University of California
Santa Cruz

Hierarchical Rendering of Complex Environments

A dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy
in
Computer and Information Sciences
by
Ned Greene
June 1995

Contents

List of Figures

Hierarchical Rendering of Complex Environments

Ned Greene

ABSTRACT

We present three related algorithms designed to accelerate rendering of very complex, densely occluded scenes composed of geometric models where surface shading is restricted to local illumination methods. The algorithms share the same basic structure, maintaining hierarchical data structures in both object space and image space to enable finding visible geometry by logarithmic search. To render a scene, the basic algorithm traverses nodes in the object-space hierarchy (an octree) in front-to-back order, testing the bounding volumes of nodes for visibility and culling hidden nodes. Thus, only visible nodes and their children in the hierarchy are visited,[1] and only primitives in visible nodes need to be rendered. This procedure culls most hidden geometry in densely occluded scenes, but some geometry will still overlap on the screen when primitives are projected. To cull remaining hidden geometry, we perform hierarchical culling in image space, using the image-space hierarchy to maintain visibility information about previously rendered geometry.

Two of the algorithms, *hierarchical z-buffer visibility* and *hierarchical polygon tiling*, have exceptional performance and are appropriate for interactive applications. The hierarchical z-buffer algorithm maintains depth samples in a pyramid, permitting z-buffer depth comparisons to be performed hierarchically, which enables very rapid culling of hidden octree cubes and hidden primitives. Visible primitives are rendered with the speed of traditional incremental scan conversion. The algorithm's hierarchical culling capabilities make it possible to compute standard z-buffer images of densely occluded scenes much faster than traditional z-buffering. Alternatively, instead of performing visibility operations by z-buffering, the second variation of the basic algorithm employs a novel polygon tiling algorithm to cull hidden octree cubes and tile potentially visible polygons. Called *hierarchical polygon tiling*, this method traverses scene polygons front to back and performs tiling by Warnock-style recursive subdivision of image space. Visibility information is maintained in an image-space pyramid of coverage masks, which permits subdivision to be driven very efficiently by boolean mask operations. When antialiasing is performed by oversampling and filtering, this tiling algorithm is much faster than hierarchical z-buffering.

The third algorithm, *error-bounded antialiased rendering*, is much slower than the other two algorithms, but is unique in its ability to produce antialiased images of guaranteed

[1] With some variations of the algorithm, some *nearly* visible nodes and their children are also visited.

accuracy. This algorithm also culls hidden octree cubes and tiles potentially visible polygons by recursive subdivision of the image-space hierarchy. By using interval methods to control subdivision of image space, the algorithm is able to identify regions where geometry or shading is complex and do as much work as necessary within those regions to produce accurate results. Consequently, each pixel of the output image can be guaranteed to be within a user-specified error tolerance of the filtered underlying continuous image. To the best of my knowledge, the images produced with this algorithm are the only computer-generated images of guaranteed accuracy that have ever been created of extremely complex scenes or scenes rendered with complex shaders.

Acknowledgements

Special thanks are due Mike Kass, Gavin Miller, and Frank Crow, colleagues at Apple Computer where this research was done. Mike collaborated on two Siggraph papers that constitute much of this thesis, making substantial contributions to the ideas and their presentation. I like to think that some of his gift for critical thinking rubbed off on me in the course of our work together. Gavin's ideas about object-space culling were my original inspiration in pursuing this research. They underlie chapter 3 and its precursor, a Siggraph paper on which he collaborated. In addition, Gavin has always been willing to listen to my half-baked ideas and lend his extraordinary technical expertise to refine them. As manager of the Graphics Research Group, Frank encouraged this research and saw that it received corporate support. Beyond that, he was largely responsible for the wonderful working environment enjoyed by the graphics research staff at Apple.

Of course I would also like to thank my advisor, Jane Wilhelms, for providing many helpful suggestions in the course of preparing this thesis and for accommodating the needs of a fair-weather graduate student. I am also indebted to Nelson Max for serving on my committee, offering insightful criticism, and providing encouragement and inspiration over the years.

Paul Heckbert suggested improvements in the temporal-coherence procedure of §3.3.3, and as editor of *Graphics Gems IV*, helped to clarify presentation of the intersection algorithm in the appendix. I am also grateful to Paul for his endless generosity in helping me to develop technical skills when we worked together at the NYIT Computer Graphics Lab. Discussions with Bruce Naylor improved my understanding of his work on visibility. Apple colleagues Eric Chen and Steve Rubin made radiosity and modeling software available for constructing test models.

Finally, I am particularly grateful to Lance Williams who introduced me to computer graphics and "the hidden-line problem" when we were undergraduates at the University of Kansas, and who has been a good friend and mentor throughout my career.

Chapter 1

Introduction

1.1 Overview

Since its inception some three decades ago, the field of computer image synthesis has witnessed enormous progress. The raw speed and storage capacity of computers has increased by orders of magnitude and comparable progress has been made in image synthesis algorithms. With this progress, the domain of practical applications has grown dramatically, and these applications increasingly involve the rendering of very complex three-dimensional environments, which is the subject of this thesis. More specifically, the topic that this thesis addresses is the rendering of complex, densely occluded scenes composed of geometric primitives where surface shading is performed with local illumination methods, as distinguished from global illumination methods. This topic is germane to applications that cover the spectrum of uses for computer graphics and computer animation.

One important applications domain is interactive systems for flight simulation, virtual reality, computer-aided design, scientific visualization, and other applications that require rapid frame updates for effective user interaction. Within this domain there is persistent demand for dealing with more and more complexity. Since rendering is usually the computational bottleneck in such systems, faster rendering algorithms are fundamental to building more powerful systems. For rendering applications that demand speed, we propose two related algorithms that effectively exploit various forms of coherence through the use of hierarchical methods to greatly accelerate rendering of densely occluded scenes. The first algorithm, *hierarchical z-buffer visibility*, is a visibility algorithm that accelerates generation of standard point-sampled z-buffer images. The second algorithm is a variation on the first, employing a more efficient tiling algorithm that we refer to as *hierarchical polygon tiling*. This algorithm is well suited to producing images with high-quality antialiasing performed by oversampling and filtering. Our results show that both of these algorithms provide a practical means of performing visible-surface determination in very complex, densely occluded scenes at rapid frame rates, at least for predominantly static environments.

The second major topic that we explore is antialiased rendering with guaranteed accuracy. In very complex scenes, numerous primitives may be visible within a single pixel, and in extreme cases, traditional methods of antialiasing by oversampling and filtering do not always yield good results. Ideally, a rendering algorithm should be able to recognize regions

of the screen where visible geometry or its shading is very complex and then work as hard as necessary to perform proper filtering, thereby producing images that are free of visual artifacts. This is the approach taken by the third hierarchical rendering algorithm, which we call *error-bounded antialiased rendering*. For a broad class of shading functions, this algorithm is able to guarantee that each pixel of the output image is within a user-specified error tolerance of the properly filtered value. While the algorithm is too slow for interactive applications on contemporary computers, it is appropriate for applications that demand very high image quality such as animation for entertainment purposes.

1.2 Goals of this Work

This research is directed toward developing efficient, accurate, and general methods for rendering very complex environments. The novel features of the methods we present relate primarily to visibility, tiling, and antialiasing. With our approach, the key to performing visibility operations efficiently is applying hierarchical methods wherever possible to exploit the various forms of coherence that are inherent in visibility computations – object-space coherence, image-space coherence, and for animation, temporal coherence. Exploiting coherence enables us to efficiently cull hidden geometry and efficiently render visible geometry. For applications that demand very high image quality, the key to accurate antialiasing is the use of interval methods that establish guaranteed bounds on error. Interval methods enable us to guarantee that each pixel in the output image is accurate to within a user-specified error tolerance. Regarding generality, although the specific procedures we present are designed for rendering polygonal models, with some effort they can be adapted to render other types of geometric primitives. Unlike some alternative methods, our algorithms can be applied to arbitrary polygonal models, and do not depend on any particular geometric characteristics of the scene, such as the sort of partitioning that is typical of architectural models. The relative advantage of our methods is most pronounced for *densely occluded* scenes – scenes in which only a small proportion of the geometric primitives that compose the scene are visible from a typical viewpoint, the remainder being occluded by primitives that are closer to the observer. Nonetheless, in many cases our methods are also suitable for rendering relatively simple scenes.

The overall efficiency goal of research into visibility and rendering algorithms is to design algorithms that do work proportional to the *visible complexity* of the scene in the output image, rather than the overall complexity of scene geometry. As we will see, hierarchical methods for culling hidden geometry and processing visible geometry permit us to approach this goal.

1.3 Organization of this Thesis

Five chapters and an appendix follow this introduction.

Chapter 2 discusses previous work relating to the rendering of complex scenes with local shading. The first part of this chapter is devoted to the visible-surface problem, the forms of coherence that are inherent in visibility computations, and traditional ways of exploiting this coherence. The second part of Chapter 2 discusses the aliasing problem and traditional methods for combating it.

Chapter 3 presents our method for accelerating z-buffer tiling of very complex scenes, *hierarchical z-buffer visibility*. This chapter is an expanded version of a paper I presented at Siggraph in 1993 [Greene-Kass-Miller93], which was written with two of my colleagues in the Advanced Technology Group at Apple Computer, Michael Kass and Gavin Miller.

Chapter 4 presents a very efficient hierarchical algorithm for tiling polygons, which is an attractive alternative to the z-buffer tiling algorithm of chapter 3, particularly when antialiasing is performed by oversampling and filtering. This chapter is a version of a paper scheduled for presentation at Siggraph '95 [Greene95] at a tutorial called *Interactive Walk-Through of Large Geometric Datasets*.

Chapter 5 presents our method for antialiased rendering with guaranteed accuracy, *error-bounded antialiased rendering*. This chapter is a slightly modified version of a paper I presented at Siggraph in 1994, which was written with Michael Kass [Greene-Kass94].

In Chapter 6, we trace the historical development of the ideas in this thesis and present our conclusions.

The appendix presents an efficient algorithm for detecting intersection of a rectangular solid and a convex polygon or polyhedron, operations that our rendering algorithms frequently perform. This material appeared in a slightly different form in *Graphics Gems IV*, published in 1994 [Greene94].

Chapter 2

Previous Work

In this chapter we review traditional techniques that can be applied to the problem that this thesis addresses: rendering of scenes composed of very complex geometric models where surface shading is restricted to local illumination methods. The overall problem can be thought of as involving two basic operations: a *visibility operation* in which we determine the relevant visible geometry for each pixel of the output image, and a *shading/filtering operation* in which we determine pixel color given the visible geometry along with its surface properties and local illumination information.[1] Performing these operations can be very challenging for complex scenes because dozens or even hundreds of primitives may be visible at individual pixels, and algorithms must be carefully chosen to keep computational expense within acceptable bounds.

Rendering methods can be arranged on a *performance/quality spectrum* that reflects the tradeoff between speed and quality, ranging from very fast methods that produce relatively low-quality images to very accurate methods that run relatively slowly. For applications in which speed is the predominant consideration, it is usually not practical to identify and filter all visible geometry in the neighborhood of each pixel. The fastest alternative is to identify a single visible primitive at each pixel and evaluate shading at a single visible point on that primitive. Traditional z-buffering is an example of this *point-sampling* approach where each pixel is shaded according to the surface point that is visible at pixel-center. It is well known that this point-sampling approach causes aliasing artifacts, so for applications at the other end of the performance/quality spectrum that require high-quality images, we identify and filter all geometry that is visible in the neighborhood of each pixel. In this case, we must employ a visibility algorithm that is able to identify all visible primitives within regions of the screen. This permits proper antialiasing when filtering is performed.

This chapter is organized in two sections. In the first section we discuss visible-surface determination, both point-sampling methods and methods that identify all visible geometry. In the second section we discuss antialiasing, reviewing the causes of aliasing in computer-generated images and examining various methods that have been proposed to combat it.

---

[1] In some cases, of course, these operations are simpler than this general framework suggests. With traditional pointed-sampled z-buffering, for example, the only "relevant visible geometry" at a pixel is the primitive that is visible at pixel-center, and shading is performed without filtering.

2.1 Visible-Surface Determination

One of the most fundamental components of algorithms for rendering three-dimensional geometric models is visible-surface determination – determining which surfaces or portions of surfaces are visible given the geometric description of a scene and a viewing projection. Visibility was one of the first topics in computer graphics to gain academic respectability and it has been the subject of extensive research that has explored a great variety of different approaches. Most of these algorithms, however, must consider all primitives in a scene one by one, regardless of whether they are visible, so they are not practical approaches to rendering very complex, densely occluded scenes. To limit our review to the more relevant techniques, our discussion of visibility will focus on a few basic algorithms and methods that have been specifically devised to accelerate rendering of complex scenes.

Efficiency is of crucial importance in performing visibility computations on very complex, densely occluded scenes. In particular, it is important to cull hidden geometry efficiently, since the vast majority of primitives are typically hidden. As we will see, exploiting *coherence* is the key to performing visibility computations efficiently, and hierarchical methods based on hierarchical data structures offer a particularly powerful approach.

