MIT Laboratory for
Computer Science

Massachusetts
Institute of
Technology

MIT LCS TR--84

Immediate-Mode Ray-Casting

JOHN ALEX
Brown Computer
Graphics Group

SETH TELLER
MIT Computer
Graphics Group

June 1999

545 Technology Square Cambridge, Massachusetts 02139 (617) 253–5851

Immediate-Mode Ray-Casting

John Alex
Brown CG Group

Seth Teller
MIT CG Group

http://graphics.lcs.mit.edu

June 1999

Abstract

We propose a simple modification to the classical polygon rasterization pipeline that enables exact, efficient raycasting of bounded implicit surfaces without the use of a global spatial data structure or bounding hierarchy. Our algorithm requires two descriptions for each object: a (possibly non-convex) polyhedral bounding volume, and an implicit equation (including, optionally, a number of clipping planes). Unlike conventional raycasters, the modified pipeline is unidirectional and operates in immediate mode, making hardware implementation feasible. We discuss an extension to the OpenGL state machine that enables immediate-mode raycasting while making no modification to OpenGL's architecture for high-performance polygon rendering. A software simulation of our algorithm generates scenes of visual fidelity equal to those produced by a conventional raycaster, and superior to those produced by a polygon rasterizer, significantly faster than either existing method alone.

1 Introduction

The problem of reducing 3-D scenes to 2-D images is fundamental in computer graphics. This "rendering" problem has typically been defined as the process of producing 2-D images (arrays of pixels) from 3-D geometric models (collections of geometric objects) and lighting descriptions. The rendering process can be split into the subproblems of visible surface determination (VSD) and illumination. VSD, as the name suggests, determines the visible surfaces as seen from a particular viewpoint; a subsequent illumination computation determines the color of the surface fragments visible to the observer. This paper addresses the VSD subproblem, which has typically been solved either through Z-buffered scan-conversion [9], or raycasting [4]. Although these methods are usually regarded as mutually exclusive, we describe a hybrid algorithm which exploits favorable properties of each. The next two sections preface this description by reviewing both existing methods.

1.1 Scan Conversion with Z-buffering

Transforming a scene description into polygons, then incrementally scan-converting each polygon onto a discrete pixel grid, was first proposed in the late 1960's [36, 31]. The frontmost polygons could be established simply by drawing the polygons, one over the other, in back-to-front order. However, depth-sorting the polygons, as in [31], proved expensive (worse than linear in the number of objects) and difficult (due to many special cases, particularly the occasional need to split polygons). In contrast, z-buffering, proposed in [9], did not require polygon ordering, but resolved visibility independently at each pixel through repeated depth comparisons. Although this method was initially regarded as overly memory-intensive, it did have time complexity linear in the scene description size; as available memory grew and cheapened, depth buffering gained favor over object space methods.

While the z-buffer was introduced to accelerate rendering of curved surface patches (tesselated into polygons), it has been extended to other surface types. Blinn describes an algorithm for scan-converting parametrically defined surfaces [7]. Heckbert describes algorithms for scan-converting general quadrics into a Z-buffer using finite differences [17]. In principle, higher-order polynomials or even non-polynomial functions could also be scan-converted through finite differencing, though this would require considerable case analysis and dedicated code for each type of surface treated.

Scan-converting has enjoyed its widest usage for polygon rendering. Specialized hardware "geometry engines" have been built to accelerate the extensive calculations inherent in polygon transformation, clipping, shading and rasterization [11]. Such dedicated rendering architectures have since formed a major development path, as successive versions have incorporated depth-buffering, texturing, and many other extended capabilities (e.g., [3, 1, 2, 22, 32]).

Polygons have proven to be a versatile geometric modeling primitive. However, they exhibit several limitations when used to render curved surfaces. First, polygonal approximations exhibit geometric aliasing due to their reduction of all shapes to piecewise-linear surfaces. One solution, of course, is adaptive subdivision, but this effectively drives the polygon size to a single pixel, defeating the economies inherent in edge setup, etc., for polygons larger than a pixel, and causing the Gouraud-interpolation hardware to be essentially unused. More sophisticated tessellation schemes exist to terminate subdivision above the pixel level [35] but for arbitrary implicit surfaces, the choice of a well-spaced set of vertices from a dynamically specified viewpoint may be computationally difficult.

Second, the *lighting* in polygonal surface rendering arises from linear interpolation of discrete lighting values from polygon vertices. Even the expensive modification of re-computing the normal at each visible surface fragment [26] does not produce an entirely correct result, due to the underlying faceted geometry's generally incorrect position.

Third, for a given desired shape, the storage space required to describe an accurate tesselation of the shape is typically much greater than that required to describe the shape as an implicit surface. This is a classic space-time tradeoff, typically resolved in favor of static tessellation. However, under dynamic viewing conditions, given the substantial power of a graphics rendering workstation and the relatively lesser power and bandwidth of the host processor and its connection to the graphics subsystem respectively, it is desirable (as we show below) to resolve the tradeoff in favor of lower space usage, i.e., in favor of more compact surface descriptions.

1.2 Raycasting

An alternative approach would be a renderer that takes curved surfaces as primitives, rendering them "exactly" – that is, sampled geometrically, then shaded, at each pixel. This approach sidesteps the drawbacks to implicit surface tesselation listed above. Ray casting, first introduced for polyhedra in [4], and implicit surfaces in [33], is such an algorithm; it cleanly and uniformly handles both polygons and general implicit surfaces. That is, ray casting renders any implicit equation $F(x, y, z) = 0$, given the ability to solve those equations for an arbitrary ray. Unlike scan-conversion, ray casting does not require any specialized knowledge of surface geometry.

However, polygon scan-conversion has remained the de facto standard for interactive work because ray casting, while very general, is also slow, at least when using a modest number of modern general-purpose processors. In order to determine correct visibility, in principle every object must be checked along each sight ray. The classic running time for a naive raycaster, assuming one sample ray per pixel, is

$O(n \cdot r)$, where n is the number of objects, and r is the number of pixels, respectively. While ray-casting acceleration hardware has been built (e.g., [12]), it does not scale well to complex scenes. However, in special circumstances, for example with large numbers of processors [23, 24], or with specialized parallel hardware [28] or for restricted scene geometries [18], or CSG applications [12] it has proven useful.

