MIT Laboratory for
Computer Science

Massachusetts
Institute of
Technology

MIT LCS TR-740

Frustum Casting for Progressive, Interactive Rendering

Seth Teller John Alex

MIT Computer Graphics Group

January 1998

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

Frustum Casting for Progressive, Interactive Rendering

Seth Teller*
MIT Computer
Graphics Group

John Alex
Brown Computer
Graphics Group

Abstract

Efficient visible surface determination algorithms have long been a fundamental goal of computer graphics. We discuss the well-known ray casting problem: given a geometric scene description, a synthetic camera, and a viewport which discretizes the camera film plane into pixels, ray casting identifies the visible surface at each pixel, i.e., that scene primitive which is first encountered by an eye ray directed through the pixel center.

Interactive rendering systems have not ordinarily been based on ray casting, due to its computational cost. Instead, the dominant method for achieving interactive rendering is hardware-assisted rasterization and depth buffering of polygons, often produced by static or dynamic tesselation of higher-level objects. Modern polygon rasterization architectures are extremely powerful, having undergone an extensive development path.

Several trends indicate, however, that alternatives to polygon rasterization and depth buffering deserve examination in the design of future interactive rendering systems. The first trend is the number and breadth of proposed algorithmic and hardware methods to lessen transformation, rasterizer and depth buffer load while viewing models of high complexity. A second, related trend is that geometric models are, increasingly often, larger than ordinary physical memories, lending greater importance to memory coherence considerations. Finally, general purpose processors have grown very powerful, enabling flexible, dynamic retargeting of computational resources to differing subtasks while maintaining responsiveness. A rendering system based on such processors could have significant advantages over dedicated hardware.

In light of the above, we explored an alternative, general rendering architecture based on ray casting. In seeking to build an interactive software ray caster, we studied existing visible surface algorithms. Combining three such algorithms, we synthesized frustum casting, a novel algorithm for per-pixel visible surface identification in general scenes. The algorithm samples discretely, but operates in object space, and is exact and efficient. We demonstrate a prototype software renderer based on frustum casting, which achieves interactivity through "just-in-time" sampling, and progressive image improvement through deferral of intersection and shading operations.

Frustum casting well addresses the technological trends listed above. We believe that it and other ray-based rendering methods may be practically incorporable by designers of future high-performance rendering architectures.

Keywords: Rendering, rasterization, ray casting, ray tracing, visibility, occlusion, spatial indexing.

*We gratefully acknowledge the support of Intel Corporation, and of MURI Award SA 1524-258 2386. The opinions and conclusions expressed in this document are not necessarily those of any sponsoring company or agency.

1 Introduction and Related Work

Achieving visually correct, interactive viewing of complex, general geometric scenes has long been a fundamental challenge in computer graphics. Early rendering systems were based on software implementations of "hidden-surface elimination" algorithms for polygonal scenes [28]. Given a scene description, these elegant algorithms identified and displayed those portions of the scene visible to a specified synthetic camera. Importantly, these algorithms compute analytic (i.e., object-space) descriptions of visible surfaces and surface fragments. In addition to a method for hidden-surface elimination, rendering systems formulate a shading model in order to display visible scene elements with simulated illumination values.

1.1 Depth-Buffered Rasterization Architectures

Early hidden-surface algorithms expend considerable effort computing descriptions of visible fragments in object space (or some hybrid of object and screen space, e.g., [30]). This can be wasteful when visibility information is required only for a finite set of samples at each pixel. Thus these algorithms are not ordinarily competitive for very complex, or very general, environments. The z-buffer, proposed in [9], 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 size; as available memory grew and cheapened, depth buffering gained favor over object space methods. Specialized hardware "geometry engines" eventually emerged to accelerate the extensive calculations inherent in polygon transformation, clipping, shading and rasterization [10]. 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., [4, 2, 3, 25, 27]).

1.2 Managing Geometric Complexity

Graphics workstations dedicate substantial hardware "pipelines" to geometric transformations, polygon clipping, application of shading and texturing operations, anti-aliasing and depth-buffering. However, geometric models have grown ever more complex as well, due to CAD, military, scientific, entertainment and other applications. In a curious twist, object-space and hybrid object/screen-space methods have again regained prominence as researchers attempt to maintain responsive visible-surface identification and rendering in the face of increasing model complexity. In effect, just as visible-surface algorithms once turned to low-level depth buffering hardware to avoid combinatorial blowup, so now do hardware pipelines turn to high-level visible-surface algorithms to avoid rendering overload!

For input models with considerable occlusion, from most viewpoints a significant fraction of scene primitives do not contribute to the rendered image. Such models are said to have high "depth complexity," resulting in "overdraw" – wasted effort[1] – during rendering.

Efficient algorithms to identify and render only potentially visible scene elements, thus reducing overdraw, appear in several forms. Frustum culling algorithms cull those scene elements lying outside the view frustum, but do not detect occlusion [17]. Occlusion culling algorithms exclude portions of the scene invisible due to nearer, opaque elements; these algorithms operate in object space [1, 29, 11, 22] or in both object and image space [20, 24, 19, 32]. All of these algorithms rely on standard rasterization hardware for rendering, and a standard or generalized screen-space depth buffer to resolve occlusion among potentially visible objects.

[1] The term "overdraw" is used in the game engine community to mean repeated reads and conditional writes of the depth buffer. Here we use it to mean any processing of invisible scene elements.

1.3 Alternative Frameworks for Interactive Rendering

Despite both algorithmic advances and specialized hardware, several trends open to question the suitability of polygon rasterization for rendering ever more complex geometric scenes. First is the number and breadth of recently proposed methods intended to lessen rendering load when viewing complex models. This suggests that perhaps the standard approach of transforming, clipping, rasterizing, and depth buffering simply does not scale well; perhaps some fundamentally different visibility resolution mechanism should be sought.

A second, related factor is the increasing importance of algorithmic memory coherence as geometric model size routinely exceeds the fast storage capacity of the workstations with which they are to be viewed. Indeed, one can reasonably expect geometric models to grow so complex that even related spatial indexing data (e.g., octree structures) will not fit in physical memory. The care with which algorithms manage access to this spatial index will become an increasingly important, and perhaps decisive, evaluation criterion. This has already occurred in the GIS community, although so far mostly for two-dimensional data [6]. We defer a discussion of algorithmic working set considerations to §1.5.