We begin our discussion of visibility with a review of coherence and how it can be exploited using hierarchical methods. Next, we discuss four basic visibility algorithms – the Warnock algorithm, the Weiler-Atherton algorithm, ray casting, and z-buffering – and examine how they exploit coherence. Finally, we review visibility algorithms specifically designed to accelerate visibility computations in complex scenes.

2.1.1 Coherence

As recognized by Sutherland, Sproull, and Schumacker more than twenty years ago in their classic survey of visible-surface algorithms [Sutherland-et-al74], exploiting *coherence* is the key to designing efficient visibility algorithms. By coherence we mean the degree to which a scene model or its projection on the screen is locally similar or *coherent*. For example, adjacent primitives of a scene model are likely to have the same visibility status in a particular view, either hidden or not. In the image plane, if a primitive is visible at a particular pixel, it is likely to be visible at an adjacent pixel. These sorts of relationships can often be exploited by visibility algorithms, since it is frequently possible to reuse information calculated for one entity (region, object, pixel, etc.) by applying it to the processing of a neighboring entity.

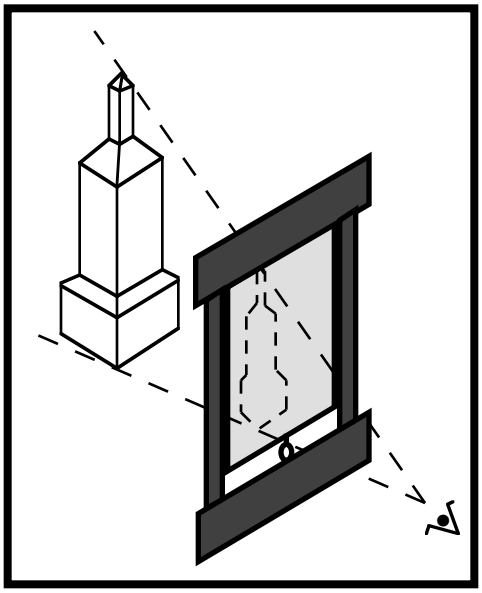
The various forms of coherence that a visibility algorithm can exploit fall into three general categories: object-space, image-space, and temporal coherence. With object-space

coherence, in many cases a single computation can resolve the visibility of an object or a collection of objects that are near each other in space. This principle is schematically illustrated in figure 2.1a, where the whole region of space occupied by a building is occluded by a single object in the foreground. Thus, the entire building model could be culled by establishing that its bounding box in object space is hidden. With image-space coherence, in many cases a single computation can resolve the visibility of an object or a collection of objects that cover a region of the screen. This is shown schematically in figure 2.1b, where the region of the screen covered by the projection of a building is also covered by a single object that is closer to the observer. As before, this indicates that a single visibility test could cull the entire building model. With temporal coherence, visibility information from one frame of animation can sometimes be used to accelerate visibility computations for the next frame. As shown schematically in figure 2.1c, if we know that a building is hidden in one frame, frame-to-frame coherence tells us that it is likely to be hidden in the next frame, a circumstance that some visibility algorithms are able to exploit. Some authors use the term *frame coherence* rather than temporal coherence.

It can be argued that the distinction we have drawn between object-space and image-space coherence is somewhat artificial, since visibility really depends on occlusion relationships in perspective space (assuming a perspective projection). Nonetheless, the distinction often makes sense in the context of specific visibility algorithms that organize geometry into regions of object space (e.g. an object-space octree) or organize projected geometry into regions of image space (e.g. an image-space quadtree). For example, according to our convention, a visibility procedure that culls a collection of objects by establishing that its object-space bounding box is hidden would be said to be exploiting object-space coherence.

2.1.2 Exploiting Coherence with Hierarchies

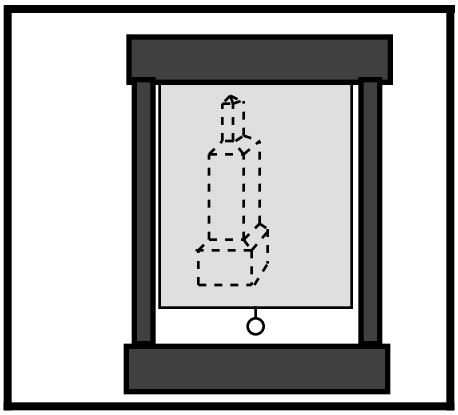
For our purposes in processing complex scenes, one of the most important tasks that can be accelerated by exploiting coherence is culling of hidden geometry. Ideally, culling can be performed hierarchically by attempting to resolve visibility in coarse regions before proceeding to finer regions. For example, we could apply a hierarchical approach to visibility computations for a building model by establishing that the building is at least partially visible before considering whether a particular room is visible, establishing that a room is at least partially visible before considering whether any of its furnishings are visible, and so forth. Hierarchical methods for culling hidden geometry and processing visible geometry can be performed in object space, image space, and in the case of animation, in space-time. Broadly speaking, hierarchical methods enable logarithmic search for features of interest,



a

Object-Space Coherence

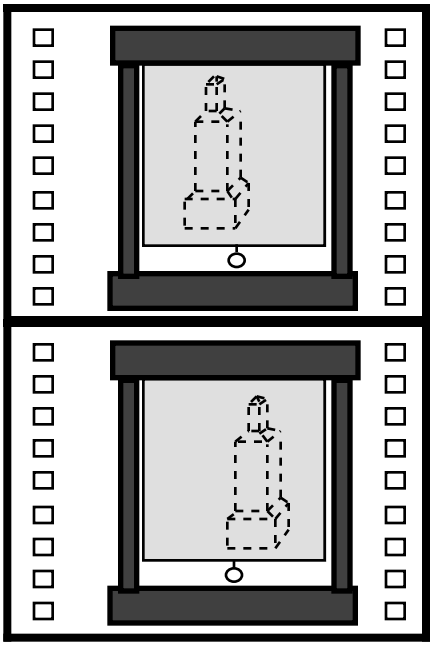
Often a single visibility computation can resolve the visibility of a collection of objects that are near each other in space.



b

Image-Space Coherence

Often a single visibility computation can resolve the visibility of a collection of objects that are near each other on the screen.



c

Temporal Coherence

Knowing the visibility of a collection of objects in one frame can often accelerate visibility computations for those objects in the next frame.

Figure 2.1 Three forms of coherence that a visibility algorithm can exploit.

thereby taking full advantage of available coherence.

Hierarchical data structures are often used to facilitate performing visibility operations hierarchically. The most common approach is to organize the geometric primitives of a scene model or their projections on the screen into nested bounding boxes or some other form of spatial partitioning. Generally speaking, to exploit object-space coherence, scene primitives are organized into an object-space hierarchy such as an octree or a three-dimensional BSP tree. Likewise, to exploit image-space coherence, the screen projections of primitives can be organized into an image-space hierarchy such as an image pyramid, quadtree, or two-dimensional BSP tree. Although organizing geometric information into hierarchies entails some overhead in both computation and storage, it can greatly accelerate some operations that visibility algorithms perform such as culling off-screen geometry, culling occluded geometry, intersection testing, and depth-ordered traversal of geometry.

Clark was one of the first to recognize the importance of hierarchical organization of geometric models in accelerating rendering of complex scenes. In a short theoretical paper from 1976, he set forth the goal of designing a visibility algorithm "in which the computation time grows linearly with the visible complexity of the scene" [Clark76]. Briefly summarized, Clark suggested organizing a scene model into a tree-structured hierarchy, associating bounding volumes with nodes in the tree, and representing individual objects at different levels of detail. He noted that including bounding volumes in the hierarchy permitted hierarchical culling of geometry lying outside the viewing frustum, or in other words, logarithmic search for on-screen geometry. Moreover, hierarchical culling to the viewing frustum accelerates clipping of on-screen geometry to frustum planes, because geometry inside bounding volumes that lie entirely inside the viewing frustum never needs to be considered for clipping. Representing models at different levels of detail permits rendering simpler versions when the screen area covered by an object is small, a technique which has been widely used to enhance the performance of flight simulators. Clark also sketched a "recursive descent visible-surface algorithm" that is sometimes able to hierarchically cull hidden geometry. However, as presented, this algorithm would probably cull only a small fraction of the hidden geometry in a typical scene. Although Clark's methods were apparently not implemented at the time, they foreshadowed subsequent developments in the field. In particular, his method of culling hierarchical models to the viewing frustum has been applied to octree models, as described in §2.1.6.

We turn now to reviewing specific visibility algorithms and analyzing how they exploit coherence using hierarchies and other methods, paying particular attention to hierarchical methods for culling hidden geometry.

2.1.3 The Warnock Algorithm

Warnock devised the first visible-surface algorithm based on recursive spatial subdivision [Warnock69]. The underlying idea is elegantly simple. We examine a region of the screen. If the projected geometry within that region is very simple, visibility relationships among primitives can be established easily, and visible geometry can be displayed conveniently. If the geometry within the region is not so simple, we subdivide the region into quadrants, which are then examined using this same procedure. Subdivision ultimately simplifies geometry, since the regions considered become progressively smaller.

The Warnock algorithm is actually a family of algorithms based on a common subdivision procedure, and the control structure varies from implementation to implementation [Rogers85]. Here we describe a typical variation. We start by projecting all primitives onto the screen and storing them in a list. Beginning with a square representing the whole screen and the complete list of projected primitives, we assign each primitive to one of three categories: (1) lying entirely outside the square (an *outside* primitive), (2) completely surrounding the square (a *surrounding* primitive), or (3) intersecting the square but not surrounding it (an *intersecting* primitive). Outside primitives are marked as hidden. Surrounding and intersecting primitives that are hidden by a surrounding primitive are marked as hidden. If the list contains no intersecting primitives and there is a single surrounding primitive, that surrounding primitive covers the square, it is added to the display list, and we are done with the square. If we have reached the subdivision limit (a specified limit on the depth of recursion), the primitive that is nearest the observer within the square is assumed to be the only visible primitive within the square, it is added to the display list, and we are done with the square. Otherwise, visibility within the square has yet to be resolved and this same procedure is applied recursively to each of the square's quadrants and the list of potentially visible primitives.

As is apparent in figure 2.2, the Warnock algorithm produces a quadtree subdivision of the screen for visible geometry, identifying regions of the screen where a single primitive is visible. The algorithm can be viewed as a logarithmic-search procedure for visible edges,[2] hierarchically tiling visible regions of primitives and hierarchically culling hidden regions, performing visibility tests at fine levels of the quadtree only where comparisons at coarser levels fail to resolve visibility. Thus, the Warnock algorithm serves as a good example of how a visibility algorithm can apply hierarchical methods to exploit image-space coherence. While the algorithm in its original form has never been widely used, its basic

[2] "The process of subdividing the picture can be thought of as a logarithmic search for points where there is a change in the characteristics of the picture." [Warnock69], page 3.

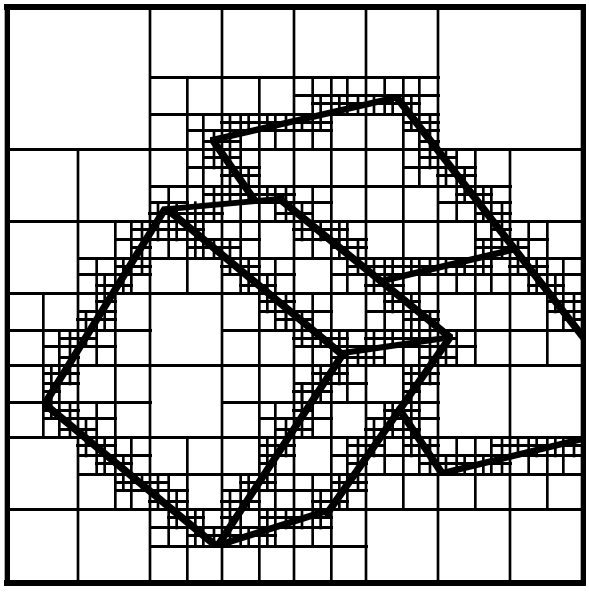


Figure 2.2 Quadtree subdivision produced by the Warnock algorithm for a simple scene. Subdivision of the quadtree continues until (1) a primitive surrounds the cell and hides all other primitives overlapping the cell, (2) no primitives intersect the cell, or (3) the maximum level of subdivision is reached. The algorithm can be viewed as a logarithmic-search procedure for visible edges.

---

divide and conquer strategy has been widely adopted. Variations of the Warnock algorithm are a component of other visibility algorithms that we will be discussing, including the Weiler-Atherton algorithm [Weiler-Atherton77], Meagher's octree rendering algorithm [Meagher82b], the ZZ-buffer algorithm [Salesin-Stolfi89], Naylor's BSP-tree rendering algorithm [Naylor92b], and the rendering algorithms we present in chapters 4 and 5.