1.3 Raycasting Bounded Objects

Bounding volume hierarchies (e.g., [10, 30]) change the practical scaling behavior of ray casting. Reasonably tight bounding volumes are often analytically discoverable for finite surfaces. (For implicit equations about which no *a priori* bounding information is known, an infinite bounding volume, while inefficient, will still produce correct results). Kay and Kajiya [20] describe methods for computing arbitrarily tight convex bounding volumes (if they exist) for polyhedra, implicit surfaces, and compound objects. Heckbert [17] describes such methods specifically for quadrics. Cameron gives efficient conservative bounding algorithms for full CSG hierarchies [8].

While tessellations need be of a certain density in order to guarantee image quality from a given viewpoint, bounding volumes are conservative, viewpoint-independent, and unrelated to image quality; they serve solely to accelerate rendering. They have generally been used in ray casting as a trivial-reject mechanism involving an extra ray-bound intersection: if a ray misses the bounding volume or intersects it at a deeper point than the closest object hit so far, the enclosed object need not be tested. In order to be useful, such bounding volumes should be simple in order to admit efficient intersection tests. This creates an intricate tradeoff: simpler bounding volumes lead to cheaper ray-bounds test, but also to more spurious ray-object tests. In practice, cuboids and spheres are often used.

Bounding volumes also enable the use of spatial data structures, such as spatial subdivisions [13, 14] or hierarchies of bounding volumes [30, 20]. These allow each ray to be tested against a tightly bounded set of objects in nearly front-to-back order; each ray can stop soon after finding an intersection. However, spatial data structures require two non-trivial operations, one-time creation and per-pixel traversal. This situation engenders another tradeoff: the more objects are in a cell (we use the term to refer to both a bounding volume or a node of a spatial subdivision), the easier it is to construct and traverse the tree, but the more time is spent inspecting each cell for intersections. This tradeoff is crucial to the speed of software raycasters, but not yet well understood. In any event, these structures are commonly employed in software ray casting implementations, but are difficult to incorporate into hardware due to the complexity of represented state.

Spatial data structures are also ill-suited to scenes with moving objects; the structure must be repopulated in each frame, a prohibitively expensive operation when the number of moving objects is large. Most importantly, hardware acceleration – the key to polygon rendering's success – is unlikely for classical ray casting because storing such large structures in hardware state is impractical. Much of the literature on raytracing in hardware has focused on parallel processing issues that are relevant but beyond the scope of this paper. That is, they concentrate on doing a fixed amount of work faster, rather than reducing the required work. An overview of such methods, which covers the acceleration of secondary as well as primary ray computations, can be found in [19].

1.4 A Hybrid of Scan-conversion and Raycasting

The separation of raycasting and scan-conversion has cemented into the now-traditional characterization of the two algorithms: scan-conversion maps single objects "forward" to many pixels, while raycasting maps one pixel "backward" to many objects. However, we note that practitioners have cleverly implemented both algorithms to run efficiently on a broad range of objects and scenes by adopting each other's techniques.

Bounding volumes are ideally simple, and scan-conversion excels at rendering simple shapes; yet bounding volumes are used most in raycasting. Actual modelled objects are ideally complex and exact, and raycasting cleanly handles such surfaces; yet such objects are instead approximated with many small triangles. We suggest a transposition: bounding volumes should be tessellated, while actual objects should be raycast. We proceed to describe a hybrid representation which combines the strengths of both approaches, while exhibiting the disadvantages of neither.

2 Immediate-Mode Ray-Casting

This section describes a set of proposed modifications to an existing polygon scan-conversion system to enable immediate-mode raycasting of bounded implicit objects. We first review the popular OpenGL [25] architecture. This requires an immediate-mode specification of a collection of polygons, with optional normal, texture, etc. coordinates specified at each vertex. The essence of the standard OpenGL polygon rasterization pipeline, for example, is summarized by the following pseudocode. For this pipeline, scene objects are issued in immediate mode as collections of polygons (here, for simplicity, assumed to be triangles):

```
For each incoming triangle
  transform and clip vertices
  if back-face cull, done
  light vertices
  for each x,y,z produced by the rasterizer
    interpolate texture coordinates; do texture lookup
    combine texture and lighting values
    read Z-buffer at (x,y)
    if fragment depth is less than current Z-buffer value
      write visible, lit fragment to depth and color buffers
```

In our proposed method, for polygonal rendering, the classical rasterization pipeline architecture is unchanged. However, to render a bounded implicit primitive in immediate mode, four items must be issued to the rasterization pipeline for that primitive:

- An implicit surface equation $f(x, y, z) = 0$;
- A polygonal description of any *bounding polyhedral volume* containing that portion of the implicit surface which is to be rendered;
- Clip plane equations, if desired; and
- Functions $u(x, y, z)$ and $v(x, y, z)$, to generate texture coordinates (u, v) from object-space points (x, y, z).

Surface normals can be produced analytically from the implicit surface equation, so need not be issued separately from the surface geometry as with polygon rasterization. This analytic solution is thus dependent on the form of implicit equations accepted by each particular implementation.

Pseudocode for our proposed hybrid algorithm is below, with changes indicated. The triangles issued now comprise bounding surfaces around the implicit scene objects.

```
-> receive surface, clip equations; store them in immediate-mode state
   for each incoming triangle
     transform vertices
->   [no lighting of vertices]
     if back-face cull, done
     for each x,y,z produced by the rasterizer
     read Z-buffer at (x,y)
->   if newly rasterized depth is closer than stored depth
->     generate ray through center of pixel in object-space
->     compute ray-object intersection using current implicit equation
->     if no such intersection, done
->     if intersection point passes all active clip planes
         if fragment depth is less than current Z-buffer value
->         compute normal and u,v using object, texture equations
->         light fragment
->         do texture lookup
           combine texture and lighting values
           write visible, lit fragment to depth and color buffers
```

Note that z values (distance along the look vector) arise from screen-space rasterization, while the t values (distance along a sight ray) arise from ray-object intersection in an arbitrary space (in our implementation, eye space). Thus the t value of any intersection must be transformed into 3D screen space for correct comparisons to stored z values. This is accomplished with an inner product against the look vector, and a scaling to map the interval $z_n..z_f$ to $-1..1$.