Finally, general purpose processors have grown enormously powerful. Most desktop workstations make available several hundred MIPS and MFLOPS for general purpose computing. This enables flexible, dynamic retargeting of compute bandwith to differing rendering subtasks (visibility, lighting, texturing, reconstruction, etc.), or tasks entirely apart from rendering (e.g., simulation), possibly to significant advantage over dedicated hardware.

1.4 An Interactive Ray-Based Rendering Algorithm

This paper proposes the viability of responsive, high-fidelity rendering algorithms based on object-space ray casting. We introduce an algorithm, frustum casting, with a number of attractive properties in light of the above criteria. It uses neither dedicated geometric transformation hardware nor depth buffering (though it could be accelerated by the former). It has a working set equivalent or smaller than that of existing algorithms in principle, yet handles more general scene geometries. Finally, within a few years, increasing general compute bandwidth will enable ray-based rendering systems to run interactively on most desktop systems without dedicated graphics hardware. The frustum casting implementation we describe may serve only to provide interactivity slightly earlier. Even so, we have engineered it to adapt gracefully (while sacrificing or augmenting image fidelity) to time-varying, insufficient, or extra compute bandwidth on today's systems.

1.5 Algorithm Characterization by Working Set and Overdraw

As the complexity and data size of ordinary models rises, an increasingly important consideration in the evaluation of visible surface algorithms is each's "working set"; that is, the extent to which, and the order in which, each algorithm accesses virtual memory regions containing nodes of the spatial index (e.g., geometric bounds) or model data (e.g., scene primitives such as triangles or other objects). Working set is closely related to "overdraw," the effort expended to render (transform, light, clip, rasterize, depth buffer; intersect rays with; etc.) surfaces which do not contribute to the rendered image.

When the visible scene geometry is larger than physical memory, researchers have proposed substituting simplified geometry [15] or imagery [13]. Such methods have complex implications for memory footprint, and are outside the scope of this discussion of visible surface algorithms.

When only a portion of the scene is visible from ordinary viewpoints, an effective algorithm could reduce overdraw by identifying and traversing only this portion. A perfect visibility "oracle" (as yet undiscovered) could do so without touching invisible scene elements, thus eliminating overdraw entirely. For specialized environments, effective oracles exist through "preprocessing" [1, 29, 16]. The principal strength of these methods is their capability to schedule physical and virtual memory accesses both before and during rendering, avoiding most sudden losses of interaction due to traversal of out-of-memory scene elements. However, such methods are so far applicable only to those architectural models whose room and adjacency structure is fairly regular, and readily discernible.

The aim of this section is to categorize a wide variety of visible-surface algorithms according to the above considerations. Although it is difficult to deduce the detailed memory behavior of so many complex algorithms without extensive experimentation, it is possible to categorize them by their per-frame memory footprint with regard to both nodes of the spatial index (if appropriate) and associated geometric scene elements. We do not consider classical object-space visible surface algorithms, as their combinatorial complexity makes them impractical for very complex environments.

Algorithms can also be characterized by the degree to which they result in overdraw. Our characterization of visibility algorithms requires the definition of the "occlusion frontier" of a frustum (the bold curves in Figure 1). The occlusion frontier is defined, for any ray emanating from the viewpoint, as the depth at which the ray first intersects a scene element. A point in the frustum lies "beyond" the occlusion frontier if the line segment joining that point and the viewpoint intersects the occlusion frontier. Finally, we say that a volume is beyond the occlusion frontier if every point of the volume lies beyond the frontier. Using the definitions above, four classes of visibility algorithm can be identified (Figure 1):

Exhaustive Traversal. In hardware, all model primitives are transformed, clipped, and rasterized; visibility is determined with a depth buffer. In ray casting, every ray is checked for intersection with every object. Either algorithm plainly touches all model data (Figure 1-a), so does not scale well to complex models, and results in extensive overdraw.

Frustum Culling. The spatial index is traversed from the root. Recursively, cells are examined for intersection with the view frustum [17]. Disjointness causes a cell's associated scene data and children (if any) to go unexamined; incidence causes traversal of the cell's children (if any) and contents. Frustum culling algorithms process nodes outside the view frustum and beyond the occlusion frontier, and scene geometry beyond the occlusion frontier, resulting in extensive overdraw (Figure 1-b).

Occlusion Culling. A collection of occluders near the viewpoint is identified either offline or during rendering, and maintained either in object space [11, 22] or (hierarchically and conservatively[2]) in screen space [20, 19, 32]. The spatial index is traversed from the root. If a cell is occluded by the current occlusion map, its children are unexamined. Otherwise, its associated geometric data and children are traversed, possibly updating the visibility map. These methods traverse significant numbers of nodes, and some scene elements[3], beyond the occlusion frontier (Figure 1-c)[4], resulting in significant overdraw.

Frustum Advance. The spatial index is traversed starting from a leaf cell, by adjacency, and constrained so as to maintain incidence with the view frustum. Examples of frustum advance algorithms are (for conservative interactive viewing of polyhedral, densely occluded, typically architectural environments) [23, 1, 16, 24], and (for analytic, usually batch rendering of general environments) ray casting with spatial indexing (e.g., [18, 14]). This class of algorithm has a nearly minimal memory footprint; it touches no nodes beyond the occlusion frontier (Figure 1-d). The algorithm we introduce falls into this category.

[2]The algorithm of [32] is not conservative; that is, it may omit visible polygons.

[3]The scene in [19] was procedurally instanced, avoiding the problems inherent in global scene traversal.

[4]In fact each of [20, 19, 11, 22, 32] uses the root-first traversal order of [17]. However each algorithm is easily modified to touch only those nodes incident on the viewing frustum, by adoption of constrained depth- or breadth-first traversal.