The Weiler-Atherton Algorithm

One shortcoming of the Warnock algorithm is that it finely subdivides the screen along edges of primitives. To avoid this behavior, Weiler and Atherton subsequently devised an analogous recursive subdivision procedure for rendering polygonal scenes which subdivides the screen along polygon boundaries [Weiler-Atherton77]. The algorithm works as follows. The first step, which is not essential but is usually done to improve efficiency, is to sort polygons in approximate front-to-back order. The nearest polygon on the list is designated the *clip polygon*. All remaining polygons are clipped to this polygon and put in two lists, *inside polygons* and *outside polygons*. Then the inside list is processed. Inside polygons that lie entirely behind the clip polygon are culled. Note that this process does not necessarily

cull all inside polygons, because depth ordering of polygons is only approximate. Each remaining inside polygon is processed recursively to clip the pieces on the inside list against it. When this recursive procedure finishes, the remaining polygons on the inside list are known to be visible and are displayed. If the outside list is null, we are done. Otherwise, the whole procedure begins again, the outside list becoming the list of polygons, its nearest entry becoming the clip polygon. When the algorithm finishes we have an object-precision visibility map of the scene.

The Weiler-Atherton algorithm as originally formulated is not an efficient way to process complex, densely occluded scenes because every polygon must be considered at least once, and for typical scenes, many polygons must be considered numerous times. In addition, the basic clipping procedure is slow and difficult to implement because it requires clipping concave polygons with holes against each other. In §2.1.6 we discuss Naylor's variation of the Weiler-Atherton algorithm, which is better suited to processing complex scenes.

2.1.4 Ray Casting

We now consider another fundamental visibility algorithm, ray casting, which was first described by Appel [Appel68] and later implemented by Goldstein and Nagel as part of the MAGI graphics system [Goldstein-Nagel71]. Ray tracing is a generalization of ray casting, popularized by Whitted several years later [Whitted80].

With ray casting in its simplest form, an image is produced by casting a ray from the viewpoint through the center of each pixel of the output image. Along each ray, all intersections with scene primitives are determined. At each pixel, the nearest of these intersection points to the observer corresponds to the visible-surface point at pixel-center, which is evaluated to shade the pixel. More accurate results can be obtained by casting rays through multiple points within each pixel and filtering the resulting shading values.

The naive algorithm just described is grossly inefficient because it requires testing each ray for intersection with every primitive in the scene. In practice, most implementations of ray casting circumvent this problem by organizing scene primitives in an octree or other object-space hierarchy [Reddy-Rubin78, Rubin-Whitted80, Glassner84, Kay-Kajiya86, Kaplan87, Jevans-Wyvill89]. In the case of an octree, rays are propagated through the octree and are intersected only with primitives in octree cubes that the ray intersects. Once the nearest opaque object along a ray has been determined, there is no need to look farther down along the ray. Thus, large hidden regions of object space in densely occluded scenes may never be visited by rays, and in general, ray casting effectively ignores most hidden geometry. Since the structure of an octree is derived from binary subdivision, ray casting

through an octree can be thought of as a logarithmic search for primitives of interest. In terms of exploiting coherence during ray traversal, ray casting through an octree or other object-space hierarchy effectively exploits object-space coherence by hierarchically culling irrelevant geometry.

From the standpoint of efficiency, the basic problem with ray casting through a subdivision of object space is that the algorithm starts from scratch with each ray, and is not able to profit from visibility computations for neighboring points on the screen, even if the same object is visible there. Although there are heuristic methods that construct estimates of the results of ray casting a pixel from the results at nearby pixels (e.g. [Badt88]), they do not guarantee accurate results. In short, most ray-casting and ray-tracing algorithms are not able to exploit image-space coherence.

One exception is the *ZZ-buffer* algorithm of Salesin and Stolfi, in which scene primitives are organized into a subdivision of image space rather than object space [Salesin-Stolfi89, Salesin-Stolfi90]. A ZZ-buffer is a subdivision of the screen into rectangular cells, each covering a fixed-size block of pixels, and each containing a list of primitives that are visible within that cell. The algorithm operates in two passes, a *tiling pass* that uses a variant of the Warnock algorithm to determine which primitives are visible within each cell in the ZZ-buffer, and a *rendering pass* that produces the output image by performing ray casting on primitives stored in the ZZ-buffer. Since potentially visible primitives are precomputed for each region of the screen, casting a ray using a ZZ-buffer requires considerably less work than casting a ray through a subdivision of object space, so this method is well suited to stochastic ray casting where numerous rays are cast at each pixel. However, this computational advantage is offset by the cost of building the ZZ-buffer, which must be done every frame, even if scene geometry is static. This can require substantial computation in densely occluded scenes, since visibility computations must be performed on every primitive, visible or not. From the standpoint of coherence, while ray casting with a ZZ-buffer successfully exploits image-space coherence, the algorithm's tiling pass misses an opportunity to exploit object-space coherence in densely occluded scenes when it considers primitives one-by-one in large hidden regions of object space. This shortcoming could be overcome by organizing scene geometry in an object-space hierarchy and hierarchically culling hidden regions of object space during the tiling pass. In fact, this is the approach we take in the rendering algorithm presented in chapter 5.

Ray-casting algorithms that exploit temporal coherence have also been explored, and while this form of coherence is not commonly exploited in practice, various techniques exist for special cases. If all objects in the scene are convex and remain stationary while the camera moves, then there are constraints on the way visibility can change which a ray

caster can exploit [Hubschman-Zucker82]. If instead the camera is stationary, then rays that are unaffected by the motion of objects can be detected and used from the preceding frame [Jevans92]. When interactivity is not an issue and sufficient memory is available, it can be feasible to render an entire animation sequence at once using space-time bounding boxes [Glassner88, Chapman-et-al91].

Summing up our review of ray casting, although existing algorithms exploit object-space, image-space, and temporal coherence, none exploits all three simultaneously. The fundamental dilemma seems to be that it is difficult to effectively exploit both object-space and image-space coherence. Ray casting through an octree or other subdivision of object space effectively ignores most hidden geometry, thereby exploiting object-space coherence, but it fails to exploit image-space coherence, starting from scratch with every ray. Conversely, the ZZ-buffer algorithm effectively exploits image-space coherence to accelerate ray-object intersections, but it may need to consider primitives in large hidden regions of object space, failing to exploit this aspect of object-space coherence.

2.1.5 Z-Buffer Scan Conversion

The z-buffer visible-surface algorithm, introduced by Catmull [Catmull74], is one of the simplest to implement in both software and hardware, and it remains one of the most widely used rendering algorithms. Briefly summarized, we associate a depth value with each screen pixel and these depth values are collectively referred to as the *z-buffer* or *depth buffer*. Following initialization of the image buffer to the background color and initialization of the z-buffer to the depth of the far clipping plane, each geometric primitive in the scene model is *scan converted*, by which we mean that the depth and shaded color of the primitive is determined at each pixel sample that lies within its screen projection. As each of these pixel samples is considered, if the sample's depth is closer to the observer than the depth value currently stored in the z-buffer, this new depth value replaces the value stored in the z-buffer and the new color value replaces the value stored in the image buffer. When all primitives have been scan converted, we are done rendering a frame.

Among the advantages of z-buffering, it is very fast for scenes having low depth complexity, memory requirements are fixed and relatively modest, different types of geometric primitives can be scan converted into the same z-buffer, and primitives can be processed in any order, which eliminates the need for sorting. On the negative side, this is a point-sampling approach, so it is prone to aliasing.

Z-buffering makes good use of image-space coherence, provided that traditional incremental methods are applied to scan converting primitives. In scan converting Gouraud-

shaded polygons, for example, implementations usually do a set-up computation for each polygon and for each scanline the polygon intersects. Then, depth and shading values at consecutive pixels across a scanline can be obtained by evaluating a simple difference equation [Foley-et-al90]. Since such incremental updates require much less computation than computing the same information from scratch, the savings from image-space coherence can be substantial.

While traditional z-buffering effectively exploits image-space coherence, it makes no use of object-space or temporal coherence. Each primitive is rendered independently, and no information is saved from previous frames. For densely occluded environments like a model of a building, this is very inefficient. In this case, traditional z-buffering will need to render every primitive of every object in every room in the building, even if the whole building cannot be seen, because the algorithm can only resolve visibility at the pixel level during scan conversion of individual primitives.

2.1.6 Visibility Algorithms for Complex Scenes

Thus far, our discussion of visibility has focused on four basic visibility algorithms – the Warnock algorithm, the Weiler-Atherton algorithm, ray casting, and z-buffering – and their strategies for exploiting coherence. None of these algorithms efficiently processes the sort of very complex, densely occluded scenes that this thesis addresses. With the Warnock algorithm, the Weiler-Atherton algorithm, and z-buffering, the problem is that all primitives must be considered, whether or not they are visible. For a complex scene composed of a very large number of primitives this is a serious limitation, because just traversing all scene geometry can take a prohibitive amount of time. With ray casting, the problem is that the basic operation of casting a ray requires considerable computation and each ray must be computed independently.

In this section, we review algorithms that have been specifically designed to accelerate visibility computations in very complex scenes. Generally speaking, these methods achieve efficiency by combining features of the basic visibility algorithms that we have already discussed, such as partitioning object space and image space into spatial subdivisions, applying hierarchical methods, and so forth. We will review Clark's hierarchical culling method [Clark76], Meagher's octree rendering algorithm [Meagher82b], the potentially visible set methods of Airey [Airey90] and Teller and Séquin [Teller-Sequin92], and Naylor's visibility algorithm for BSP trees [Naylor92b].

Culling Scene Geometry to the Viewing Frustum

As mentioned in §2.1.2, an early paper by Clark outlined a number of advantages of organizing scene geometry in a spatial hierarchy, among them the ease and efficiency of hierarchically culling off-screen geometry [Clark76]. Clark's culling procedure was later implemented for octree representations of geometric models by Garlick, Baum, and Winget [Garlick-et-al90]. According to their method, we begin by organizing scene geometry into an octree. As we will discuss in §3.3.1, there are various ways to build an octree, but suppose we use the simplest strategy and associate each primitive with the smallest enclosing octree cube. After building the octree, to cull off-screen geometry we begin at the root node and recursively subdivide the octree, classifying each octree node encountered as entirely outside the frustum, entirely inside the frustum, or partially inside the frustum. When a node lies entirely outside the frustum, it is ignored, effectively culling the sub-octree that it represents. When a node lies entirely inside the frustum, all primitives in that sub-octree are known to be on-screen, so they are rendered. Primitives associated with a node that lies partially inside the frustum need to be tested individually to see if they are on-screen; off-screen primitives are culled, on-screen primitives are rendered. Non-leaf nodes that are partially inside the frustum are subdivided, and this same culling procedure is applied recursively to their children. When this recursive subdivision procedure finishes, all on-screen geometry has been rendered and all off-screen geometry has been culled.

This hierarchical culling procedure can provide substantial acceleration when a large fraction of the scene is off-screen, since it hierarchically culls large parts of the model without having to consider their primitives one by one. Traversal of the octree is very efficient, because the procedure visits only octree nodes that intersect the viewing frustum and their children, and octree nodes are visited at most once. However, this procedure does not address the more challenging problem of culling on-screen geometry that is occluded by objects that are closer to the observer. Thus, it fails to exploit much of the available object-space coherence in densely occluded scenes.

Meagher's Octree Rendering Algorithm

Previous work on visibility includes a volume rendering algorithm that efficiently culls occluded regions of the model, culling hierarchically in both object space and image space. In the early 1980's Meagher devised a very efficient algorithm for displaying three-dimensional models represented as octrees, where non-empty leaf nodes in the octree represent opaque cubes of a specified color [Meagher81, Meagher82a, Meagher82b, Meagher85, Meagher91].

A scene is rendered by traversing the octree nodes in front-to-back order[3] and projecting the bounding cube of each non-empty octree node onto the screen, where its silhouette is a convex polygon. The screen is represented as a quadtree, and as with tiling polygons using the Warnock algorithm, the polygonal projections of octree nodes are hierarchically tiled into the quadtree. Since octree nodes are traversed in strictly front-to-back order, once a quadtree cell has been completely covered by the projections of opaque octree nodes, that region of the screen is known to hide any octree node that is encountered thereafter, allowing trivial culling of octree nodes without depth comparisons [Meagher81]. Since octree nodes are nested, this procedure culls hidden regions of object space hierarchically. As with the Warnock algorithm, the quadtree tiling procedure also culls hierarchically in image space. In short, Meagher's volume rendering algorithm effectively exploits both object-space and image-space coherence, culling hierarchically in both domains. As Meagher points out, computational requirements for rendering depend primarily on the visible complexity of the scene rather than the scene's overall complexity [Meagher82b]. Although this algorithm is only capable of rendering volume models represented as octrees, a geometric model can be displayed by first converting it to an octree representation[4] and then displaying the octree. However, when a geometric model is converted to a volume model, some geometric and shading information is lost or degraded (e.g. surface orientation), and consequently, conventional rendering algorithms for geometric models are capable of producing more accurate results. Another shortcoming is that this is not a good general-purpose rendering algorithm, because octrees are not an efficient or accurate representation for relatively simple geometric models.