Figure 1 shows a screen-space view of the process. The algorithm could either be serialized to operate one pixel at a time, first scan-converting and then conditionally raycasting, or could be evaluated in parallel at every pixel in the manner of Pineda rasterization [27].

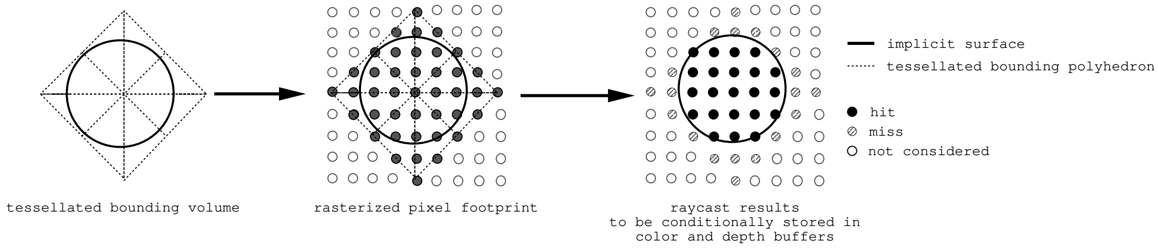

Figure 1: Raycasting a completely visible sphere.

Consider the behavior of this hybrid algorithm on objects whose bounding volumes are completely occluded. Only the bounding volume each such object will be scan-converted; no raycasting will be done, since all scan-converted samples fail the conservative depth test. The depth test has been extensively accelerated with special-purpose hardware, so will be quite fast.

Visible or partially visible objects incur some additional work (Figure 2): their potentially visible samples (those samples, generated from bounding volume rasterization, that pass the z-test) will be raycast, providing exact object sample points.

The bounding volume provides a close approximation to the actual screen-space footprint of the object. This has two implications. First, rays are cast only where this tight bound passes the depth

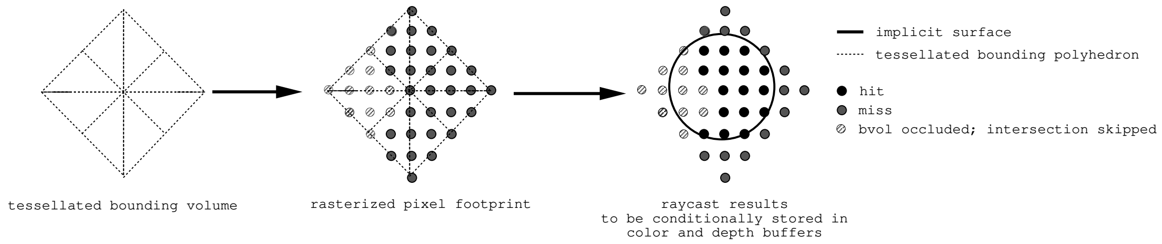

Figure 2: Raycasting a partially occluded sphere.

test; thus, relatively few rays will miss the underlying object. Second, of those rays that hit the object, few will produce fragments which fail the z-test, as the bounding volume provides a tight (but conservative) approximation to depth. In other words, most visible pixels arising from bounding volume scan-conversion will produce visible fragments. Thus, the method achieves tight generally non-convex bounds around each rendered object, while avoiding tests for ray intersection with a complex bounding hierarchy, as would be performed by classical ray casting.

2.1 Discussion

The modified rasterization pipeline above, like a scan-converter, takes objects to pixels, albeit indirectly. The scan conversion step takes bounding volumes to pixels; the raycasting step, given pixels and implicit equations, produces samples lying on implicit surfaces. It is also general, making no assumptions about the implicit surfaces (convexity of either the surface or the bounding volume, for example, is not required).

Our algorithm does not require global state for complex scenes, as would traditional efficient raycasters. By processing each object completely, one at a time (rather than processing each ray completely) and storing its results in a z-buffer, raycasting can be done in immediate mode, with all the advantages that entails. By adding ray-object intersection capabilities to the already formidable polygon rendering capabilities of today's hardware, we create a "one-way" raycasting pipeline, amenable to the same optimization and parallelization that polygon hardware has undergone.

The time complexity for this algorithm is relative to the number of pixels covered by each object's bounding polyhedron, and the frequency with which such pixels pass the depth test. Assuming one bounding volume per object, the time complexity is a combination of rasterization effort and raycasting effort:

$$
\begin{array}{rcl}
Time = O(\#\text{ objects} & \times & \#\text{ front-facing polygons per bounding volume} \\
 & \times & \#\text{ samples per front-facing polygon} \\
 & + & \\
\#\text{ objects} & \times & \#\text{ front-facing polygons per bounding volume} \\
 & \times & \#\text{ visible samples per front-facing polygon})
\end{array}
$$

This cost will in general be less than that for rendering tesselated implicit surfaces, for two reasons. First, our method requires far fewer (5-10x, according to our data) polygon scan conversions, since polygons are used only to describe bounding polyhedra, and can do so quite crudely. Second, normals need not be issued along with polygon vertices, as in standard rasterization. The cost will also be less than that for ray-casting implicit surfaces, as intersecting rays are discovered not by ray-bounds tests, but by relatively cheap scan-conversion of bounds. Finally, we note that while classical scan conversion gains nothing from front-to-back traversal, our algorithm exploits this by avoiding most unnecessary ray casts.