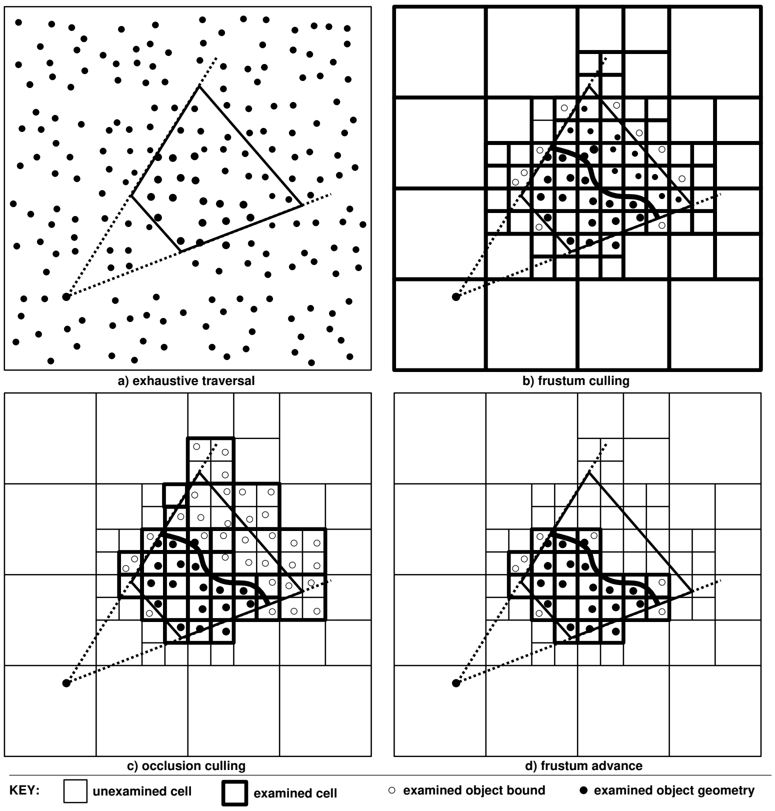


Figure 1: Memory footprints of four classes of visibility algorithm.

1.6 Discussion

Consider the latter two classes of algorithms: occlusion culling and frustum advance. Of occlusion culling algorithms, only [20, 19, 32] are likely to scale well to large environments, and then only if they adopt the spatial traversal ordering (i.e., eye outward, constrained to within the frustum) of frustum advance methods.

Now consider the *overall* processing performed by a hardware rasterization system when augmented by any of the proposed algorithms of [20, 19, 32], modified to adopt frustum advance traversal. Such systems would

- Traverse the spatial index eye outward, processing only nodes incident on the view frustum;
- Keep track of scene elements, and their depths, as they are encountered;
- Perform repeated (possibly hierarchical), conservative depth comparisons at each pixel to determine the visible surface there.

These three properties apply as well to ray casting algorithms. In other words, the above visible surface methods, whether formulated as object-space or hybrid algorithms, *strongly resemble ray casting algorithms*. We consider this development – that current techniques are converging to ray casting – strong evidence that ray casting deserves examination as a design alternative for interactive rendering systems.

2 Preliminaries

Frustum casting is a novel synthesis of three well-known algorithms: screen-space subdivision [30], beam tracing [21]; and fast ray walking through a spatial subdivision [18]. Each of these algorithms introduced a significant innovation to yield efficiency in an important problem domain. Yet, each has a significant computational disadvantage, as we describe. Frustum casting combines efficient aspects of the three algorithms.

2.1 Screen-Space Subdivision

Warnock's hidden-surface elimination algorithm operates as follows [30]. Given a root (screen space) viewport and a collection of polygons, all polygons are transformed to screen space (retaining depth) and classified as disjoint, covering, or incident on the root viewport. The viewport is then recursively subdivided in quadtree fashion, under the following criteria: subdivision terminates if either 1) no polygon intersects the viewport; 2) some polygon covers the viewport, and its maximum depth within the viewport is less than the minimum depth of all other polygons, or 3) the viewport has been subdivided to a single pixel (in which event special-case processing emits visible vertices and edge fragments, or simply sorts by depth at the pixel center, depending on application). For efficiency, polygon incidence lists are "inherited" by child viewports during subdivision.

The significant innovation of Warnock's algorithm is its introduction of a simple test (of a polygon versus a viewport edge) which, when successful, shields many thousands of pixels (on one side of the edge) from any interaction with the excluded polygon (on the other side). However, the algorithm has a significant disadvantage: to initialize the root viewport, it must classify (e.g., process) every polygon in the scene, even those polygons eventually determined to be invisible.

2.2 Beam Tracing

Heckbert and Hanrahan's beam tracing algorithm was formulated for ray tracing polyhedral environments, and operates as follows [21]. A root frustum is defined to encompass the entire viewport, and represents an initial visibility "beam" emanating from the eye. A sweep plane algorithm traverses polygons in depth order[5], and "subtracts" them from the beam (using generalized intersection operations [31]). The masked portion of the beam is then reflected from the occluding polygon and recursively propagated through the scene.

Beam tracing's significant innovation is its use of polyhedral beams to advance many rays through the scene at once, attaining significant acceleration of ray tracing. However, in the balance it surrenders the ability to handle non-polygonal primitives. Moreover, it necessarily computes all apparent (visible) intersections among polygon edges, effectively subdividing along all such edges. Its running time can therefore grow quadratically with scene complexity, even in the non-recursive (i.e. ray casting) case.

2.3 Ray Walking

Glassner's octree-based ray-walking algorithm provided for the efficient propagation of rays through an axial spatial index [18]. This algorithm's significant innovation was to localize ray-object intersection tests; rather than involving all scene primitives, ray-walking limits processing to the set of spatial cells

[5] The algorithm assumes no depth cycles.

between the ray origin and its eventual encounter with a scene primitive (or the background), and any scene elements associated with those cells.

Ray-walking significantly accelerates a fundamental operation: tracing a *single* ray through a complex scene. Indeed, it exploits coherence by excluding most sets of related scene elements from interaction with a given ray. However, it does not address the complementary problem, that of excluding most sets of related *rays* from interaction with most scene elements. That is, ray-walking does nothing to accelerate the tracing of successive, *similar* rays through the spatial index, even though their paths (and thus the set of objects against which they are tested for intersection) will generally be similar.

In ray tracing, this is perhaps a minor concern, as the origin and direction of recursively spawned rays decoheres quickly due to reflection and refraction operations, and such rays form the great majority of the total rays traced. Thus, the time "wasted" by failing to exploit the coherence of primary rays is generally insignificant in relation to total time. Yet ray *tracing* is at present a domain in which image generation may require minutes or hours. Here our goal is the realization of ray *casting*, of primary rays only, in a fraction of a second; the failure of individual ray-walking to exploit (the significantly greater) coherence present in this restricted situation is a significant weakness of this approach.