Potentially Visible Set Methods

To exploit object-space coherence in rendering architectural models, Airey et al. [Airey90, Airey-Rohlf-Brooks90] and subsequently Teller and Séquin [Teller-Sequin91, Teller-Sequin92, Teller92] proposed dividing models up into disjoint cells and precomputing the *potentially visible set* (PVS) of primitives that are visible from each cell. To render the scene from an arbitrary viewpoint, the cell containing the viewpoint is determined, and then the primitives in that cell's PVS are rendered, preferably with a graphics accelerator. This approach has proved to be very effective for walk-through animation of architectural models where cells correspond to rooms. Teller and Séquin report that their implementation

[3] The traditional algorithm for efficient front-to-back traversal of octree nodes with respect to an arbitrary viewpoint is described in [Foley-et-al90], page 695, which we reiterate in §3.3.1.

[4] Kaufman discusses tiling methods for converting geometric models to volume models in [Kaufman86].

running on a multiprocessor graphics workstation can display building models consisting of hundreds of thousands of polygons at sub-second frame rates. Luebke and Georges have developed a faster, simpler method than that of Teller and Séquin for computing PVSs [Luebke-Georges95]. They report that their method is fast enough to permit on-the-fly PVS computations during walk-through animation of complex architectural models, eliminating the need to precompute PVS data and extending the usefulness of the method to dynamic scenes.

However, PVS methods suffer from an important limitation. To achieve full acceleration, the cells must be regions of space that are almost entirely enclosed by occluding surfaces, so that most cells are hidden from most other cells. The method often works well for architectural models, since the cells can be rooms that are enclosed except for windows and doors, but for outdoor scenes and other non-architectural environments, PVS methods are not nearly as effective.

Naylor's BSP-Tree Visibility Algorithm

Visibility algorithms that cull in both object space and image space include a variation of the Weiler-Atherton algorithm [Weiler-Atherton77] devised by Naylor [Naylor92b]. According to this method, a polygonal scene is organized into an object-space hierarchy represented as a three-dimensional BSP tree [Fuchs-Kedem-Naylor80], and a recursive subdivision procedure that is analogous to the Weiler-Atherton control structure produces an object-precision visibility map in the form of a two-dimensional BSP tree. Naylor contends that representing the scene as a BSP tree permits the basic geometric operations to be performed simpler, faster, and with better numerical stability. One obvious advantage of using BSP trees is that it permits rapid traversal of primitives in strict front-to-back order. More importantly, bounding volumes defined by partitioning planes in the scene BSP tree permit hierarchical object-space culling, so many hidden polygons in the tree may never need to be considered. However, the algorithm apparently does not cull subtrees occluded by multiple faces in the final visibility map, unless it happens to be possible to merge the multiple faces into a single face at a higher level of the image-space BSP tree. In view of this shortcoming, the effectiveness of the algorithm for densely occluded scenes is difficult to estimate, and Naylor only reports performance figures for simple scenes [Naylor92b]. Although the effectiveness of the algorithm for densely occluded scenes has yet to be demonstrated, this is one of very few visibility algorithms that cull in both object space and image space, and it appears to be a promising approach to accelerating visibility computations in complex scenes. Since the algorithm generates an object-precision visibility map, it supports high-quality antialiasing.

2.1.7 Summary of Visibility Discussion

Efficient visible-surface determination in very complex, densely occluded scenes requires efficient culling of hidden geometry. In reviewing traditional visibility algorithms we have seen that, as Sutherland, Sproull, and Schumacker observed many years ago [Sutherland-et-al74], exploiting *coherence* is the key to efficient culling. The various forms of coherence that a visibility algorithm can exploit fall into three general categories: object-space, image-space, and temporal coherence. Ideally, a visibility algorithm should exploit all of these forms of coherence. We have seen that neither of the two dominant visibility algorithms, z-buffering and ray casting, is capable of efficiently rendering densely occluded scenes. Traditional z-buffering effectively exploits image-space coherence but not object-space coherence. Conversely, ray casting through a spatial subdivision effectively exploits object-space coherence but not image-space coherence. Although image-space culling with a ZZ-buffer [Salesin-Stolfi89] improves the efficiency of ray casting, it is still necessary to traverse every scene primitive at every frame, which impairs performance for densely occluded scenes. Previous work includes various attempts to simultaneously exploit both object-space and image-space coherence. In the domain of volume rendering, Meagher's algorithm effectively exploits both of these forms of coherence, but this method is not directly applicable to rendering geometric models [Meagher82b]. Potentially visible set methods effectively exploit both object-space and image-space coherence (assuming z-buffering of primitives), but their usefulness is limited to models having certain geometric characteristics [Teller92]. Naylor's BSP-tree rendering algorithm culls in both object space and image space, but it has not actually been demonstrated to efficiently process densely occluded scenes [Naylor92b]. Thus, our review of previous work shows that there is a need for a visibility algorithm applicable to arbitrary geometric models that effectively exploits both object-space and image-space coherence to accelerate visibility computations in densely occluded scenes. For motion sequences, there is the additional challenge of exploiting temporal coherence. In chapter 3 we present a visibility algorithm that can exploit all three of these forms of coherence.

2.2 Antialiasing of Computer-Generated Images

Generating high-quality images of very complex scenes is particularly challenging because dozens or even hundreds of primitives may be visible within individual pixels. Proper filtering of this visible geometry requires a thorough understanding of aliasing, the phenomenon responsible for jaggies and other disturbing visual artifacts. In this section we discuss the causes of aliasing and traditional approaches for dealing with the problem. We

also include a discussion of interval analysis, since the rendering algorithm we present in chapter 5 applies interval methods to antialiasing.

2.2.1 The Causes of Aliasing

The potential for aliasing arises in computer graphics because the mathematical representations that we ordinarily use to describe images (e.g. polygons) contain energy at arbitrarily high spatial frequencies, while the sampled rasters we use to display images are limited to a finite range of spatial frequencies. Let $I(x, y)$ be the vector-valued function that gives the color of each point in a rectangle of $\Re^2$ corresponding to the screen for the idealized mathematical representation of a computer-graphics image. If we compute a raster image by directly sampling $I(x, y)$ at the center of each output pixel, then any spatial frequency content in $I(x, y)$ beyond half the sampling rate will alias to a lower frequency and cause disturbing visual artifacts [Foley-et-al90]. In particular, this is the cause of aliasing artifacts in z-buffer images.

2.2.2 Approaches to Antialiasing

There are three basic approaches for dealing with the aliasing problem. The first approach is to adjust the number, locations, or weights of the samples to attenuate the visible aliased energy. The second approach is to try to detect aliasing artifacts in the rendered image and remove them by post-processing. The third approach, and the only one capable of guaranteed accuracy, is to compute or approximate the convolution $I(x, y) * f(x, y)$ of the image $I(x, y)$ with a low-pass filter $f(x, y)$ at each output pixel. We review each of these methods in turn.

Adjusting Samples and Filter Kernels

The first approach to antialiasing, adjusting the number and location of samples and the filter kernels used to combine them, is probably the most widely used approach, and it can go a long way toward reducing the severity of aliasing artifacts. The simplest strategy is uniform oversampling, where the image is generated at higher resolution than the output image, and each pixel in the output image is obtained by filtering a neighborhood of the higher-resolution image. With or without oversampling, for any fixed sampling rate the patterned aliasing artifacts that arise with regular sampling can be converted into less disturbing noise by placing samples stochastically [Cook-et-al84, Cook86, Dippe-Wold85]. However, it is not known in advance what sampling rate will be required for any particular

region of the image, and any algorithm that uses a fixed sampling rate (e.g. [Carpenter84]) will be unable to deal with pixels within which a great number of primitives is visible. While stochastic sampling lessens the problem somewhat by converting aliasing to noise, the noise may be unacceptably high in some regions of the image. In short, this approach provides no guarantees about the quality of the result, and when applied to rendering very complex scenes, it may produce unacceptable errors.

Some authors have suggested using the variance of a collection of rays through a pixel to determine a local sampling rate for ray tracing [Lee-Redner-Uselton85, Dippe-Wold85], and this can produce good results in many cases. A serious limitation, however, is that the accuracy of the result is proportional to the square root of the number of samples. It can be shown using analysis methods described by Lee et al. [Lee-Redner-Uselton85] that it may be necessary to cast hundreds of rays through a pixel having complex visible geometry to achieve high confidence that the pixel value is accurate. Thus, this approach is not an efficient way to make very accurate images. Moreover, accuracy is statistical and some pixels will be expected to have significant error. In short, traditional oversampling methods can be applied to antialiasing scenes in which numerous primitives are visible at individual pixels, but they have shortcomings in both quality and efficiency.

Post-Processing

The second approach to combating aliasing, post-processing, has limited potential because it begins after the sampling process. If geometric primitives are large compared to pixels, then a post-process can sometimes effectively infer visible edges from the sampled image and soften them to attenuate aliasing artifacts [Bloomenthal83]. However, if large numbers of primitives are visible within individual pixels, too much information is lost in the sampling process to allow a post-process to compute an acceptable reconstruction. To improve antialiasing of edges, Fujimoto and Iwata keep track of visible edges in the course of rendering the original image and then redraw visible edges with an antialiased line tiler [Fujimoto-Iwata83]. While this may work well when only one or two edges cross a pixel, it would not be expected to produce good results when numerous edges are visible within a pixel.

Convolution before Sampling

The third approach to antialiasing, convolution before sampling, was first advanced by Crow [Crow77] and Catmull [Catmull78] for images of polygonal scenes, and it is the only technique that is, in principle, capable of eliminating aliasing entirely. From a theoretical

perspective, if we convolve $I(x, y)$ with the appropriate *sinc* function, it will be low-pass filtered below the Nyquist rate before sampling and no aliasing will occur [Foley-et-al90]. From a practical perspective, however, some authors (e.g. [Dippe-Wold85]) have observed that the ideal *sinc* function generates ringing (Gibbs phenomenon) at step edges, and have suggested other filters such as triangular filters, Gaussians, raised cosine filters, Hamming windows, etc. Whatever filter is chosen, rendering with this approach requires identifying the visible geometric primitives affecting each output pixel and then filtering them. Visible-surface algorithms that are capable of finding all the geometric primitives potentially affecting a single pixel include [Catmull78], [Catmull84], [Weiler-Atherton77], [Warnock69], [Sharir-Overmars92], [Sechrest-Greenberg82], [Sequin-Wensley85], and [Naylor92b]. Once the visible polygons affecting a pixel have been identified, they need to be convolved with the desired filter and summed. Fast algorithms using lookup tables have been developed to compute the convolution quickly for arbitrary filters with flat-shaded polygons [Feibush-Levoy-Cook80, Abram-et-al85], and for Gaussian or box-filtered texture-mapped polygons [Williams83, Crow84].

Coverage Masks

Algorithms that perform antialiasing by oversampling and filtering are often able to accelerate visibility and filtering operations on geometry in pixel neighborhoods by using specially constructed bit masks called *coverage masks* [Carpenter84, Sabella-Wozny83, Fiume-et-al83, Fiume91]. Typically, the bit pattern of a coverage mask indicates which samples within a pixel are covered by a particular geometric entity – the samples lying inside an edge, the visible samples lying inside a polygon, etc. Representation as bit masks permits various coverage and visibility operations to be performed very efficiently with boolean mask operations. For example, all possible tiling patterns for a single edge crossing a grid of samples within a pixel can be precomputed and later retrieved, indexed by the points where the edge intersects the pixel's border. The coverage mask for a convex polygon can be constructed by looking up the coverage masks of its edges and ANDING them together. Once the coverage masks of the polygons that intersect a pixel have been constructed, coverage masks for the *visible* samples on each polygon can be constructed by performing simple compositing operations on their masks, processing them either front to back or back to front. These steps are essentially how Carpenter's A-buffer algorithm determines visible subpixel samples on each polygon that intersects a pixel [Carpenter84]. Carpenter also applies coverage masks to accelerate box filtering of visible samples. For each polygon that is visible at a pixel, a single shading value is computed and the contribution of this shading value to the pixel is weighted by the bit count of the polygon's coverage mask. Abram, Westover,

and Whitted advance similar methods that permit convolution with arbitrary filter kernels, jitter, and evaluation of simple shading functions by table lookup [Abram-et-al85].