2.2 Comparison and Contrast

Our scheme is similar in one respect to the hierarchical z-buffering architecture of [15], in that it uses visibility information about a conservative object bound to pass, or suppress, processing of the information inside that bound. There are two important differences, however. First, in that scheme the appearance of even one visible pixel would mean the processing of the entire contents of the bounding volume. Second, that scheme was not unidirectional; as there could be an unbounded amount of state inside each cell of the hierarchy, a feedback mechanism was required to elicit from the hardware information about the bounding box visibility. If the bounding volume was reported visible, the rendering host would proceed to rasterize the contents of the volume. This organization would introduce significant latency and memory traffic in practice.

The hybrid rasterization scheme presented here bears some similarity to the SOID renderer described nearly fifteen years ago by Heckbert [17], with three important differences, detailed below.

First, Heckbert's work included only quadratic implicit surfaces, and derived from each implicit equation the screen-space bounding box of the corresponding implicit surface. We demonstrate our method using quadratic implicit surfaces as well, but note that the issues of equation complexity and root-finding are independent of the ideas presented here. That is, should an improved (faster, more general, etc.) root-finder be proposed, its incorporation into our algorithm is trivial, and immediately enables the renderer to use higher-order surfaces. (An analogous design consideration arises in renderers based on surface tesselation; for high-order surfaces, either parametric descriptions must be obtained, or the surface must be sampled, to produce polygons suitable for tesselation.)

Secondly, Heckbert's algorithm cast a ray for every pixel within the screen-space axis-aligned bounding box of the implicit object. Our method casts significantly fewer rays, both because the bounding polyhedron will (in general) map to fewer pixels than will any axial bounding box, and because bounding volume samples which are occluded or back-facing will not cause rays to be cast.

Finally we note that, whereas Heckbert's method used axial (screen- aligned) bounding boxes, our method handles general, non-convex, bounding polyhedra, which can be computed off-line if desired. Thus an interesting tradeoff, not present in earlier methods, arises: a rendering system may expend off-line computation (for example, to discover a simpler, or tighter, bounding polyhedon) to achieve better performance during subsequent rendering.

We believe the immense power of polygon rasterization hardware – which are growing on a performance curve even steeper than that of general-purpose CPUs – makes it worthwhile to re-examine older techniques, to determine whether they are now suitable for wider adoption in the service of particular problem domains. Ray-casting is one such older technique; rendering curved, implicit surfaces with accurate geometry and shading is one such problem domain. In the next section we discuss how a combination of the two techniques can be effectively realized through a modest set of changes to a popular existing immediate-mode rendering architecture, to achieve a hybrid algorithm superior to existing methods.

3 Implementation Issues

This section describes the issues that arose in the implementation of the hybrid scheme.

3.1 Generation of Bounding Polyhedra

One implementation issue is the efficient creation of effectively tight bounding polyhedra. These bounding volumes need be generated only once for a static model, but may be difficult to compute efficiently. For convex implicit shapes like cylinders or spheres, successively better approximations that remain on the "outside" of the object are easy to construct. The intersection of slabs produced by the algorithm described in [20] can be used to produce convex bounding volumes. We do not here address the problem of generation of efficient, non-convex bounding polyhedra, except to state that, as improved techniques for such generation emerge, they can be easily incorporated into the method we describe.

We also make an observation that may prove useful in the creation of these bounding volumes. These volumes can be thought of as "windows" onto their contained objects. The object can only be seen "through" its bounding volume, that is, only where its bounding volume contributes visible pixels. This suggests that in certain situations, a strict bounding volume is not required. For example, if the user is constrained to the inside of a building, and a given implicit surface is outside the building and visible only through a certain window, a polygon covering the window could be used as the "bounding volume" of the surface and would work correctly.

3.2 Near-Plane Clipping

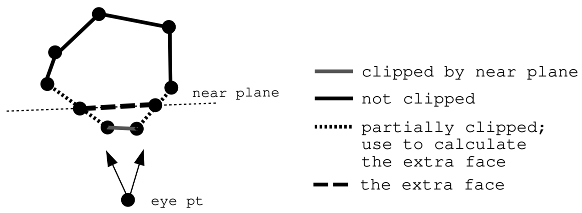

Figure 3: Near plane clipping.

Conceptually, the polyhedral bounds supplied with each object represent solid volumes, in the sense that the corresponding implicit object could be anywhere inside the given bounds. Thus when the near clipping plane clips a front-facing polygon of the bounding volume, but not the back-facing polygon along the same sightline (Figure 3), the pseudo-code given above would render incorrectly. The front face is clipped, and generates no rasterized samples. Samples on the back-face lie "behind" the implicit surface as viewed from the eye, and may be obscured by previously rendered fragments, even though the surface itself is not; thus these too generate no rasterized samples. Thus, if our method does not account for this case, fragments of the implicit surface which should appear would not get drawn. There are at least two possible methods for handling near-plane clipping correctly.

First, when clipping, the host could generate a new face, coplanar with the near plane, that covers all the pixels of the clipped faces. The algorithm to accomplish this is simple. One additional polygon should be constructed; its vertices are generated in order from any faces that cross the near plane [34]. The faces would be drawn in adjacency order using for example a winged-edge data structure [5].

The second method uses graphics hardware to apply a "capped" solid technique (e.g., [29, 21]) *to the bounding polyhedron* of each implicit object. This would ensure the generation of polygon fragments on the near plane, causing in turn the appropriate ray-object intersection to occur at each pixel in the capping area.

3.3 OpenGL Extensions

We retained, of course, OpenGL's rapid polygon-rasterization capability. We modified the OpenGL pipeline as described in §2, to enclose each primitive with a BEGIN/END pair, including a description of the implicit equation (and any clipping planes) to be rendered. The polyhedral bounding volume was then issued using OpenGL's existing mechanisms. Whenever a bounding sample passed the rasterization depth test, ray casting was performed on the underling implicit surface. OpenGL's existing lighting model was then applied on a per-pixel basis.