2.4 Frustum Subdivision

We implemented an exploratory ray casting algorithm which combined Warnock's algorithm and beam tracing (but not ray-walking), as follows. Given a root frustum and a collection of scene objects, all object bounds are classified as incident or disjoint with respect to the frustum in object space, that is, by classifying the objects' 3D bounding boxes with respect to each of the frustum bounding planes. The root frustum is then recursively subdivided under the following termination criteria: subdivision terminates if either 1) no element intersects the current frustum; 2) some element covers the frustum, and its maximum depth within the frustum is less than the minimum depth of all other elements, or 3) the frustum has been subdivided to a single pixel. Incidence information for children is computed by the parent frustum and shared as in Warnock's algorithm. In termination case (1), the relevant pixels are trivially shaded, e.g. with the background color. In termination case (2), the visible element is known at every pixel in the region; for shading, either its intrinsic color can be used, or (if a depth and/or surface normal is needed) a fast (unconditional) ray-object intersection can be applied to the element. In termination case (3), an ordinary ray-object intersection is performed on all incident objects, and the closest intersection returned as usual.

This "frustum subdivision" algorithm for ray casting significantly outperforms octree-based ray-walking for scenes of low depth complexity. Yet, for scenes of high depth complexity, it has the same disadvantage as Warnock's algorithm: it traverses and classifies all objects "at the root," thereby touching the entire scene, and potentially wastes significant effort on invisible objects. Frustum subdivision is therefore unsuitable for application in complex environments.

3 Frustum Casting

Recently, several practically-motivated criteria have been proposed for visibility culling algorithms: that such algorithms be general, run at interactive rates, provide effective culling, be amenable to implementation, and scale well to complex models [32]. Here we suggest a somewhat more theoretically motivated set of design considerations germane to crafting visible surface algorithms. Such algorithms should:

- Spend little or no effort considering occluded objects, thus exhibiting a working set not much greater than the set of visible objects (and those cells that index them, if any);

- Classify at once multiple pixels with respect to most objects, and multiple objects with respect to most pixels, thus exploiting the geometric coherence of primary rays, and the spatial coherence of scene geometry (when present); and
- Perform no subdivision along object space silhouettes, or at object space vertices (apparent intersections of silhouettes), thus avoiding combinatorial blowup.

We propose a novel algorithm which, in principle, satisfies both sets of criteria. That it indeed scales to interactive rates on complex scenes is not yet proven, although initial results are encouraging (§5).

3.1 Data Structures and Subdivision

The frustum casting method depends on the following data structures. Before interaction begins, the scene is organized into a hierarchical spatial index, as in [18], which supports efficient population of scene objects and location of the synthetic viewpoint. The spatial index also supports efficient traversal by propagating rays, and determination of a given ray's "arrival cell" – the adjacent cell into which a ray, exiting a given cell, emerges. (If there is no such adjacent cell, the cell descriptor is returned as a special value, NULL.)

Another data structure of the method is a frustum descriptor (Figure 2), defined by a point of view, four extreme rays centered on viewport pixels (lower left, lower right, upper right, and upper left) and four bounding planes (left, right, bottom, and top). Each bounding plane defines a "positive halfspace" in world coordinates; the intersection of these four halfspaces is the spatial "interior" of the frustum.

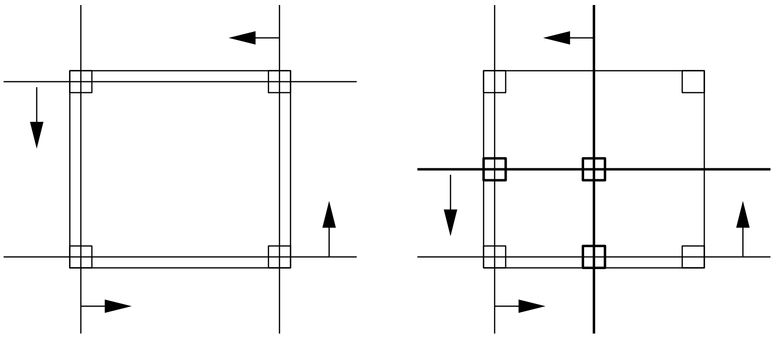


Figure 2: A frustum descriptor (left), and frustum subdivision (right).

A frustum can be subdivided into four "child frusta," each one quarter the size of the original frustum, through the computation of two new bounding planes (one separating left from right, one separating bottom from top) to serve as boundary planes of the child frusta, and five rays (one in the viewport center, and one along the midpoint of each edge) to serve as extreme rays of the child frusta. Whenever a frustum is divided into child frusta, the child frusta can reuse all of the information contained within the parent, and share information assembled by the parent. For example, each child frustum reuses two bounding planes (appropriately oriented) and one extreme ray with the parent. Each child shares both of the bounding planes generated by the parent, and three of the five rays generated by the parent (shared elements are bold in Figure 2).

3.2 Recursive Description

The frustum casting method is most simply described as a recursive algorithm, although our implementation uses non-recursive continuations (§5). Each instance of the algorithm is invoked with a frustum descriptor, and a cell descriptor which describes the cell in which the frustum's extremal rays are to commence traversal. A "root frustum" is initialized, to which correspond all pixels in the viewport; the current frustum is then set to the root frustum, and the current cell to the cell containing the viewpoint (Figure 3-a shows a two-dimensional example with a nine-pixel viewport).

The set of elements in the current cell whose bounding volumes impinge on the frustum interior is then determined. If the set of impinging elements is empty, there can be no ray/element intersections in the current cell (as in Figure 3-a). In this case, the arrival cell for each extreme ray is determined. If all extreme rays have the same arrival cell, there are two cases. If the arrival cell is NULL, the frustum has exited the spatial index, and no element intersections exist for rays within the current frustum (Figure 3-b). Otherwise, the current frustum is cast through the arrival cell (Figure 3-c). If the extremal rays have different arrival cells, the frustum is divided into child frusta as described above, and the four child frusta are recursively cast through the arrival cell (Figure 3-d).