2.2.3 Interval Analysis

Interval analysis is a branch of applied mathematics that was originally developed to analyze error resulting from the limited precision with which real numbers can be represented in digital computers. In addition to this application, the method has been applied to solving equations and systems of equations, optimization, differential equations, and integral equations [Moore66, Moore79, Alefeld-Herzberger83]. The method employs an alternative number system based on *interval numbers*, each interval number corresponding to a range of real values which is specified by a pair of numbers [a,b], the lower and upper bounds of the interval. It is often straightforward to establish bounds for interval expressions. For example, the expressions for interval addition, subtraction, multiplication, and division are as follows [Mitchell91].

```
[a,b] + [c,d] = [a+c,b+c]
[a,b] - [c,d] = [a-d,b-c]
[a,b] * [c,d] = [min(ac,ad,bc,bd),max(ac,ad,bc,bd)]
if 0 ∉[c,d],  [a,b] / [c,d] = [a,b] * [1/d,1/c]
```

In recent years, interval analysis has been applied to a variety of problems in computer graphics [Snyder92, Mitchell91], particularly to performing geometric operations with guaranteed accuracy such as finding ray-surface intersections. In chapter 5 we present an error-bounded rendering algorithm that uses interval methods to evaluate shading functions with guaranteed accuracy.

Chapter 3

Hierarchical Z-Buffer Visibility

3.1 Overview

As discussed in chapter 2, for rendering applications in which speed is the predominant consideration rather than image quality, point-sampling algorithms such as z-buffering are the most practical approach. In this chapter we present a very efficient algorithm for generating z-buffer images of complex, densely occluded scenes. The algorithm achieves efficiency by applying hierarchical methods to culling hidden geometry in both object space and image space, and by rendering visible geometry with the speed of traditional z-buffer scan conversion.

To enable hierarchical culling of geometry in hidden regions of object space, we organize scene geometry in an octree. To render a frame, we recursively subdivide the octree in front-to-back order as with Meagher's volume rendering algorithm [Meagher82b]. During traversal of the octree, octree nodes hidden by the z-buffer are culled, and geometry inside visible octree nodes is rendered into the z-buffer. Culling hidden octree nodes removes most hidden geometry, but some geometry still overlaps when projected onto the screen. To accelerate culling of remaining hidden geometry, z-buffer depth samples are maintained in a pyramid, which permits hierarchical culling in image space. Thus, the algorithm exploits coherence with both object-space and image-space hierarchies. For animation, we are also able to exploit temporal coherence by using geometry that was visible in the preceding frame to construct a starting point for the algorithm. This appears to be the first visibility algorithm that materially profits from object-space, image-space, and temporal coherence simultaneously. For very densely occluded scenes, the algorithm sometimes achieves orders of magnitude acceleration compared with ordinary z-buffer scan conversion. Although the algorithm performs general-purpose z-buffer rendering, we refer to it as a *visibility* algorithm because its novel features accelerate visibility operations and are unrelated to shading.

3.2 Introduction

Extremely complex scenes offer interesting challenges for visibility algorithms. Consider, for example, an interactive walk-through of a detailed geometric model describing an entire city complete with gardens and trees, buildings with furnishings, etc. Traditional visibility

algorithms running on contemporary computers cannot come close to rendering scenes of this complexity at interactive rates, and it will be a long time before faster hardware alone will provide the needed performance. In order to get the most out of available hardware, we need faster algorithms that exploit properties of the visibility computation itself. Our ultimate objective is a visibility algorithm that does work proportional to the *visible complexity* of the scene in the output image, rather than the complexity of the overall geometric model. While this objective is unattainable because it would allow no work to be expended in culling hidden geometry, culling can be done very efficiently using hierarchical methods.

Reiterating our review of visibility in chapter 2, the key to accelerating visibility computations is exploiting three forms of coherence: object-space coherence, image-space coherence, and for animation sequences, temporal coherence. Ideally, a visibility algorithm should be able to exploit all of these forms of coherence. However, no traditional visibility algorithm succeeds in doing this. Traditional z-buffering effectively exploits image-space coherence but not object-space coherence. Conversely, ray casting through a spatial subdivision effectively exploits object-space coherence but not image-space coherence. Although image-space culling with a ZZ-buffer [Salesin-Stolfi89] improves the efficiency of ray casting, it is still necessary to traverse every primitive in the scene at every frame, which impairs performance for densely occluded scenes. Prior work includes various attempts to simultaneously exploit both object-space and image-space coherence. In the domain of volume rendering, Meagher's algorithm effectively exploits both of these forms of coherence, but this method is not directly applicable to rendering geometric models [Meagher82b]. Potentially visible set methods effectively exploit both object-space and image-space coherence (assuming z-buffering of primitives), but their usefulness is limited to models having certain geometric characteristics [Airey90, Teller-Sequin91, Teller92]. Naylor's BSP-tree visibility algorithm culls in both object space and image space, but it has not actually been shown to efficiently process densely occluded scenes [Naylor92b]. In short, no existing visibility algorithm has been demonstrated to effectively exploit both object-space and image-space coherence in processing arbitrary geometric models. For animation, there is the additional challenge of harnessing temporal coherence, which traditional algorithms rarely exploit in practice.

In this chapter, we present a z-buffer visibility algorithm that exploits object-space coherence as effectively as ray casting through a spatial subdivision, exploits image-space coherence even more effectively than traditional incremental scan conversion, and for animation sequences is also able to exploit frame-to-frame coherence. To exploit object-space coherence, we use an octree spatial subdivision of the type commonly used to accelerate ray

tracing [Reddy-Rubin78, Rubin-Whitted80, Glassner84, Kay-Kajiya86, Kaplan87, Jevans-Wyvill89]. To exploit image-space coherence, we augment traditional z-buffer scan conversion with an image-space z-pyramid that allows us to cull hidden geometry very rapidly. To exploit temporal coherence, we use the geometry that was visible in the preceding frame to construct a starting point for the algorithm. The result is a z-buffer visibility algorithm which is orders of magnitude faster than traditional z-buffering for some densely occluded scenes we have experimented with. The algorithm is not difficult to implement and it works for arbitrary geometric models consisting of polygons and other primitives that can be efficiently scan converted. Moreover, the algorithm has modest memory requirements and it is amenable to parallel computation and to implementation in hardware.

In §3.3 we present the hierarchical z-buffer visibility algorithm, beginning with the data structures that it employs to exploit object-space, image-space, and temporal coherence. In §3.4 we discuss methods for building and maintaining the octree spatial subdivision. In §3.5 we describe our implementation and show results for some complex models containing hundreds of millions of polygons. Finally, in §3.6 we state our conclusions.

3.3 The Hierarchical Z-Buffer Visibility Algorithm

The hierarchical z-buffer visibility algorithm uses an octree spatial subdivision to exploit object-space coherence, a z-pyramid to exploit image-space coherence, and to exploit temporal coherence in animation sequences, it keeps track of octree nodes that were visible in the preceding frame. While the full advantage of the algorithm is realized by using all three of these together, the octree and the z-pyramid can also be used separately. Whether used separately or together, these data structures make it possible to compute exactly the same result as ordinary z-buffering at less computational expense.

3.3.1 The Object-Space Octree

As discussed in §2.1.4, octrees have been used previously with great effectiveness to accelerate ray casting and ray tracing [Rubin-Whitted80, Glassner84, Kay-Kajiya86, Kaplan87, Jevans-Wyvill89] and rendering of volume models [Meagher82b]. With some important modification, the principles of this previous work can be applied to z-buffer rendering. The result is an algorithm that can accelerate z-buffering by orders of magnitude for models with sufficient depth complexity. By depth complexity, we mean the average number of primitives that overlap at each pixel on the screen.

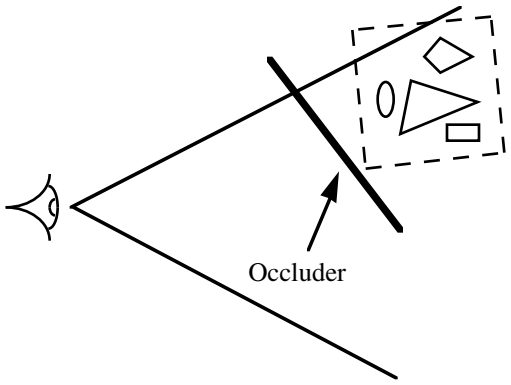


Figure 3.1 If a cube (or other bounding volume) is hidden, then all geometry it contains is also hidden.

In order to be precise about the algorithm, we begin with some simple definitions. We will say that a polygon or other primitive is hidden with respect to a z-buffer if no depth samples on the tiled primitive are closer to the observer than the corresponding z values already in the z-buffer. Similarly, we will say that a cube is hidden with respect to a z-buffer if all of its faces are hidden polygons. Finally, we will call an octree node hidden if its bounding cube is hidden. Note that these definitions depend on the sampling of the z-buffer. A primitive that is hidden at one z-buffer resolution may not be hidden at another.

Testing Octree Cubes for Visibility

With these definitions, we can state the basic observation that makes it possible to combine z-buffering with an octree spatial subdivision. As schematically illustrated in figure 3.1, if a cube is hidden with respect to a z-buffer, then all geometry fully contained in the cube is also hidden. Actually, this principle applies to any bounding volume, not just a cube. It follows that if we tile the faces of a cube and determine that it is hidden, we can safely ignore all the geometry contained in that cube. The following pseudocode outlines a procedure for determining whether an octree cube is visible, returning either TRUE or FALSE.

```
IsCubeVisible(OctreeNode N)
{
   if N is completely outside the viewing frustum
   then return FALSE
   if the viewpoint is inside N
   then return TRUE
   if N intersects the "near" face of the viewing frustum
   then return TRUE
   for each front face F of N {
      if F is visible at one or more pixels
      then return TRUE
   }
   return FALSE
}
```

This procedure's first step is determining whether a node's bounding cube intersects the viewing frustum. We have developed a fast algorithm for the more general problem of detecting intersection of an axis-aligned rectangular solid and a convex polyhedron. Our implementation uses this method, which is presented in the appendix and also in [Greene94]. With this method, determining whether a cube intersects the viewing frustum requires evaluating between one and thirty inequalities derived from line and plane equations.

The Basic Rendering Algorithm

Given this cube-visibility test, the basic rendering algorithm is easy to construct. We begin by organizing scene geometry into an octree. There are various ways of building an octree, but for the moment let's assume that each primitive is associated with the smallest enclosing octree cube. At the beginning of a frame, we clear the image buffer to the background color (or image) and clear the z-buffer to the far clipping plane. Then, starting at the root node, we process the octree according to the following recursive steps. If the octree node is not visible, either outside the viewing frustum or hidden by the z-buffer, we are done. Otherwise, we tile any primitives associated with the octree node into the z-buffer and then recursively process its children, if any, in front-to-back order using this same procedure. When the recursion finishes, we have a standard z-buffer image of the scene. It should be noted that Meagher's volume rendering algorithm, reviewed in section §2.1.6, also uses this depth-first recursive subdivision procedure to traverse an octree [Meagher82b]. The procedure for rendering a scene is outlined in pseudocode below.

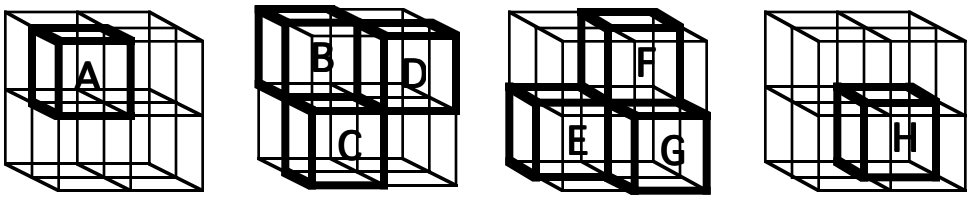


Figure 3.2 Ordering of a cube's octants into four clusters of equivalent visibility priority: {A},{B,C,D},{E,F,G},{H}.

```
RenderScene(OctreeNode Root)
{
   clear image buffer to background
   clear z-buffer to far clipping plane
   ProcessOctreeNode(Root)
}

ProcessOctreeNode(OctreeNode N)
{
   if IsCubeVisible(N) returns FALSE
   then return
   for each primitive P associated with N
      tile P into the z-buffer
   for each child C of N in front-to-back order
      ProcessOctreeNode(C)
}
```

Note that procedure `ProcessOctreeNode()` requires front-to-back ordering of the octants of an octree node with respect to the viewpoint. This ordering during recursive subdivision guarantees strict front-to-back traversal of nodes in the octree. This is crucial to the algorithm's efficiency, because it guarantees that any geometry that can occlude a node is tiled into the z-buffer before that node is processed. As a result, even though the z-buffer is usually only partially formed when cube-visibility tests are performed, these tests are definitive and succeed in culling all octree nodes that are hidden.