We considered the issue of placement of the lighting operation in the hybrid method. While in raycasting samples must be lit per-sample, polygons are lit per-vertex in OpenGL. For our hybrid algorithm, we chose to retain per-vertex lighting while rendering ordinary polygons, and to perform per-pixel lighting while rendering implicit primitives. This seemed the fairest choice in terms of timing comparisons. In the first case, our method's behavior matches that of OpenGL. In the second case, we "charge" the hybrid method for per-pixel shading, reasoning that this was an upper bounding on the shading work that would be performed by any polygon-based algorithm. Note that the hybrid method still should gain the advantage, since per-pixel lighting is done only for samples that pass the ray casting step (whereas ordinary rasterization, in contrast, unconditionally lights every vertex of every polygon, regardless of visibility).

3.4 Geometry, Normals and Texture

The set of allowable implicit equations is in no way constrained by our algorithm. Indeed, any root-finding method can be incorporated into our algorithm. As a compromise between generality and ease of hardware implementation, we chose a ten-parameter approach, sufficient to describe any quadric: three for the squared terms, six for the linear and bilinear terms, and one constant. As implicit surfaces can be unbounded, we adopted OpenGL's semantics of up to twelve additional clipping planes, augmenting the six planes of the view frustum. Note that the shape so produced (e.g., a bounded hyperboloid) can be non-convex, and as such would admit a non-convex bounding polyhedron. Just as in ordinary rasterization, in our method one would typically render bounding polyhedra with back-face culling enable. Otherwise, front and back faces of a bounding volume that cover the same screen-space sample would trigger duplicate ray-object intersection tests.

There is no need to issue per-vertex normals in our method, as normals can be generated, analytically and on-demand, from the implicit surface equation.

The usual notion of per-vertex texture coordinates does not apply, since there are no longer vertices to be rendered. Instead, we adopted OpenGL's notion of texture coordinate generation using distance from a plane specified in eye, world, or object space. More sophisticated approaches, for example the issuance of a function that takes a surface point as input and returns texture coordinates, could be incorporated straightforwardly into OpenGL syntax.

3.5 Back-Sample Culling and Capped Solids

We note two straightforward extensions of OpenGL semantics to the implicit surface rasterization scheme presented here, neither of which we implemented. First, we extend back-face culling to "back-sample culling" by (conditionally) discarding those *samples* produced by raycasting which are back-facing with respect to the look vector. If this rejection is disabled, as when back-face culling is disabled in standard OpenGL, the observer can "look inside" of an ordinarily closed implicit volume and see its "back-samples". This effect is achieved in our algorithm simply by retaining, and generating a pixel write for, the nearest back-facing fragment identified along the sample ray.

Second, the capping technique described in [29, 21] may be extended to the rendering of implicit surfaces clipped by an arbitrary plane. In this case, when processing all the roots generated by the ray-surface intersection, rather than simply identify the closest intersection, we identify the closest parametric *interval* which contains the surface, then modify the nearer (front-facing) t value so that the generated sample lies on the clipping plane, and replace the sample's normal with that of the clip plane. Note that this technique requires that the boundary polyhedron itself to be rendered as a capped solid (§3.2) in order to ensure that scan-converted samples are appropriately generated, which will in turn cause the production of the required ray-cast samples.

4 Results

This section demonstrates the implemented algorithm on a commercially available workstation of moderate power (a Sun Ultra 2 Model 2/200). The machine has Creator 3-D graphics capability, which in our tests was used only for writing the 24-bit framebuffer. We instrumented the existing and proposed algorithms while rendering several models. Below, we report each algorithm's performance under a number of metrics.

We implemented our proposed extensions inside Mesa, a public domain pure-software implementation of the OpenGL rendering architecture [25]. We then rendered a variety of scenes using three methods:

RC, an optimized raycaster with ray-cell walking [14];

Mesa, unmodified Mesa (using a tesselated version of each object); and

IMR, our modified pipeline, implemented as a software extension to Mesa.

4.1 Test Models

We used the following test models from Eric Haines' 1987 "Standard Procedural Databases":

Rings, a sphere-and-cylinder ring model;

Tree, a sphere-and-cylinder tree; and

Shell, a seashell model, which states in its documentation that it "tends to bring ray tracers to their knees"[16] as it is composed of several thousand tightly emplaced spheres.

The number of *tessellated* polygons created for each scene in order to supply it to Mesa is reported in table 1. We chose a tesselation level to make the following comparisons "fair" in the following sense. In a preprocessing step, we applied adaptive tessellation until each object's tessellated rendering covered approximately ninety percent of the pixels covered by its raycast equivalent. This metric is easily computable, guarantees a certain quality level, and is resolution independent.

4.2 Performance Measures

We instrumented each rendering method to measure various performance attributes, which we tabulate below. For RC, we always used a spatial subdivision, since naive raycasting is far too slow. For both immediate-mode algorithms (Mesa and IMR), objects were rendered both in random order and with rough front-to-back ordering, at two resolutions: 129x129 ("lo") and 513x513 ("hi"). To our knowledge, these are the first reported comparisons between scan-conversion and ray casting on scenes with curved objects. The instrumented quantities were:

Rasterizer Load, the number of triangles issued, and the number of pixel samples arising;

Raycasting Load, the number of bounding box and implicit surface intersections performed, as well as the number of ray-cell traversals;

Raycasting Efficiency, the ratio of hits to tests in ray casting;

Lighting Load, the number of point sample lighting operations executed.

Rendering Time, the average seconds per frame required;

Below, "M" refers to millions, "K" to thousands, and "s" to seconds.

4.3 Rasterizer Load

This section reports the amount of polygon scan-conversion done by Mesa and IMR (RC performs no scan-conversion). Values for back-to-front rendering are identical to those for unordered rendering. We do not report rasterizer efficiency (in terms of number of samples rejected by the depth test), as Mesa and RC are roughly equivalent in efficiency for a given rendering order; that is, if the actual tessellated object passes the depth test, the tessellated bounding volume (though it is slightly larger) also passes the depth test in the vast majority of cases.