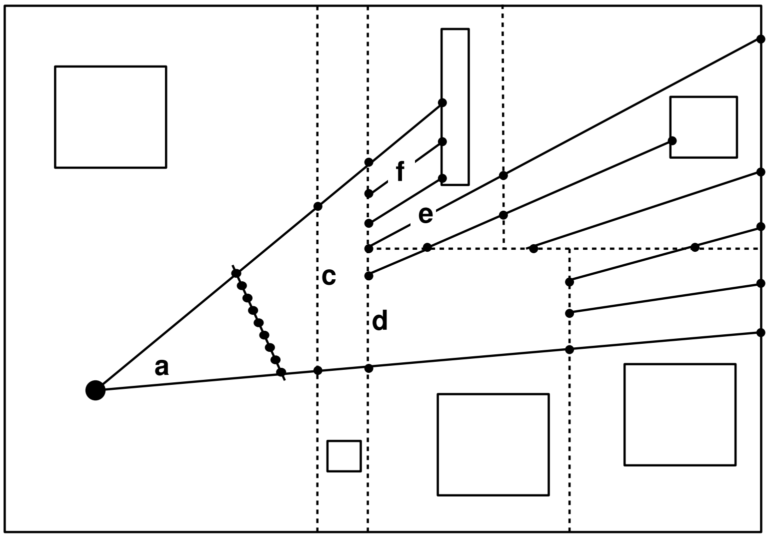


Figure 3: Frustum propagation, subdivision, and intersection testing.

If the set of elements impinging on the current frustum is non-empty, there are two cases. If the frustum has reached a resolution of two by two (in 3-D; two by one in our 2-D example), there can be no advantage to further frustum subdivision; the four extremal rays are simply propagated individually through the spatial index until they encounter a scene element or exit the spatial index (Figure 3-e). Otherwise, the frustum is divided into child frusta as described above, and the four child frusta are cast through the current cell (Figure 3-f).

It is easy to see that frustum casting terminates, as all rays of the root frustum either (1) eventually become extreme rays through frustum subdivision, then propagate through the spatial index until an element intersection is found or the ray exits the spatial index, or (2) are known to have no intersection with the scene, as their containing frustum has exited the spatial index. Upon termination, then, the first (i.e., visible) scene element, if any, has been determined at every pixel. A shading operation can then be performed at the identified visible surface point, to produce a color for display at the corresponding pixel.

4 Optimizations

There are a few immediately apparent optimizations. Frusta can be classified with regard to sign and major propagation direction, and checked against incidence lists using 1-D object sorting, as in [12]. The difference is that, while unitized rays always have a unique major direction, frusta may not.

Second, if all four extreme rays hit the same convex element in the current cell, and it is the only or minimum-depth element impinging on the frustum, all rays interior to the frustum must have their first intersection with the element.

Third, a ternary classification (ABOVE, BELOW, STRADDLING) can be deduced from each plane/bound test; once an object is found to be straddling with respect to both left and right (or top and bottom, or all four) frustum planes, it is known to be straddling for all analogous planes bounding child frusta, and need be subjected to no further tests.[6]

Finally, the four planes defining the root frustum can be augmented by "near" and "far" clipping planes as in the standard graphics pipeline, eliminating ray-object intersection checks close to the eye and deep in the spatial index.[7]

There is one special case to be considered. If the initial viewpoint is outside of the spatial index root node, the current cell is set to NULL, signifying that the spatial index hasn't yet been encountered. Frustum subdivision (unhindered by plane tests, as there can be no impinging objects) ensues until all child frusta identify appropriate arrival cells.

4.1 Discussion

We revisit the discussion of §1.6. Modern visible-surface algorithms resemble ray casting; frustum casting resembles ray casting. Both require a spatial index in order to scale. What then is the distinction between these algorithms and frustum casting?

Hierarchical depth buffers are "bottom up"; they rasterize, then cluster up to produce conservative z bounds. They essentially subdivide the viewport to full resolution, then rasterize into it. Hierarchical queries determine occlusion of subsequently rendered objects/bounds, ordered by forward traversal through a spatial index. Scene elements are polygonal, though may be tesselated, and have easily computable bounds. Primitives map to pixels through a screen-space inclusion test. Rendering a single primitive requires state proportional to the resolution of the depth buffer.

Frustum casting is "top down"; it makes queries of the whole viewport, then subdivides if the query can not be simply answered. Hierarchical queries determine whether further propagation into the spatial index is required. Scene elements are implicit, generally requiring no tesselation, but computing their bounds is in general more difficult than for polygons. Primitives map to pixels implicitly, through an object-space intersection test. Rendering a single primitive requires state proportional to the visual complexity of the scene in the vicinity of the primitive.

It remains to be seen which of these algorithms, when afforded equivalent computational bandwidth, is superior in practice, and in what regime of performance, model complexity, and viewport resolution.

[6] A related idea has been developed independently [8].

[7] Note that near and far planes are not *required* by ray casting, in contrast to approaches which use standard perspective transformation.

5 Implementation and Example Results

We implemented a simple software prototype of a progressive rendering system based on frustum casting. The major modules of the system are the spatial index module, the visible surface module, a shading module, and a reconstruction module which performs just-in-time, progressive reconstruction of successive rendered images. All of these are controlled through an extensive graphical user interface, enabling interactive motion, visualization of the algorithm data structures and queries, control of rendering quality and resource tradeoffs, and display of generated frames. We describe each module in turn, then give some examples of the system in use.

5.1 Spatial Index

The spatial index is an implementation of $k-d$ trees [7], augmented to support ray casting and primitive intersections tests as in [18], and frustum casting as described here. (Note that any spatial decomposition – fixed grid, octree, tetrahedralization, hexahedralization, etc. – would serve here so long as it supports object population, point location, and ray traversal.) The currently supported set of objects includes spheres, axial parallelepipeds and polygons (but is of course generalizable to any primitive amenable to intersection with a ray). At run time, each object is read into memory; its world-space bounding box is determined, and the object is stored in the root node of the spatial index. The spatial index is then recursively subdivided, and objects are distributed to its leaves in standard fashion.

5.2 Visible Surface

The visible surface module is a straightforward implementation of the frustum casting algorithm presented in §3, with two modifications. First, instead of a depth-first, recursive formulation, we use a statically scheduled, dynamically dispatched "job queue" as follows. Each job record contains an active/inactive flag, a frustum descriptor and the current spatial cell to be traversed. Every job record also has a pointer to its four children (subfrusta) in the static job queue (Figure 4).