The nested, rectilinear structure of an octree makes it easy to establish front-to-back ordering of octants. We use the method described in [Foley-et-al90], whereby the octant corresponding to the nearest corner of the cube is known to be "frontmost," and the three octants which share a face with the frontmost octant all have the same visibility priority, just behind frontmost. Symmetrically, the octant opposite frontmost is "backmost," and the three octants which share a face with the backmost octant all have the same visibility priority, in front of backmost and behind all the other octants. Thus, this procedure clusters the eight octants of a cube into four groups of equivalent visibility priority, as illustrated in

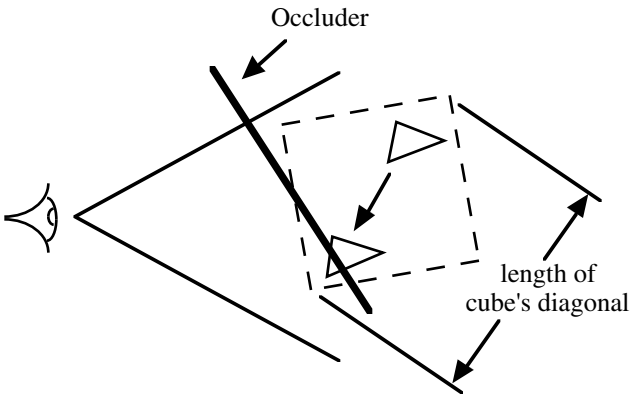


Figure 3.3 Any primitive inside a visible cube is within the length of the cube's diagonal of being visible.

figure 3.2. This ordering algorithm works regardless of the cube's orientation and whether or not the viewpoint lies inside the cube.

The basic rendering algorithm outlined above has some desirable properties. First of all, the algorithm only traverses and tiles primitives contained in octree nodes that are visible. Some of the tiled primitives may be hidden, but as illustrated in figure 3.3, each primitive in a visible octree cube is "nearly visible" in the following sense: there is some place we could move it where it would be visible which is no farther away than the length of the diagonal of its bounding cube. Thus, the algorithm only tiles primitives which are visible or nearly visible. In addition, the algorithm only visits visible octree nodes and their hidden children, so it does not waste time on irrelevant portions of the octree. Finally, the algorithm never visits an octree node more than once during the rendering of a frame. This stands in marked contrast to ray casting through an octree, where the root node is visited for every pixel rendered, and other nodes may be visited thousands of times. As a result of these properties, the algorithm is very efficient at both culling hidden geometry and traversing visible geometry.

Building the Octree

Recall that scene geometry must be organized into an octree prior to rendering. We can construct the octree with a simple recursive procedure. Beginning with a root cube large enough to enclose the entire model and the complete list of geometric primitives, we perform the following steps recursively. If the number of primitives is sufficiently small, say less than or equal to m, we associate all of the primitives with the cube and return. Otherwise, we associate with the cube any primitive that intersects any of the three axis-aligned planes that

bisect the cube. We then subdivide the octree cube and call the procedure recursively with each of the eight child cubes and the list of primitives that fit entirely in that cube. When this recursive procedure finishes, each primitive is associated with the smallest enclosing octree cube in the hierarchy and each leaf node contains a maximum of m primitives.

One weakness of this algorithm for building octrees is that it associates some small primitives with large cubes if the primitives happen to intersect the planes that separate the cube's children. For example, a small triangle that crosses the center of the root cube will be associated with the root cube and it will need to be rendered anytime the entire model is not hidden. To avoid this behavior, there are two basic choices. One alternative is to clip the problematic small primitives so they fit into much smaller octree cubes. This has the disadvantage of increasing the number of primitives in the model. The other alternative is to place some primitives in multiple octree cubes. We chose to implement the latter alternative. To do this, we modify the recursive construction of the octree as follows. If we find that a primitive intersects a cube's dividing planes but is small compared to the cube, then we no longer associate the primitive with the whole cube. Instead we associate it with each of the cube's children that the primitive intersects.[1] This subdivision procedure continues recursively until the primitive is associated with cubes of the appropriate size. The same strategy can also be used to place long, skinny primitives into multiple cubes.

Primitives that are associated with more than one octree node may be encountered more than once during rendering. To avoid rendering a primitive more than once, we mark it with the frame number when it is rendered. This permits us to know whether a primitive has already been rendered in the current frame, without having to clear a flag for each primitive at the beginning of each frame.

3.3.2 The Image-Space Z-Pyramid

The basic rendering algorithm described in the preceding section spends most of its time tiling cube faces and primitives. The object-space octree allows us to cull large hidden portions of the model at the cost of tiling the faces of the visible octree cubes and their hidden children. Actually, only hidden cube faces need to be tiled completely, since encountering a visible pixel on a face establishes that a cube is visible, permitting tiling to stop. Even so, tiling of cube faces requires considerable computation because many faces are large, and since cubes are nested, faces may overlap densely on the screen. Primitives associated with visible cubes must also be tiled, and they also overlap on the

[1] For polygonal primitives, we test for cube-polygon intersection using the method presented in the appendix.

screen, sometimes densely. Even if we employ fast incremental methods to accelerate scan conversion as described in §2.1.5, tiling of cube faces and primitives is costly, because it requires traversing each primitive pixel by pixel, even if it is entirely or mostly hidden. Thus, there is a need to reduce the tiling requirements of the algorithm. To accomplish this, we apply hierarchical methods to accelerate culling of hidden geometry in image space.

To support hierarchical culling in image space, we maintain z-buffer depth samples in a *z-pyramid*, an image pyramid similar to those used in texture mapping (e.g. [Williams83]) and image processing (e.g. [Burt-Adelson83]). Frequently, the z-pyramid makes it possible to conclude very quickly that a cube face or primitive is hidden, making pixel-by-pixel scan conversion unnecessary.

Figure 3.4 shows a z-buffer image of a densely occluded scene and its corresponding z-pyramid. The finest level of the pyramid is an ordinary z-buffer. At all other levels, each z sample is the farthest z from the observer in the corresponding 2×2 window of the next finer level. Every entry in the pyramid therefore represents the farthest z sample for a square region of the screen. At the coarsest level of the pyramid there is a single z value that is the farthest z sample from the observer in the whole image. A z-pyramid requires 1/3 more memory than a conventional z-buffer.

Maintaining the z-pyramid is an easy matter. Every time we modify the z-buffer, we propagate the new z value through to coarser levels of the pyramid. As soon as we reach a level where the entry in the pyramid is already as far away as the new z value, propagation can stop, since coarser levels of the pyramid will not be affected.

The method we use to test the visibility of primitives with respect to the z-pyramid is illustrated schematically in figure 3.5. First, we find the finest-level sample of the pyramid whose corresponding image region encloses the primitive. Then, if the nearest z value of the primitive is farther away than this z-pyramid sample, we know immediately that the entire primitive is hidden. Very often, we are able to show with this single depth comparison that an entire cube, cube face, or primitive is hidden.

While the basic z-pyramid test can cull a substantial fraction of hidden primitives, it suffers from a similar difficulty to the basic octree method. Because of the structure of the pyramid, a small primitive covering the center of the image will be compared to the z value at the coarsest level of the pyramid. While the test is still accurate in this case, it is not very powerful.

A definitive visibility test can be constructed by applying the basic test recursively through the pyramid. If the basic test fails to show that a primitive is hidden, we go to the next finer level in the pyramid where the parent pyramid region is divided into four quadrants. Here we attempt to prove that the primitive is hidden in each of the quadrants

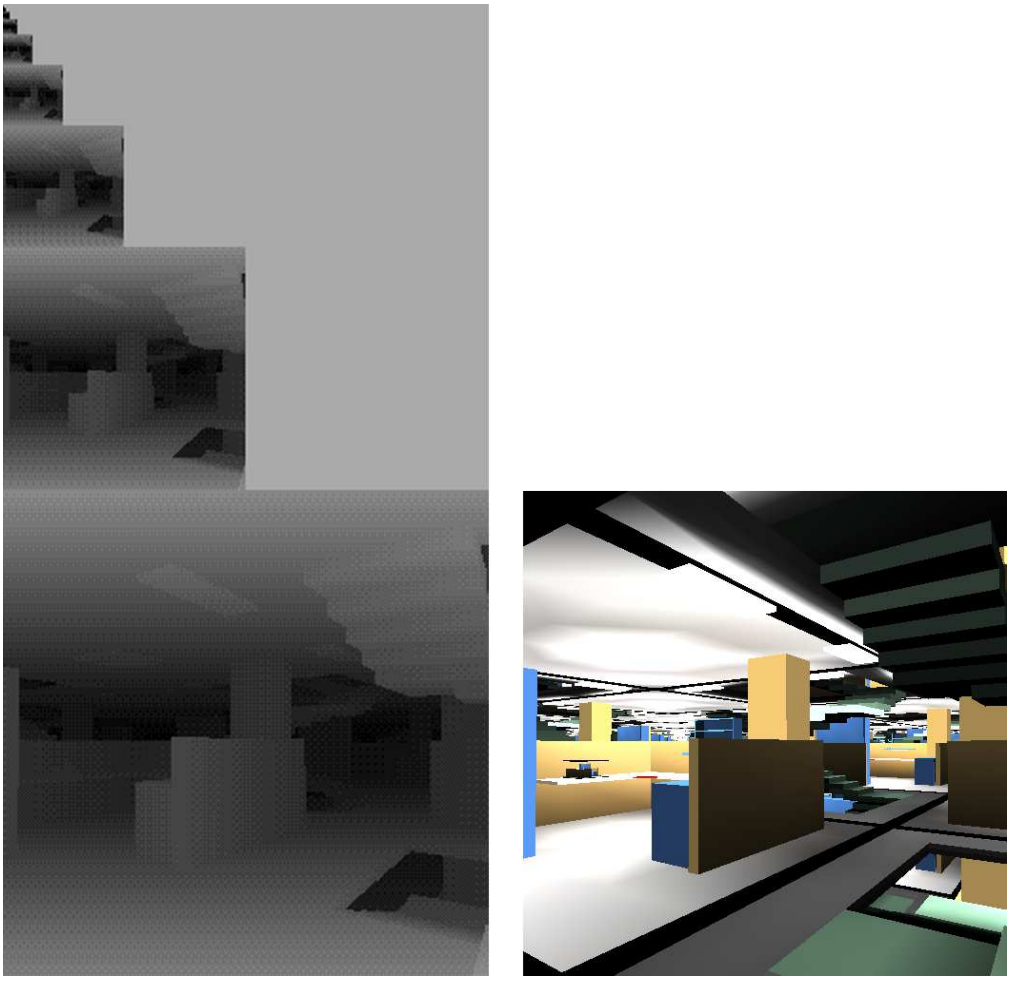


Figure 3.4 A scene and its corresponding z-pyramid. The finest level of the pyramid is the ordinary z-buffer. At all other levels, each z sample is the farthest z from the observer in the corresponding 2×2 window of the next finer level. Every entry in the pyramid therefore represents the farthest z for a square region of the screen.

that it intersects. For each of these quadrants, we compare the closest z value of the primitive within the quadrant to the value in the pyramid. If the pyramid value is closer, we know that the primitive is hidden in the quadrant. If we fail to prove that the primitive is hidden in one of the quadrants, we go to the next finer level of the pyramid for that quadrant and try again. Ultimately, we either prove that the entire primitive is hidden, or we recurse down to the finest level of the pyramid and find a visible pixel. This recursive procedure is essentially a logarithmic search for a visible pixel. Note that if we find all visible pixels on a primitive this way, we are tiling the primitive hierarchically. The polygon tiling algorithm we present in chapter 4, while not a z-buffer algorithm, performs tiling hierarchically and

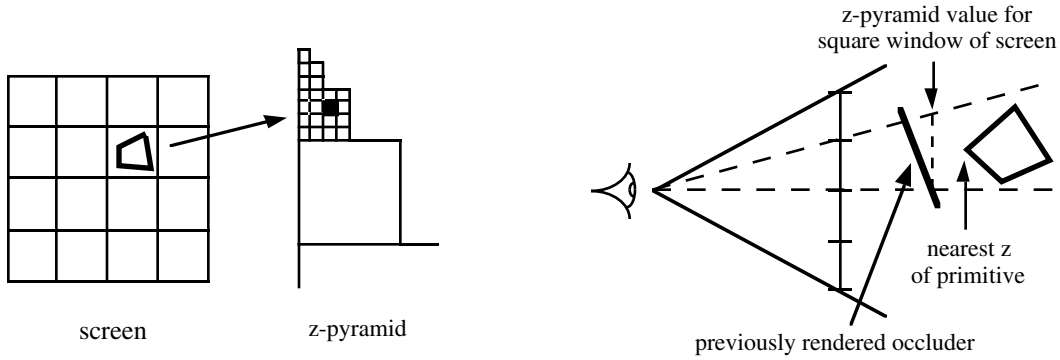


Figure 3.5 The procedure for determining whether the z-pyramid hides a primitive. Using the z-pyramid, often a single depth comparison can show that an entire cube, cube face, or primitive is hidden. If a single test fails to show that one of these geometric entities is hidden, we can apply the same test recursively in smaller windows of the screen.

finds visible samples by logarithmic search.