Model	Mesa	IMRBIS	IMRBIS savings
Rings	450K	93K	4.8x
Tree	1.7M	330K	5.15x
Shell, lo	2.5M	460K	5.4x
Shell, hi	4.1M	460K	8.9x

Table 1: Number of triangles sent down pipeline

Table 1 shows that at high resolution, Mesa required more triangles to render the shell model so as to maintain a curved appearance for the bigger spheres in the scene. Note that IMR processes a significantly lower number of triangles, since it requires only that the triangles describe conservative bounds around each curved object.

Model	Mesa	IMR	IMR increase
Rings, lo	17K	20.6K	21%
Rings, hi	289K	331.5K	15%
Tree, lo	1131	1436	27%
Tree, hi	22.7K	27.5K	21%
Shell, lo	631.2K	726.9K	15%
Shell, hi	10.2M	11.5M	12.7%

Table 2: Number of pixels covered by triangles per frame.

The number of pixels covered by the conservative bounds, while higher than that for the tessellated objects, is not significantly higher (see Table 2); this reflects the decreasing return in terms of accuracy as the number of triangles per object increases. The "sweet spot" for bounding volume tessellations (where added triangles pay off the most in tighter bounding volumes) is far lower than the object tessellation required to get the accuracy provided by raycasting, as the data in §4.9 will demonstrate.

4.4 Raycasting Load

Table 3 reports the raycasting load for each algorithm, as measured by the number of ray-bound tests, the number of ray-object tests, the number of ray-cell walks, and the success rate of the ray-object tests. We group the ray-cell walks with the other ray-object tests because each walk includes a ray-bound test against a cell boundary.

Model	RC total (ray-bound, ray-object, ray-cell walks)	IMR ray-object	IMR ray-object savings	IMR total savings
Rings,lo	897K (800K, 25K, 72K)	7.1K	3.5x	126.3x
Rings,hi	14M (12.5M, 386K, 1.1M)	114K	3.3x	122.8x
Tree,lo	403K (291K, 2.8K, 109K)	1216	2.3x	331.4x
Tree,hi	6.3M (4.6M, 46K, 1.7M)	22K	2.1x	286.3x
Shell,lo	5.5M (4.8M, 700K, 7.7K)	295K	2.4x	18.6x
Shell,hi	86M (75M, 11M, 121K)	4.6M	2.4x	18.6x

Table 3: Ray-object intersection tests.

Use of a spatial data structure and front-to-back ordering decreased the number of ray-object intersections by approximately 10% in all cases. Note that IMR performs significantly fewer ray intersections than the optimized raycaster. IMR does not use standard cell-walking; however, when rendering front-to-back, it does initialize and traverse a $k-d$ tree [6]. IMR does not test bounding-boxes for ray intersection; rather, it scan-converts each bounding volume. The third column, ray-object savings, illustrates the effect of IMR's more complex bounding volumes over RC's axially-aligned bounding boxes. The fourth column shows the overall work reduction achieved by IMR.

4.5 Raycasting Efficiency

The following table (Table 4) shows the number of ray-object hits compared to the number of ray-object intersection tests. IMR, because of its polygonal bounding volumes, is far more efficient with its tests. Where RC finds a ray-object intersection as few as one in thirty times, IMR finds an intersection at least five in six times, and often nearly nineteen out of twenty times.

Model	RC	IMRBIS
Rings,lo	3.13	93.8
Rings, hi	3.08	94.1
Tree,lo	33.4	87.8
Tree, hi	32.3	85
Shell,lo	19.1	87.8
Shell, hi	19.1	88.7

Table 4: Percentage of ray-object hits vs. tests

4.6 Lighting Load

Table 5 reports the amount of *lighting* work performed. We chose this metric because the less work done lighting, the more effective the visible surface technique. Note that RC exhibits minimal lighting load, as it defers lighting until after visibility determination.

Model	RC	Mesa (vs. RC)	IMRBIS (vs. RC)
Rings,lo	4.5K	1.3M(288.9x)	6.7K(1.5x)
Rings, hi	72K	1.3M(18.0x)	107K(1.48x)
Tree,lo	899	5.9M(6562.8x)	1058(1.17x)
Tree, hi	14.3K	5.9M (412.6x)	19K(1.33x)
Shell,lo	2.9K	7.6M(2620.7x)	262K(90.34x)
Shell, hi	45K	12.5M(277.8x)	4.1M(91.1x)

Table 5: Number of fragment lighting operations performed by Mesa, RC and IMBRIS

However, since IMR lights only those fragments which pass the depth test, it incurs far fewer lighting operations than Mesa, which unconditionally lights every vertex in the scene. That is, Mesa wins only when there are a small number of large polygons; i.e., for geometrically simple scenes. Finally, we observe that the number of lighting calculations incurred by IMR decreased 5-10% given front-to-back ordering, matching the decrease in ray-object tests as shown in previous tables. Again, front-to-back ordering does not affect Mesa, and is not applicable to RC.

4.7 Overall Rendering Speed

This section (see Table 6) reports the frame time of each algorithm, measured by repeatedly drawing the scene from a fixed viewpoint and calculating the average time per frame. With one exception, IMR is

Model	RC	Mesa	IMRBIS
Rings, lo	2.8	3.5/3.9	.75/1.1
Rings, hi	44.4	4.4/8.2	2.3/6.1
Tree, lo	1.7	15.8/15.5	2.8/2.4
Tree, hi	28	15.8/19.6	3.4/6.9
Shell, lo	11.5	11.6/11.6	4.1/4.6
Shell, hi	167.2	39/44	34/33

Table 6: Effective frame time, in seconds (front-to-back/unordered).

two to twenty times faster than traditional ray casting (RC) and two to five times faster than standard scan-conversion (Mesa). In the latter case, IMR achieves the speedup despite producing more accurate geometry and lighting.

The exception for RC was the tree model at low resolution, where the traditional raycaster was 30% faster than IMR. In this case, most of the primitives had very small pixel footprints (taking away the screen coherence exploited by scan-conversion) and most of the scene was occluded, allowing RC to take maximal advantage of its spatial data structure. On the other hand, even in this case case, IMR without a spatial data structure is still competitive with RC.