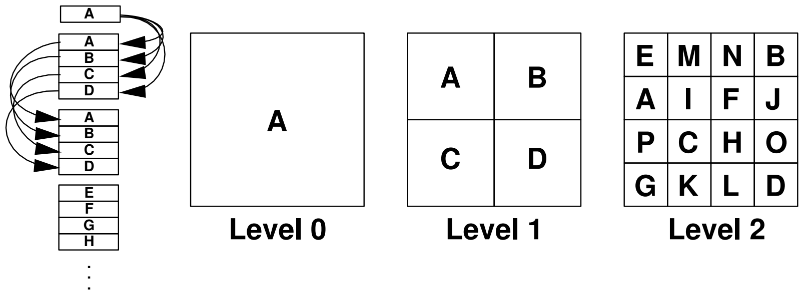


Figure 4: The job queue, and interleaved screen subdivision.

At the beginning of time an interleaved, breadth-first elaboration order is defined for the root viewport. The invariant maintained is that all subfrusta at depth q are scheduled after completion of all subfrusta at depth $p < q$, while subfrusta at the same depth are scheduled in interleaved order. The queue is initialized with a single job representing the root frustum (which includes the entire screen space viewport) advancing from the spatial cell containing the viewpoint. When a frustum f subdivides, it inserts coherent subdivision information (current cell; incident and straddling lists; etc.) into its children in constant time, and marks the children active. If f does not subdivide, its children are left untouched.

A simple dispatching algorithm executes jobs in order, while monitoring the elapsed time (in the first case) or the number of completed samples (in the second case), ceasing work on the current frame when appropriate. Any inactive job records encountered on the queue (due to prior termination of the parent frustum) are simply skipped. All completed samples are then subjected to reconstruction to produce the displayed image (§5.4).

This formulation has several advantages. First, it allows state (in the form of uncompleted jobs) to persist across multiple frames, allowing deferred intersection and shading computations. Second, the interleaved sampling schedule ensures that samples are distributed roughly equally throughout the viewport, regardless of whether or when job queue execution is interrupted (for periodic reconstruction and display). This interruptable scheduling mechanism supports either fixed-time or fixed-quality rendering.

5.3 Shading

The shading module implements Phong lighting [26], as well as diffuse texture lookups and shadow rays [5], at each pixel. From visible surface points identified by frustum casting, the shading model generates "confirmed" sample values for use by the reconstruction module and eventual display (§5.4). Shading tasks can be interactively designated as either "immediate" or "deferred." During user motion, the shading module performs only immediate subtasks. When the user pauses (i.e., when the synthetic camera is identical to that of the previous frame), any deferred subtasks are invoked. This exploits the fact that users want responsiveness, yet frequently pause to inspect the displayed scene; during such pauses, progressively higher-fidelity images are produced.

5.4 Reconstruction

The rendering system is designed to guarantee either image quality (by working until a specified number of samples are gathered), or frame rate (by sampling until a specified frame interval elapses). In either case, the number of samples gathered may be less than the number of pixels in the image to be displayed. We therefore face the well-known "reconstruction" problem of deducing values at every image pixel from a set of samples. However, in our case the problem has specific structure, in that samples arise from the interleaved order of §5.2.

Every sample in the sample buffer maintains a confirmed value (written only by the shading module) and an interpolated value (written only by the reconstruction module). The reconstruction algorithm is invoked with four confirmed sample values at the corners of the root frustum, and operates recursively as follows. If samples along the boundary midpoints have no confirmed values, they are generated by interpolation of the two relevant corner values. Similarly, the viewport's center sample is generated by interpolation of boundary values if necessary. The root viewport is recursively subdivided until all pixels are found to have been confirmed, or are interpolated.

This formulation allows the reconstruction algorithm to use the most recent values produced by the shading module; it is a kind of "just-in-time" rendering algorithm crafted in anticipation of our prototype's extension to a small number of simultaneously working processors. Our reconstruction filter is admittedly naive; however the choice of filter is an issue largely independent of our concerns here.

We note that for efficiency there is no case in which large program buffers (e.g., of samples or job descriptors) are synchronously cleared. Instead, such frame-related variables maintain a monotically increasing frame time; if it does not match the globally-maintained time, the variable is treated as uninitialized by the first relevant routine to consider it.

5.5 User Interface and Display

The user interface is based on XForms [33]. Significantly, the UI copies successive reconstructed sample buffers to the framebuffer for display. It also provides for interactive motion of a synthetic camera through a geometric scene, and control methods to specify either fixed frame rate or fixed quality, but not both ("pegging" either attribute causes the other to vary freely; this portion of the UI is shown in Figure 5).

Each of the shading calculations (ambient, emissive, diffuse, specular, shadowing) can be made immediate or deferred, as can diffuse texture lookup. Finally, many program data structures and queries, such as the spatial index, object bounds, and ray traversal algorithms, can be visualized directly in a second window (Figure 6).

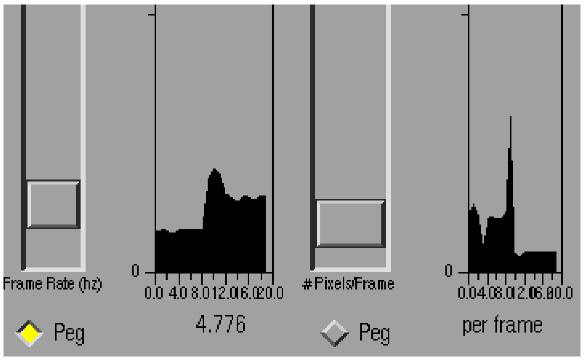


Figure 5: Pegging frame rate (left) or number of samples (right). The strip charts scroll off the right; for about half the frames number of samples was pegged, whereas for other half frame time was pegged. The spike in sample number at the midpoint of the right-hand strip chart this occurred when the user pressed the "peg frame time" button (bottom left), and paused to observe the results.

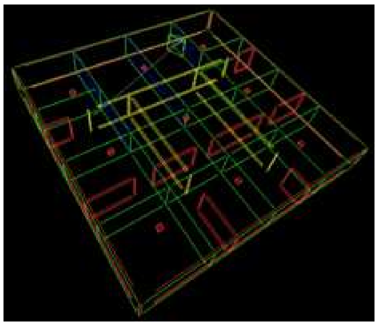


Figure 6: The spatial data structure visualization window.