In practice, it is expensive to perform definitive visibility tests on primitives in this hierarchical manner because it is necessary to determine which quadrants intersect the primitive and to find the farthest z value of the primitive within each intersected quadrant. An alternative is to perform a much faster but non-definitive test which is often able to detect hidden primitives. According to this method, we construct a primitive's screen-space bounding rectangle and set its depth to the primitive's closest z value. Then the recursive subdivision procedure described above can quickly establish whether there is at least one visible pixel on the bounding rectangle. If not, we know that the corresponding primitive is hidden. When this test fails to prove that a primitive is hidden, we revert to ordinary z-buffer scan conversion to establish visibility of cube faces and to tile primitives into the z-buffer. Our current implementation uses this method.

One interesting variation of this hierarchical culling procedure is to maintain a "znear pyramid" of depth samples in addition to the "zfar pyramid" described above. Like the zfar pyramid, the finest level of the znear pyramid is the ordinary z-buffer, which both pyramids share. At all other levels, each sample in the znear pyramid is the nearest z sample from the observer in the corresponding 2×2 window of the next finer level. Just as the zfar pyramid can establish that a primitive is hidden in a region of the screen, the znear pyramid can

establish that a primitive is at least partially visible in a region of the screen.[2] Thus, visible primitives can usually be identified without subdividing down to the pixel level, which is always necessary if we only consult the zmax pyramid. We implemented dual z-pyramids and found that the overhead of maintaining the second pyramid cancelled out the time saved by improved culling efficiency, so overall performance remained about the same. However, this was not a careful implementation, so no definitive conclusion should be drawn.

It is worth mentioning one other variation of the hierarchical culling procedure. If, during pixel-by-pixel scan conversion of front cube faces, a z-buffer depth comparison shows that a face is hidden at a particular pixel, we know that we are completely done with that pixel, because front-to-back traversal of octree cubes guarantees that any primitives subsequently traversed will be farther away at that pixel. We can keep track of these "completed" pixels (e.g., by assigning them a special value in the z-pyramid), and this status information can be propagated to coarser cells in the pyramid, since a pyramid cell is "complete" if its child cells are complete. This can accelerate culling because it often eliminates the need to perform depth comparisons. For example, if the z-pyramid sample for the screen region which encloses an entire primitive is complete, we know immediately that the primitive is hidden, and there is no need to even establish the primitive's depth. We have not implemented this variation of hierarchical culling, so we can not report whether it improves the algorithm's overall performance.

3.3.3 Exploiting Temporal Coherence

We turn now to a method for exploiting frame-to-frame coherence during generation of animation sequences that can be applied if a z-buffer hardware accelerator is available. Note that the basic hierarchical visibility algorithm can not make use of ordinary z-buffer accelerators because they do not maintain a z-pyramid and they can not perform visibility tests on cube faces.

When we render an image of a densely occluded scene with the hierarchical visibility algorithm, typically only a small fraction of the cubes in the octree are visible. When we render the next frame, most of the cubes that were visible in the last frame will probably still be visible. Some of the cubes visible in the last frame will become hidden and some cubes hidden in the last frame will become visible, but the frame-to-frame coherence that is typical of most animation ensures that there will be relatively few changes in cube visibility for most

[2] The znear pyramid can also establish that a primitive is entirely visible within a region of the screen, in which case, scan conversion can proceed without z comparisons. This occurs if the primitive's farthest z value within the region is nearer than the corresponding sample in the znear pyramid.

frames, except for scene changes and camera cuts. When a z-buffer hardware accelerator is available, we can exploit this fact in a simple way with the hierarchical visibility algorithm.

We associate a frame number with each octree cube, indicating the last frame that it was known to be visible. Frame numbers can be initialized to any number that is not a legal frame number. We render the first frame of an animation sequence with the usual algorithm, marking visible cubes with the number of the first frame. On all subsequent frames, we use the following two-pass algorithm to render a frame. The first pass begins by initializing the image and z-buffers. Then, we traverse all cubes in the octree that were visible in the preceding frame and render all of their primitives into a conventional z-buffer using the hardware accelerator. This procedure is very fast because we are simply traversing lists of primitives maintained in the octree and rendering them with the graphics accelerator. The final step of the first pass is to build a z-pyramid from the resulting z-buffer. At this point, the only thing missing from the image is geometry contained in cubes that have come into view since the last frame, and typically this is only a small fraction of all visible geometry. Likewise, the depth image stored in the z-pyramid is usually almost complete.

We render the "missing" geometry in the second pass, which is very similar to the original recursive subdivision procedure for rendering a frame. We recursively subdivide the octree and traverse cubes in front-to-back order, testing octree cubes for visibility, culling hidden cubes, and if the primitives associated with a visible cube have not already been rendered by the graphics accelerator, we render them into the z-pyramid with software scan conversion. Visible cubes are marked with the current frame number. When traversal of the octree finishes, we have a standard z-buffer image of the scene and all visible cubes have been marked with the current frame number. Typically, the second pass runs very rapidly because only a small amount of missing geometry needs to be rendered. Moreover, since the z-pyramid is usually almost complete, hierarchical image-space culling of hidden cubes often requires less subdivision than it normally would, resulting in better performance. When there is a high degree of frame-to-frame coherence, this algorithm for exploiting temporal coherence is much faster than the original algorithm because it renders nearly all primitives with the graphics accelerator rather than with software scan conversion, and because it permits more efficient hierarchical image-space culling with the z-pyramid.

One way of thinking about how this temporal-coherence procedure accelerates culling is that we begin by guessing the final solution. If our guess is very close to the actual solution, the hierarchical visibility algorithm can use the z-pyramid to verify the portions of the guess that are correct faster than it can construct them from scratch. Only the portions of the image that it cannot verify as being correct require further processing.

3.4 Building and Maintaining the Octree

Since the hierarchical visibility algorithm requires that scene primitives be organized into an octree, the cost of building and maintaining octrees is an important practical consideration. In this section we show that the cost of building an octree from n unorganized primitives is normally proportional to $n \log(n)$. By comparison, the cost of tiling a fixed-resolution image of a scene with naive z-buffering is linear in the number of primitives, since each primitive can be tiled in constant time. This disparity in complexity shows that an unorganized list of primitives is not a good representation for very complex models, and underscores the need to understand the asymptotic performance of the algorithms applied.

We begin our discussion of this topic by considering the cost of building an octree from an unorganized list of primitives using either variation of the algorithm presented in §3.3.1. To estimate asymptotic cost, it is necessary to make some assumptions. First, we assume that the average depth of a leaf node in the octree is $O(\log(n))$ and that, on average, the depth of insertion of a primitive is also $O(\log(n))$. We also assume that each primitive is inserted into no more than some constant number of nodes at each level. Given these assumptions, the cost of building the octree is $O(n \log(n))$, since each of the n primitives must be inserted into, on average, $O(\log(n))$ levels. Our assumption that the average depth of a leaf node is $O(\log(n))$ is valid, unless the octree is very poorly balanced, which is unlikely to occur unless the underlying geometry is distributed in a very unusual way. Average depth is lowest for a perfectly balanced octree. In this case, assuming that all n primitives are associated with leaf nodes and each leaf node has m primitives, the tree's depth is $\log_8(n) - \log_8(m) + 1$. Although average depth is higher for octrees that are not so well balanced, as Aho et al. show for binary trees [Aho-et-al83], even octrees with "random" branching patterns have $O(\log(n))$ average depth. In the worst case, the octree's branching structure is "linear," average depth is $O(n)$, and the cost of building the octree is $O(n^2)$.

When the scene model is static, the cost of building the octree is usually not an issue, because it may be considered a precomputation expense which can be amortized over all of the frames. For most interactive applications, the primary concern is maintaining rapid frame updates, and it may be acceptable to pay a high precomputation cost. The same considerations apply if the scene is predominantly static, with only a small number of moving objects. In this case, we can precompute an octree for the static components, render this part of the scene with the usual recursive subdivision procedure, and then complete the frame by tiling the moving objects into the z-buffer. The ability to render objects in any order is one of the advantages of z-buffering.

The cost of building and maintaining an octree is more problematic when animating

dynamic scenes having numerous moving primitives. Although the static component of the scene, if any, can be handled as described above with a *static octree*, this leaves numerous *dynamic primitives* that must be organized each frame. One strategy is to build a *dynamic octree* from the dynamic primitives at the beginning of each frame and, assuming that it is registered with the static octree, traverse the static and dynamic primitives simultaneously during the recursive rendering procedure. However, building the dynamic octree from d unorganized primitives normally requires $O(d \log(d))$ time and we would like to reduce this cost if possible. Fortunately, octree construction can often be accelerated by exploiting frame-to-frame coherence or by using lazy evaluation.

To exploit coherence in building the dynamic octree, we note that due to frame-to-frame coherence in the motion of primitives, it is usually faster to update the dynamic octree for the preceding frame than to build a new octree from scratch. Typically, a dynamic primitive moves only a short distance from one frame to the next, and in this case the primitive usually remains in the same octree node or moves to a nearby octree node. On average, such repositioning would be expected to require much less work than insertion into the $\log(d)$ levels of a new octree. While this approach can save considerable computation, the cost of updating an octree is at least linear in the number of dynamic primitives, since each primitive must be individually processed.

Now let's consider lazy evaluation as a strategy for reducing the cost of building an octree from unorganized primitives. With lazy evaluation, instead of building the complete octree and then traversing its visible nodes, we organize primitives within octree nodes as they are encountered during the recursive rendering procedure. We begin by associating all scene primitives with a root node having no children. During recursive subdivision, we determine that an octree node is visible before organizing its primitives into octants. This method avoids organizing primitives in hidden octree nodes, so in densely occluded scenes it can avoid most of the work required to build the fully subdivided octree. However, even with lazy evaluation the work required to build the octree is at least linear with respect to the number of primitives, since each primitive must be considered when the root node is processed, except in the trivial case that the root cube is hidden.

Thus far, the discussion of building and updating octrees has presumed that scenes are represented as an unorganized list of primitives. Consequently, the methods presented need to consider every dynamic primitive at every frame, and their performance is, at best, linear in the number of dynamic primitives. Fortunately, it is usually possible to obtain better performance, but this generally requires some higher-level structure in the scene model, some form of *coherence* that can be exploited. For example, some primitives could be organized in bounding volumes, the model could include instances of replicated geometry,

there could be limits on the range of motion of primitives or instances, and so forth. Let's consider each of these circumstances in turn, bearing in mind that the proposed methods may require modifying the basic rendering algorithm.

If the model includes bounding volumes, it's not necessary to insert primitives individually into the octree. Rather, the cluster of primitives bounded by a volume can be inserted all at once into the octree node that encloses the volume. If the model includes instances of a repeating module, the module can be organized into bounding volumes, and transformed instances of the module can then be organized within the octree by transforming and inserting their component bounding volumes. To exploit limited range of motion, if we know that an object moves a maximum of some fixed distance each frame, we can construct a bounding volume that is guaranteed to enclose the object over a sequence of k frames. It follows that the octree entry for this object only needs to be updated every k frames. In the case that the range of motion of a cluster of objects is limited to a particular bounding volume over all frames, the objects can be inserted into the static octree and never updated. This would be appropriate, for example, for an anchored, articulated object such as a robot arm.

In principle, it should be possible to perform space-time culling of hidden dynamic geometry by combining some of the methods outlined above. The central idea is to cull geometry within regions of space that are hidden over intervals of time. For example, if the motion of a cluster of primitives is predetermined, then for any sequence of frames we can construct a bounding volume for the primitives. Then, when processing any frame in the sequence, we can safely cull all primitives in the cluster if that volume is hidden (or if an octree node enclosing the volume is hidden). This approach can be applied hierarchically in both space and time. If the bounding volume is visible in a given frame, we can subdivide in either space or time as appropriate to construct a smaller bounding volume and then proceed as before. While the details remain to be worked out, this appears to be a promising approach to culling dynamic geometry.

In conclusion, a variety of strategies for exploiting coherence can accelerate building and maintaining octrees for dynamic scenes, and avoid the expense of considering every dynamic primitive at every frame of animation. The effectiveness of particular methods depends a great deal on geometric characteristics of the model and how the model is represented, so it is difficult to draw general conclusions. But regardless of the specific circumstances, it is often possible to find methods that effectively exploit coherence in order to avoid wasting time on geometry within hidden regions of space, whether or not that geometry is moving.

3.5 Implementation and Results

Our implementation of the hierarchical visibility algorithm uses the object-space octree, the image-space z-pyramid, and optionally, the temporal-coherence procedure. Software is written in C. To test the algorithm, we constructed a modular polygonal model of an office interior. The model is organized in cubic modules constructed from transformed instances of office cubicles and stairwells, each module consisting of approximately 15,000 polygons. To construct complex office interiors, instances of the module are replicated in a three-dimensional grid. The repeating module was designed to create environments having very complex occlusion relationships in which it is possible to see deep into the scene from most vantage points. This was accomplished by making support columns thin, limiting the height of cubicle walls, and including large open stairwells that make it possible to see parts of neighboring floors.