The exception for Mesa was the shell rendering at high resolution; here IMR was only 10% faster than Mesa. In this case, the spheres were piled atop one another, making the bounding volume depth test less effective in reducing raycasting effort. Even in this worst case, however, IMR is still faster (and more accurate) than classical scan-conversion.

4.8 Resolution Sensitivity

This section tabulates the slowdowns incurred by each method when the framebuffer resolution was increased by a factor of sixteen.

Model	RC ray-object	RC total	IMRBIS (front-to-back/unordered)
Rings	15.4	15.6	15.8/16.1
Tree	16.4	15.6	16.9/18.1
Shell	15.7	15.6	19.8/15.5

Table 7: Factor of raycasting work increase when image size increased by 15.8 (129x129 to 513x513).

The data in Table 7 shows that the workload of both IMR and RC increases nearly linearly with window size.

Model	RC	Mesa(front-to-back/unordered)	IMRBIS (front-to-back/unordered)
Rings	15.85	1.25/2.1	3.06/5.5
Tree	16.4	1.0/1.26	1.20/2.87
Shell	14.5	3.36/3.79	8.29/7.17

Table 8: Factor of slowdown when image size increased by 15.8 (129x129 to 513x513).

However, this does not carry over into overall frame time. The data in Table 8 demonstrates that IMR scales far better with screen resolution than RC, even though both demonstrate a linear relation in the amount of raycasting work required. This is due to IMR's hybrid nature. It spends only a fraction of its time raycasting; the rest is spent rasterizing. The rasterization time, as demonstrated by the Mesa results, is nearly independent of screen resolution (and, due to the lower polygon count and disabling of lighting, is far lower than Mesa's time). Thus only a fraction of IMR's time complexity increases linearly with screen resolution. It does not scale as well as standard scan-conversion; this is to be expected since IMR must do more raycasting, while standard scan-conversion need only interpolate more interior pixels, a relatively cheap operation. On the other hand, this cost has its benefits. IMR continues to provide sharp curves (both in the silhouette and in the internal geometry) at the greater resolution, while curves in Mesa degrade (only sharp polygon edges are maintained). Mesa fares worse in the case of the Shell model where the object tessellation was increased for Mesa in order to remove highly visible artifacts of linearity at the higher resolution. This explains the jump in Mesa's slowdown factor for the Shell model.

4.9 Bounding Volume Complexity

An important factor in the speed of this algorithm is the tightness of the bounding volume. This section examines the tradeoff between amount of tessellation and pixel accuracy (measured as a ratio of screen footprint sizes: the actual object's divided by the bounding volume's) for spheres and cylinders. The "tessellation values" for tables 9 and 10 are the number of slices and stacks for the spheres and the number of slices for cylinders. The objects were tessellated using the same algorithm as OpenGL's gluSphere and gluCylinder functions. The radius for a bounding object was precomputed by generating a tessellation of the given complexity at an arbitrary radius, then calculating the closest distance of any point on the generated surface to the center (the center point of the sphere or the central axis of the cylinder). The arbitrary radius was then scaled by the ratio of the closest distance to the desired radius. This resulting radius, when passed to gluSphere or gluCylinder, is the smallest possible one that still bounds the object. In these tables, the tessellation value is on the vertical axis; the implicit object's screen footprint is on the horizontal axis.

Tesselation value	113 pixels	2933 pixels	13457 pixels
4	77.9	77.4	81.56
7	87.59	87.7	89.19
10	90.4	95.2	94.55
15	96.58	97.2	97.01
20	100	98.92	98.3
25	98.26	99.52	99.23

Table 9: Accuracy of bounding volume silhouette for a sphere.

Tesselation value	135 pixels	6707 pixels	14339 pixels
4	63.28	73.24	91.93
7	100	93.55	93.4
11	98.4	95.2	95.1
15	100	97.44	99.15

Table 10: Accuracy of bounding volume silhouette for a cylinder.

The data shows that the benefit per triangle falls steadily as the number of triangles increases. Even at very low tessellation values, the accuracy rate is around 90% and only weakly dependent on the pixel size of the object, which suggests that low tessellation values are ideal for the fastest rendering of visible objects with IMR. This number can also be interpreted, from a traditional scan-conversion perspective, as the silhouette error for a given tessellation. While 90% is unusually high as a raycasting hit rate, it is not as good a figure for visible silhouette error.

We did not instrument *depth* error produced by the algorithms, though doing so would further demonstrate IMR's advantages. IMR calculates depth exactly at each sample; scan-conversion does not. Thus IMR renders both lighting and geometry (e.g., interpenetrating curved surfaces) more accurately than does Mesa.

5 Conclusion

This paper describes a hybrid algorithm which combines rasterization and ray casting to produce an efficient, immediate-mode algorithm suitable for expression in hardware. There are at least two aspects of the method that we do not address. First, though our algorithm uses polyhedral bounding volumes, we leave the method for their generation unspecified. Similarly, our algorithm depends on polynomial root-finding in its innermost loop. As improved algorithms for both sub-tasks emerge, they can be incorporated straightforwardly into our algorithm.

By utilizing bounding volumes, we believe raycasting renderers can become not only interactive but superior in performance to polygon renderers, exploiting fast rasterization capability to provide a useful visibility test (rather than simply a polygon fragment generator). We described an algorithm that combines the strengths of both rendering methods: the speed of a polygon rasterization pipeline, and the generality and precision of raycasting. Its key step is to feed tessellated bounding volumes to a Z-buffered scan-converter, implicating tight sets of rays to be intersected with each object.

Our algorithm is efficient, exact, general, and amenable to hardware implementation. It uses no spatial index or bounding hierarchy, and so necessarily touches every scene object while rendering. However, it should be suitable for rendering large scenes quickly in hardward. We propose that this hybrid scan-conversion and raycasting algorithms find adoption in a next-generation graphics workstation, to enable the rendering of curved primitives at interactive rates while freeing processor and connection bandwidth for other purposes.