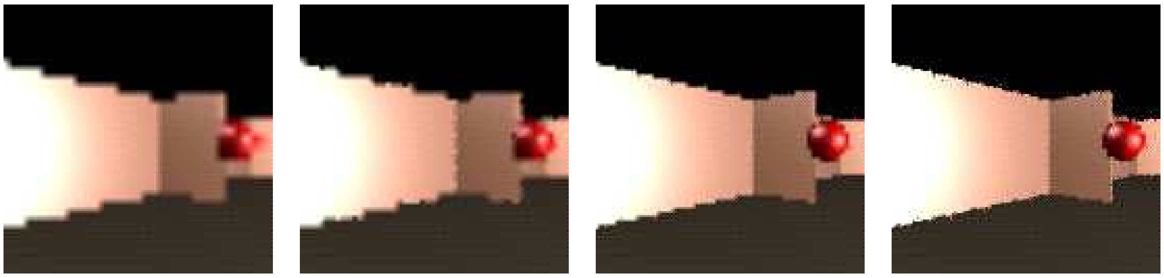
5.6 Examples and Discussion

We include here a series of example operating conditions and accompanying screen snapshots to illustrate the flexibility of the prototype system. Rather than show the system running on a fast multi-processor, we show two scenarios of modest uniprocessor operation to show its adaptability. All render window snapshots are shown at $129 \times 129 = 16,641$ pixels.

The first set of examples (Figures 7 - 10) was run on a single MIPS R4400 at 200 MHz.

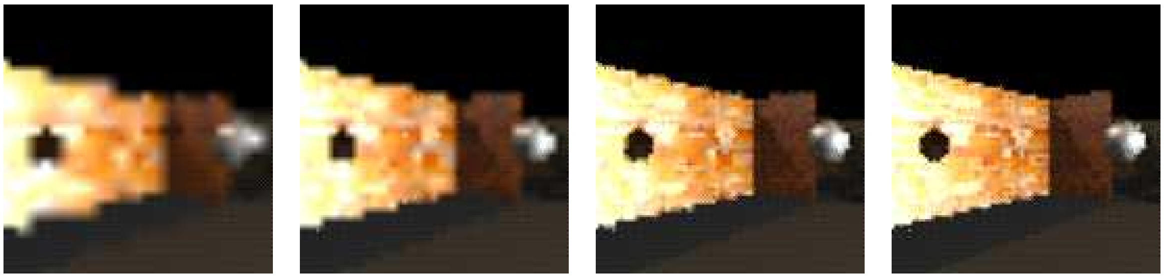
The second set of examples (Figures 11 - 12) was run on a single MIPS R5000 at 200 MHz, a considerably more powerful machine. The model contains 1,685 polygons. Frustum casting in no way assumes or depends upon architectural structure. Figure 12 shows the model viewed from "outside," i.e. as an ordinary geometric object.

Although our prototype system uses only general-purpose computational resources, it could of course benefit from dedicated hardware, already present in principle in at least one modern hardware rasterization pipeline [27], to transform rays and normals, perform plane sidedness tests, fetch object bounding volumes, object geometry, and texels from main memory, and perform sample buffer reconstruction.



Frame 1 Frame 2 Frame 3 Frame 4

Figure 7: Phong lighting per-pixel at two FPS; progressive rendering enabled.



Frame 1 Frame 2 Frame 5 Frame 8

Figure 8: Phong lighting with texturing and immediate shadow rays, 5 FPS.

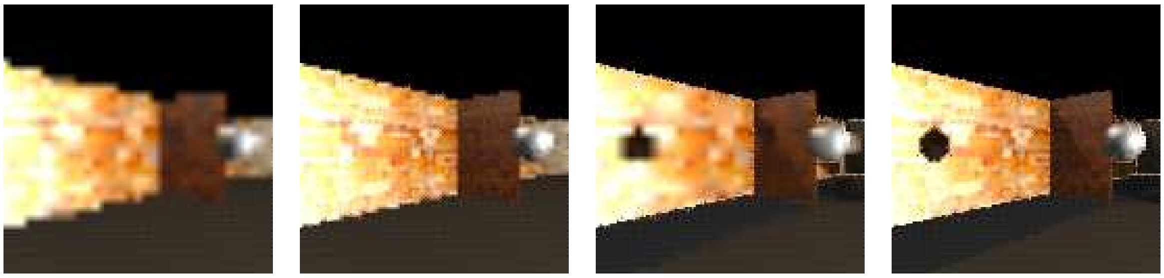
6 Conclusion

This paper makes a case, on several grounds, for the consideration of ray casting as a useful, fundamental primitive for interactive rendering. Ray casting straightforwardly enables high-fidelity imagery, with correct per-pixel depth, silhouettes, highlights, texturing, and shadowing. It is argued, from observations of several classes of visible surface algorithms, that the working set of ray casting is equivalent or superior to that of existing polygon-based rendering algorithms which rely on standard graphics pipelines.

Ray casting is computationally expensive. After analysis of three visible surface algorithms, we synthesized frustum casting, a novel, simple method which propagates rays in groups, rather than individually, yet samples in screen space, avoiding combinatorial growth and robustness problems inherent in analytic object space algorithms, while maintaining a working set equal or superior to that of existing visible-surface algorithms.

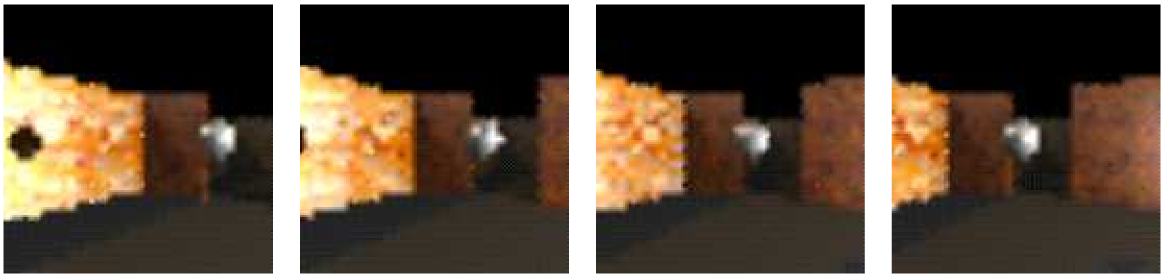
A prototype software implementation of frustum casting is demonstrated, which achieves interactivity through "just-in-time" rendering, and progressive image generation through deferral of intersection and shading operations. This sort of dynamic resource allocation – for example, tradeoffs between resolution and shading quality – is not easily attainable on today's hardware rasterization systems without idling a large fraction of dedicated hardware.

