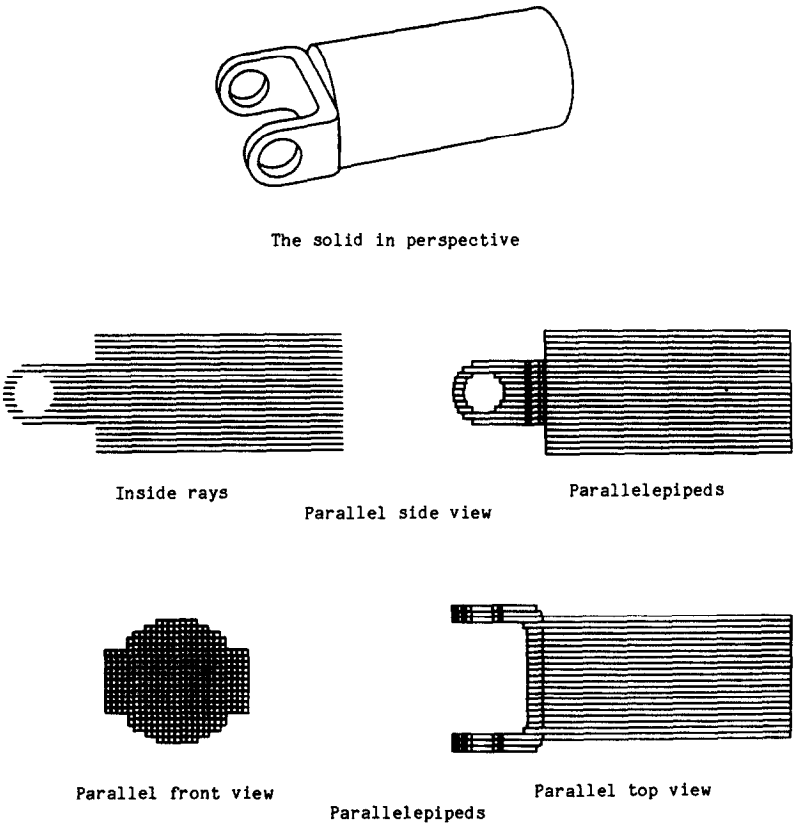


FIG. 21. Parallelepiped approximation to a solid model.

However, if a solid with curved surfaces is originally modeled as a polyhedron, the maximum resolution is fixed.

The mass properties algorithm, described previously, conceptually converts a solid model into a polyhedron—but as a general conversion algorithm, it must be refined for succinctness, precision, and special applications such as FEA mesh generation.

SUCCINCTNESS. To minimize the polyhedron description, polygons internal to the polyhedron should be deleted and contiguous, coplanar polygons should be merged. The best merge and simplify strategy depends on the application. In particular, it does not apply to FEA meshes because elements must remain distinct.

PRECISION. For precision or uniformity in the approximation, the density of polygons should be related to the curvature and dimensions of the analytical surfaces being modeled. One way to achieve this is to initially probe the solid with a high density of rays and then apply the "make succinct" operation above using a maximum deviation of surface normals in the merge criteria.

For certain applications, such as calculating surface area, deep slivers in the solid are important to detect. By deep sliver, I mean a thin slot in the solid parallel to the sampling rays that may have a small volume but significant surface area. This is a special case of the aliasing problem described in Section 5.1. The binary search algorithm described can be used here also. One way to detect deep, isolated slivers is to cast rays in three orthogonal directions. The only parts of the solid that can

escape detection are very thin poles, smaller in diameter than the spacing between rays. If the rays are sufficiently dense, any overlooked sliver should only negligibly contribute to the total surface area of the solid.

FEA MESHES. Ray casting should help to generate meshes for finite element analysis. The basic solid element used by NASTRAN and other FEA systems is the 20-noded iso-parametric brick. Eight of the 20 nodes are the vertices of the brick and the other 12 are midpoints along the brick's edges. The midnodes need not be collinear with the edge's end-points. When not, NASTRAN fits a conic through the three points. This is ideal for solids modeled with quadric surfaces because bricks are easy to generate and when they lie along the surfaces of a solid, the brick's midnodes can be found on the solid's surfaces to accurately model the shape. The positions of the midnodes would naturally be found by casting rays midway between the rays that pass through the brick's vertices.