References

[1] Akeley, K. The Silicon Graphics 4D/240GTX superworkstation. *IEEE Computer Graphics and Applications 9*, 4 (July 1989), 71–83.

[2] Akeley, K. RealityEngine graphics. *Computer Graphics 27*, Annual Conference Series (1993), 109–116.

[3] Akeley, K., and Jermoluk, T. High-performance polygon rendering. *Computer Graphics 22*, Annual Conference Series (1988), 239–246.

[4] Appel, A. Some techniques for shading machine renderings of solids. In *Proceedings of SJCC* (1968), Thompson Books, Washington, D.C., pp. 37–45.

[5] Baumgardt, B. Winged-edge polyhedron representation. Tech. Rep. No. CS-320, Stanford Artificial Intelligence Report, Computer Science Department, 1972.

[6] Bentley, J. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18* (1975), 509–517.

[7] Blinn, J. F. A scan line algorithm for displaying parametrically defined surfaces. *Computer Graphics 12*, 3 (Aug. 1978), 27–27.

[8] Cameron, S. Approximation hierarchies and s-bounds. In *Proc. of 1991 ACM/Siggraph Symposium on Solid Modeling Foundations and CAD/CAM Applications* (1991), pp. 129–137.

[9] Catmull, E. E. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, University of Utah, Dec. 1974. Also TR UTEC-CSc-74–133, CS Dept., University of Utah.

[10] Clark, J. H. Hierarchical geometric models for visible surface algorithms. *CACM 19*, 10 (1976), 547–554.

[11] Clark, J. H. The geometry engine: A VLSI geometry system for graphics. *Computer Graphics 16*, 3 (July 1982), 127–133.

[12] Ellis, J. L., Kedem, G., Lyerly, T. C., Thielman, D. G., Marisa, R. J., Menon, J. P., and Voelcker, H. B. The raycasting engine and ray representations: A technical summary. *Internat. J. Computational Geometry and Appl. 1*, 4 (1991), 347–380.

[13] Fuchs, H., Kedem, Z., and Naylor, B. On visible surface generation by a priori tree structures. *Computer Graphics (Proc. Siggraph '80) 14*, 3 (1980), 124–133.

[14] Glassner, A. S. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*, 10 (1984), 15–22.

[15] Greene, N., Kass, M., and Miller, G. Hierarchical Z-buffer visibility. In *Proceedings of Siggraph '93* (Aug. 1993), pp. 231–238.

[16] Haines, E. A. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications 7*, 11 (Nov. 1987), 3–5. also in SIGGRAPH '87, '88, '89 Introduction to Ray Tracing course notes, code available via FTP from princeton.edu:/pub/Graphics.

[17] Heckbert, P. The mathematics of quadric surface rendering and SOID. Tech. Rep. 3-D Technical Memo No. 4, New York Institute of Technology, July 1984.

[18] Isakovic, K. 3D Engines List DOS & Doom/Wolfenstein, `http://cg.cs.tu-berlin.de/~ki/3de_hard_dos_doom.html`. Tech. rep., 1997.

[19] Jansen, E., and Chalmers, A. Realism in real time In *Fourth Eurographics Workshop on Rendering* (June 1993), M. F. Cohen, C. Puech, and F. Sillion, Eds., Eurographics, pp. 27–46. held in Paris, France, 14–16 June 1993.

[20] Kay, T. L., and Kajiya, J. T. Ray tracing complex scenes. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (Aug. 1986), D. C. Evans and R. J. Athay, Eds., vol. 20, pp. 269–278.

[21] Kurt Akeley, S. G. I. `capping.c`, demonstration of gl/opengl rendering of capped solids, 1991.

[22] Montrym, J. S., Baum, D. R., Dignam, D. L., and Migdal, C. J. InfiniteReality: A real-time graphics system. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 293–302. ISBN 0-89791-896-7.

[23] Muuss, M. J. *Workstations, Networking, Distributed Graphics, and Parallel Processing.* Springer-Verlag, 1990.

[24] Muuss, M. J. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95* (1995).

[25] Neider, J., Davis, T., and Woo, M. *OpenGL Programming Guide.* Addison-Wesley, 1993.

[26] Phong, B.-T. Illumination for computer generated pictures. *Communications of the ACM 18*, 6 (June 1975), 311–317.

[27] Pineda, J. A parallel algorithm for polygon rasterization. *Computer Graphics (Proc. Siggraph '88)* (1988), 17–20.

[28] Potmesil, M., and Hoffert, E. M. The pixel machine: A parallel image computer. J. Lane, Ed., vol. 23, pp. 69–78.

[29] ROSSIGNAC, J., MEGAHED, A., AND SCHNEIDER, B.-O. Interactive inspection of solids: Cross-sections and interferences. *Computer Graphics 26*, 2 (July 1992), 353–360.

[30] RUBIN, S. M., AND WHITTED, T. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*, 3 (July 1980), 110–116.

[31] SCHUMACKER, R. A., BRAND, B., GILLILAND, M., AND SHARP, W. Study for applying computer-generated images to visual simulation. Tech. Rep. AFHRL TR-69-14, U.S. Air Force Human Resources Laboratory, 1969.

[32] SILICON GRAPHICS, I. O^2 unified memory architecture. Tech. Rep. Whitepaper 1352, Silicon Graphics, Inc., 1997.

[33] WHITTED, T. An improved illumination model for shading display. *CACM 23*, 6 (1980), 343–349.

[34] WINGET, J. Advanced graphics hardware for finite element results display. *Advanced Topics in Finite Element Analysis, PVP 143* (June 1987).

[35] WITKIN, A. P., AND HECKBERT, P. S. Using particles to sample and control implicit surfaces. *Computer Graphics 28*, Annual Conference Series (July 1994), 269–278.

[36] WYLIE, C., ROMNEY, G., EVANS, D., AND ERDAHL, A. Half-tone perspective drawings by computer. In *Proceedings of AFIPS 1967 FJCC* (1967), vol. 31, pp. 49–58.