Finally, we observe that many of the atomic operations in ray casting (e.g., matrix transformations, texture fetching) are similar to those in modern polygon pipelines, and could therefore be similarly accelerated in a hardware-assisted ray casting architecture. We conclude that the adoption of visible surface techniques based on ray casting is a design alternative worthy of consideration in the development of future responsive, high-fidelity rendering systems.



Frame 1 | Frame 5 | Frame 17 | Frame 28

Figure 9: Phong lighting with texturing and deferred shadow rays, 5 FPS.

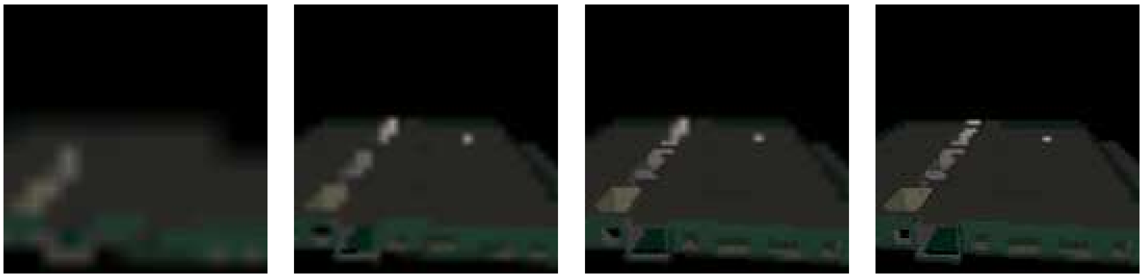


Frame 1 | Frame 2 | Frame 3 | Frame 4

Figure 10: Sampling specified at 2,500 per frame; frame rate about 2 FPS.

Interior, 10 FPS. | Interior, 5 FPS. | Interior, 2 FPS. | Interior, 1 FPS.

Figure 11: All rendering is immediate mode.



Exterior, 10 FPS. | Exterior, 4 FPS. | Exterior, 2 FPS. | Exterior, 1 FPS.

Figure 12: All rendering is immediate mode.

References

[1] Airey, J. M., Rohlf, J. H., and Brooks, Jr., F. P. Towards image realism with interactive update rates in complex virtual building environments. *ACM Siggraph Special Issue on 1990 Symposium on Interactive 3D Graphics 24*, 2 (1990), 41–50.

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

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

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

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

[6] Arge, L., Vengroff, D. E., and Vitter, J. S. External-memory algorithms for processing line segments in geographic information systems. *Lecture Notes in Computer Science 979* (1995), 295.

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

[8] Bishop, L., Eberly, D., Whitted, T., Finch, M., and Shantz, M. Designing a pc game engine. *IEEE Computer Graphics and Applications 18*, 1 (January/February 1998), 46–53.

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

[11] Coorg, S., and Teller, S. Temporally coherent conservative visibility. In *Proc.* 12^{th} *Annual ACM Symposium on Computational Geometry* (1996).

[12] Fournier, A., and Poulin, P. A ray tracing accelerator based on a hierarchy of 1D sorted lists. In *Proceedings of Graphics Interface '93* (Toronto, Ontario, May 1993), Canadian Information Processing Society, pp. 53–61.

[13] Francois Sillion, George Drettakis, B. B. Efficient impostor manipulation for real-time visualization of urban scenery. In *Eurographics '97* (Oxford, UK, 1997), Eurographics, Blackwell Publishers.

[14] Fujimoto, A., and Iwata, K. Accelerated ray tracing. *Computer Graphics: Visual Technology and Art (Proc. Computer Graphics Tokyo '85)* (1985), 41–65.

[15] Funkhouser, T., and Séquin, C. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (Proc. Siggraph '93) 27* (1993), 247–254.

[16] Funkhouser, T., Séquin, C., and Teller, S. Management of large amounts of data in interactive building walkthroughs. In *Proc. 1992 Workshop on Interactive 3D Graphics* (1992), pp. 11–20.

[17] Garlick, B., Baum, D. R., and Winget, J. M. Interactive viewing of large geometric databases using multiprocessor graphics workstations. *Siggraph '90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation)* (1990).

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

[19] Greene, N. Hierarchical polygon tiling with coverage masks. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 65–74. held in New Orleans, Louisiana, 04-09 August 1996.

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

[21] Heckbert, P., and Hanrahan, P. Beam tracing polygonal objects. *Computer Graphics (Proc. Siggraph '84) 18*, 3 (1984), 119–127.

[22] Hudson, T., Manocha, D., Cohen, J., Lin, M., Hoff, K., and Zhang, H. Accelerated occlusion culling using shadow frusta. In *Proc.* 13^{th} *Annual ACM Symposium on Computational Geometry* (1997), pp. 1–10.

[23] Jones, C. A new approach to the 'hidden line' problem. *The Computer Journal 14*, 3 (1971), 232–237.

[24] Luebke, D., and Georges, C. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *1995 Symposium on Interactive 3D Graphics* (Apr. 1995), P. Hanrahan and J. Winget, Eds., ACM SIGGRAPH, pp. 105–106. ISBN 0-89791-736-7.

[25] 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.

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

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

[28] Sutherland, I. E., Sproull, R. F., and Schumacker, R. A. A characterization of ten hidden-surface algorithms. *Computing Surveys 6*, 1 (1974), 1–55.

[29] Teller, S., and Séquin, C. H. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proc. Siggraph '91) 25*, 4 (1991), 61–69.

[30] Warnock, J. A hidden-surface algorithm for computer generated half-tone pictures. Tech. Rep. TR 4–15, NTIS AD-733 671, University of Utah, Computer Science Department, 1969.

[31] Weiler, K., and Atherton, P. Hidden surface removal using polygon area sorting. *Computer Graphics (Proc. Siggraph '77) 11*, 2 (1977), 214–222.

[32] Zhang, H., Manocha, D., Hudson, T., and Hoff III, K. E. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), T. Whitted, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 77–88. ISBN 0-89791-896-7.

[33] Zhao, T. C., and Overmars, M. X-Forms `http://bloch.phys.uwm.edu/xforms`.