6. EXTENSIONS AND CONCLUSIONS

This section presents two unrelated ways to extend ray casting: "on" classification and adding new primitives. The possibility and ramifications of a ray being "on" the surface of a primitive, a state other than "in" or "out," is first discussed. The second part of this section explains the ease with which a ray casting system can be extended with new primitives. New primitives, of course, would be bounded by new surface types.

6.1. On Classification

Ray classification, as I have presented it, distinguishes between IN and OUT only, disregarding the ON case. Not singling out the ON case has a subtle, albeit highly improbable, side effect: "dangling" edges. Accounting for the ON case, however, increases the complexity of the ray casting algorithm and inhibits extensions to new surface types.

Only & and − operators can produce a dangling edge, and then only in certain circumstances. For example, consider blocks *A* and *B* that share a face but are otherwise disjoint. Their intersection could produce the effect shown in Fig. 12—a dangling surface that appears as an edge in the screen. For this to occur in practice,

—The operation must be the peculiar (basically empty) one shown.
—The surface left dangling must be perpendicular to the screen (i.e., parallel to the rays).
—The rays must be cast so that they lie "exactly" on the surfaces.

If seven digits of position accuracy are maintained, "exactly" means that the rays cast must be within $0.000001*D$ of the dangling surface, where D is the diameter of the composition.

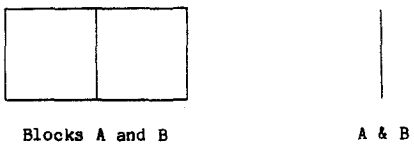


FIG. 22. Dangling edge.

Actually, the dangling edge effect can be eliminated without restoring to ON classification. A dangling edge is a pixel wide face in the screen. By anti-aliasing the edge with additional rays (see Section 5.1), it would disappear. Pixel wide faces are exceptional and easy to detect, so the cost of removing any dangling edges is affordable. But the chances of any appearing are so small, the extra coding does not seem to be worth it.

If ON classification is desired, however, neighborhood models must be used [1, 2]. A neighborhood model for a line segment that is ON a solid describes where the solid's interior is with respect to the line. ON classifications must then be combined using "regularized" combine operators [2]. A combine operation is regularized by the "closure of the interior."

6.2. Adding New Primitives

Extensibility is a major advantage of the ray casting method for solid modeling. Dimensions always multiply—intersecting an entity of order N with an entity of order M yields a problem of order $N * M$. Fortunately, rays are first order, 1-D entities in 3-space, and intersecting rays with surfaces is the most complicated calculation in a ray casting system.

A new surface type may be added by introducing a new primitive bounded by the surface. To support a new primitive, routines are needed to perform the following tasks: create primitive, move primitive, find ray–primitive intersections, compute surface normal given a point, enclose primitive.

This list exhausts the case analysis for primitives, and most of these routines require less than one page of code.

To "create primitive," a new solid data structure must simply be allocated and its attributes assigned (see Table 1). One procedure for creating all primitives suffices, but it must be updated to accommodate additions.

The implementation of "move primitive" depends largely on how scaling is to be treated—whether radii or lengths are involved. Rotations and translations should be the same for all primitives.

"Find ray–primitive intersections" reduces to finding ray–surface intersection points and a check to determine if the points found are within the bounds of the primitive. This computation is in the fixed, local coordinate system of the primitive, simplifying the problem. If the surface can be represented by a 4th-order polynomial or less, the problem has a closed form solution. Whitted [14] describes an algorithm based on work by Catmull and Clark [15] for finding the points where a line intersects a bicubic B-spline surface. With a surface patch capability, sculptured or "free form" objects could be modeled. An algorithm has yet to be implemented, however, and speed is important.

"Compute surface normal given a point" is usually not very difficult and, to simplify the problem, it can be done in the local coordinate system of the primitive.

Last of all, "enclose primitive" only requires a list of vertices that defines a polyhedron enclosing the primitive in its local coordinate system, as explained in Section 4.2.

6.3. Conclusions

If a CAD/CAM system is to provide more than automated drafting—to fully support design, analysis, and manufacturing—then it must have a solid modeling

capability. The ray casting solution can satisfy this need. The method is simple, and therefore maintainable, with a very broad geometric coverage.

Subsecond CPU times for picture generation are needed and, as I will explain, are attainable while interactively building and editing solids. The average CPU time for making Figs. 15a–e was less than 13 sec. All were drawn at a resolution of 780 × 780 pixels. By drawing the pictures at half the resolution, 390 × 390, the CPU time used would be about 13/(2 * 2) sec.—say 3.5 sec. The image quality would be like that in the quadrants of Figs. 15b and 15e, but, of course, large enough to fill the display. Providing the user with control over picture resolution is easy. Finally, the 3.5 sec. could be divided by at least 4 (more likely 10) on average by using the update mode described in Section 5.1. That is, as primitives or subcompositions are edited or newly added to composition, only the area of the screen affected would be recomputed and refreshed.

To further improve performance, the implementation could be reworked. The ray–solid evaluator RAYCAST and the inner loops of the edge finding and following code could be written in assembly language and optimized. This might cut the execution time of the PL1 version in half. Replacing the floating point calculations with scaled integer calculations would also make a measurable improvement. This would be particularly appropriate for a microprocessor implementation.

REFERENCES

1. J. W. Boyse, *Preliminary Design for a Geometric Modeller*, GMR-2768, Computer Science Department, GM Research Labs, Mar. 1978.
2. H. B. Voelcker and A. A. G. Requicha, Geometric modeling of mechanical parts and processes, *Computer*, **10**, 12, Dec. 1977.
3. A. A. G. Requicha, Representations for rigid solids: Theory, methods, and systems, *ACM Computing Surveys*, **12**, 4, Dec. 1980.
4. R. A. Goldstein and R. Nagel, 3-D visual simulation, *Simulation*, **16**, 1, Jan. 1971, 25–31.
5. *TIPS-1, Technical Information Processing System*, Institute of Precision Engineering, Hokkaido University, Sapporo, 060 Japan, 1978.
6. G. Spur, Status and Further Development of the Geometric Modeling System COMPAC, Proc. Workshop on Geometric Modeling, CAMI Inc., Mar. 1978.
7. I. C. Braid, New Directions in Geometric Modeling, Proc. Workshop on Geometric Modeling, CAMI Inc., Mar. 1978.
8. R. B. Tilove, Set membership classification: A unified approach to geometric intersection problems, *IEEE Trans. Comput.* **C-29**, 10, Oct. 1980, 874–883.
9. R. A. Goldstein and L. Malin, 3D Modeling with the Syntha Vision System, First Annual Conference on Computer Graphics in CAD/CAM Systems, MIT, pp. 244–247, Apr. 1979.
10. S. D. Roth, Stereo 3-D Perception for a Robot, Ph.D. Thesis, California Institute of Technology, Pasadena, Calif., Mar. 1978.
11. W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1979.
12. S. M. Selby, (Editor-in-chief) *Standard Mathematical Tables*, 19th Edition, p. 106, The Chemical Rubber Co., Ohio, 1971.
13. S. R. Levine, *Interactive 3-D Motion Graphics with Large Data Bases*, (Ph.D. Thesis, Stanford University), SLAC 192, Stanford Linear Accelerator Center, Mar. 1976.
14. T. Whitted, An improved illumination model for shaded display, *Comm. ACM*, **23**, 6, June 1980, 343–349.
15. E. Catmull and J. H. Clark, Recursively generated B-spline surfaces on arbitrary topological meshes, *Computer Aided Design*, **10**, 6, Nov. 1978, 350–355.