Given our simple replication scheme, it was possible to represent the model as an "octree of octrees" where the super-octree references translated instances of a conventional octree for the repeating module. Thus, it was only necessary to build one conventional octree from the 15,000 polygons in the repeating module. To build this octree, we used the method described in section §3.3.1 that avoids placing small polygons in large octree cubes. Assignment of polygons to octree cubes obeyed the following recursively applied rule: if the area of a polygon was smaller than one-tenth the area of one of a cube's faces, then the polygon was associated with the cube's children that it intersected. After building the complete octree, if there were fewer than a total of 20 polygons in any octree node and its children, we eliminated that node, associating its polygons with the parent node. In the final octree, between 20 and 66 polygons were associated with each leaf node, the average being 31.3, and each polygon was associated with an average of 1.9 octree nodes.

3.5.1 A Simple Scene

For simple models with low depth complexity, the hierarchical visibility method can be expected to take somewhat longer than traditional scan conversion due to the overhead of performing visibility tests on octree cubes and the cost of maintaining a z-pyramid. To measure the algorithm's overhead on simple models, we rendered a single module consisting of approximately 15,000 polygons at a viewpoint from which a high proportion of the model was visible. On a 50Mhz R4000 SGI Crimson, rendering time for a 512×512 image was 1.52 seconds with the hierarchical visibility method and 1.30 seconds with traditional scan conversion, indicating a performance penalty of 17%. When we rendered three instances

of the module (45,000 polygons), the running time was 3.05 seconds for both methods, indicating that this level of complexity was the break-even point for this particular model.

3.5.2 A Complex Scene

The chief value of the hierarchical visibility algorithm is, of course, for scenes of much higher depth complexity. To illustrate the point, we constructed a 33×33×33 replication of the office-interior module, producing a model having approximately 538 million polygons. Figure 3.6 shows two consecutive frames of animation of the office environment. The frame on the right was rendered with the temporal-coherence procedure (shading anomalies are explained below). In figure 3.7, we show a top view of the viewing frustum and the octree subdivision. 59.7 million polygons lie inside the viewing frustum, about one-tenth of the entire model. The cubes shown in wireframe in figure 3.7 are the octree nodes visited during rendering, that is, the visible octree nodes and their hidden children. Note that the algorithm is able to prove that many large octree nodes in the background are hidden. The z-pyramid for the scene is shown in the left panel of figure 3.4. Even at fairly coarse resolutions, the z-pyramid contains a recognizable representation of the major occluders in the scene.

During recursive subdivision of the octree, our implementation of the hierarchical visibility algorithm culls hidden cubes using the fast z-pyramid test described in §3.3.2, which determines whether a cube face's screen-space bounding rectangle is visible at the depth of its nearest vertex. If this test fails to show that a cube's faces are all hidden, we assume that the cube is visible without performing a definitive visibility test by tiling the cubes's front faces. We found that performing definitive tests took more time than it saved. For the scene of figure 3.6 (left panel), the z-pyramid test was invoked on 5910 octree cubes and succeeded in culling 3450 of them. The remaining 2460 potentially visible cubes contained approximately 47,900 polygons. It follows that the z-pyramid test on octree cubes culled .9992 of all polygons inside the viewing frustum, leaving only .0008 of the polygons to be rendered. Of the 47,900 polygons in potentially visible cubes, approximately 19,300 were front facing. Each front-facing polygon was tested for visibility with the fast z-pyramid test, and if it failed to prove that the polygon was hidden, the polygon was tiled pixel by pixel into the z-pyramid using software scan conversion.

Rendering this frame at 512×512 resolution using only software scan conversion took 4.88 seconds on a 50Mhz R4000 SGI Crimson. Of the 4.88-second frame time, 15% was consumed by z-pyramid tests on octree cubes and polygons (.73 seconds), 82% was consumed by software scan conversion of the polygons inside potentially visible octree cubes (4.02

Figure 3.6 Two consecutive frames from animation of an office environment rendered with the hierarchical visibility algorithm. The model is organized as an octree of octrees and contains 538 million replicated polygons. The frame on the left was rendered at 512×512 resolution with all-software scan conversion in 4.88 seconds on a 50Mhz R4000 SGI Crimson. The frame on the right was rendered with the temporal-coherence procedure, which begins by using the Crimson's hardware accelerator to render all primitives inside octree cubes that were visible in the preceding frame. The polygons rendered with hardware acceleration are Gouraud shaded. The remaining polygons, shown in magenta to distinguish them, were rendered in a second pass using software scan conversion. Using this temporal-coherence procedure, this 512×512 frame took 2.02 seconds to render on the Crimson, which has a VGX hardware accelerator. The high degree of frame-to-frame coherence of visible octree cubes that is apparent in this example is typical of walk-through animation of this environment.

seconds),[3] which left 3% (.13 seconds) for all other operations – clearing the image buffer and z-pyramid, testing octree cubes for intersection with the viewing frustum, etc. Note that the hierarchical visibility algorithm spent the vast proportion of its time (82%) tiling visible or nearly visible polygons, indicating that the efficiency of traversing the octree and culling hidden octree cubes and polygons with the z-pyramid was very high.

To compare the performance of the hierarchical visibility algorithm with naive z-buffering, we produced the identical z-buffer image by performing software scan conversion of all 59.7 million polygons lying inside the viewing frustum. This process took 33.6 minutes on our 50Mhz Crimson, 413 times longer than the hierarchical visibility algorithm.

[3] It should be noted that our unoptimized scan conversion code only renders about 5,000 polygons a second. A careful implementation would probably run several times faster.

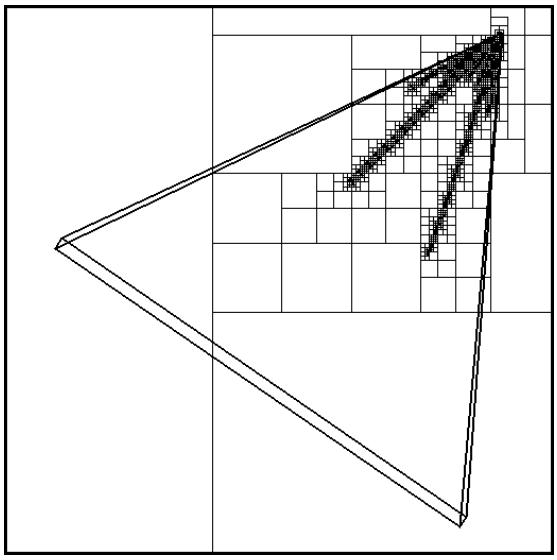


Figure 3.7 Top view of viewing frustum and octree cubes visited while rendering the office model of figure 3.6.

3.5.3 Depth Complexity of Tiling Operations

As the preceding example shows, the hierarchical visibility algorithm spends nearly all of its time performing tiling – hierarchically tiling cubes faces to determine whether cubes are visible and tiling the model primitives inside visible or nearly visible cubes. Consequently, we can visualize where the algorithm is spending its time by constructing images that indicate the number of times that each pixel is traversed during tiling operations. We will say that such images depict the "depth complexity" of tiling operations. Constructing depth-complexity images for conventional z-buffer scan conversion is very straightforward: we simply count the number of times each pixel is traversed in the course of tiling and encode these numbers in an image. We can also depict how much work is done during hierarchical culling with the z-pyramid by keeping track of the number of times each z-pyramid sample is accessed and amortizing each access over the corresponding square region of the screen. Specifically, when a pyramid sample representing an $n \times n$ block of pixels is accessed, we add $1/n^2$ to the value for depth complexity at each pixel in that region of the screen.

Figure 3.8 shows depth-complexity images of the complete scene model and various tiling operations performed by the hierarchical visibility algorithm. These are log-scale images in which intensity encodes the log of the number of times each pixel is traversed. The upper-left panel is a z-buffer image of the scene, also shown in figure 3.6. The upper-right panel shows the depth complexity of the geometry in the entire scene, which is 83.7, meaning that on average, 83.7 polygons overlap at each pixel. In other words, if we were to render

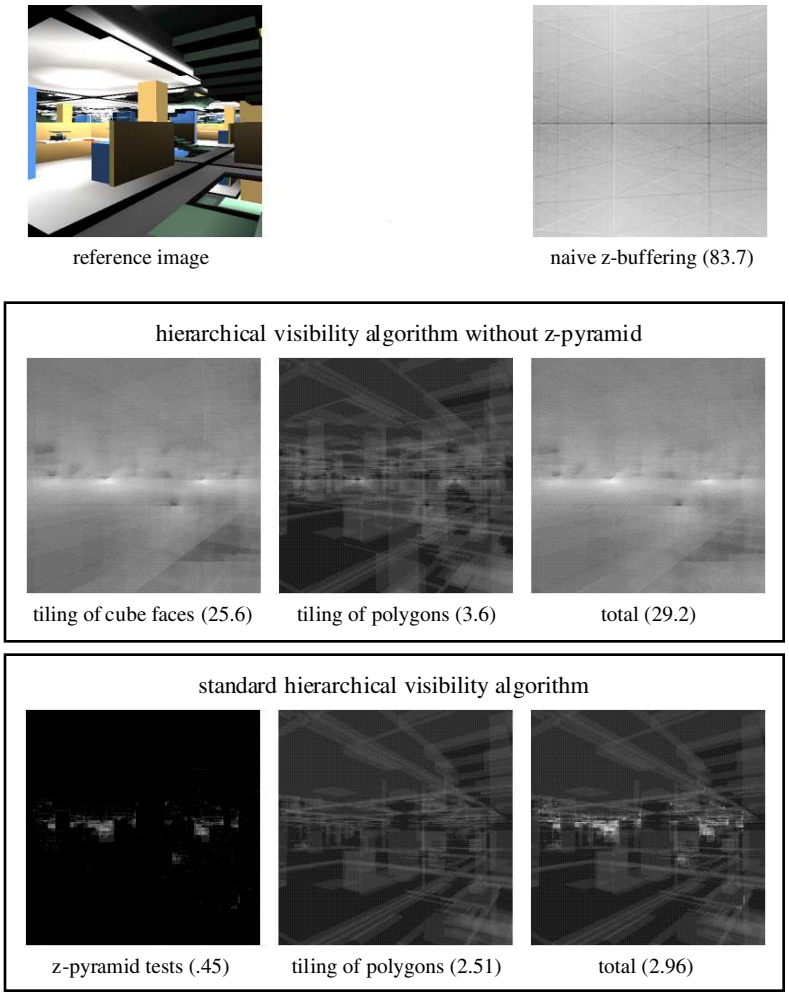


Figure 3.8 Log-scale depth-complexity images that help to visualize where various tiling procedures are spending their time. The right column of panels shows the total depth complexity of tiling for:

top: naive z-buffering (83.7)

middle: the hierarchical visibility algorithm without z-pyramid culling (29.2)

bottom: the standard hierarchical visibility algorithm (2.96).

---

the scene by casting a single ray through each pixel center, the average number of polygons intersected by each ray would be 83.7. As previously mentioned, we obtained this image by z-buffering the 59.7 million polygons that lie inside the viewing frustum, which took 33.6 minutes.

The bottom tier of panels in figure 3.8 shows the depth complexity of tiling operations performed by the hierarchical visibility algorithm in the course of producing the same z-buffer image. The bottom-left panel shows the depth complexity of z-pyramid tests on cube

faces and polygons (.45), encoded as described above, the bottom-center panel shows the depth complexity of tiling polygons (2.51), and the bottom-right panel shows the sum of these two images. Total average depth complexity is 2.96. As previously mentioned, our implementation uses the fast z-pyramid test described in §3.3.2 to cull hidden cubes and polygons. The low average depth complexity of .45 for these operations (.20 for cubes, .25 for polygons) is an indication of how efficiently the z-pyramid performs hierarchical culling. The bottom-center panel shows the depth complexity of scan converting the polygons inside potentially visible cubes. The bottom-right panel shows the sum of the z-pyramid and polygon-tiling depth-complexity images, and therefore indicates the total depth complexity of all tiling operations performed by the hierarchical visibility algorithm in rendering this scene, which is 2.96. As previously mentioned, the depth complexity of naive z-buffering is 83.7, higher by a factor of 28.3.

We also created depth-complexity images for running the hierarchical visibility algorithm without the z-pyramid in order to estimate the value of hierarchical culling in image space. The results are shown in the middle tier of panels in figure 3.8. When a z-pyramid is not available, visibility of cube faces and polygons must be determined by conventional pixel-by-pixel scan conversion. The middle-left panel shows the depth complexity of tiling cube faces (25.6), the middle-center panel shows the depth complexity of tiling the polygons that are inside visible cubes (3.6), and the middle-right panel shows the sum of these two images. Thus, without the hierarchical image-space culling that is enabled by a z-pyramid, the depth complexity of all tiling operations for this scene is 29.2, nearly ten times higher than when we use a z-pyramid.

Summing up, we have compared the depth complexity of three different tiling methods for producing a standard z-buffer image of a densely occluded scene having complex occlusion relationships. The depth complexities of tiling operations for hierarchical visibility, hierarchical visibility without hierarchical image-space culling, and naive z-buffering are 2.96, 29.2, and 83.7, respectively.

The office environment of figure 3.6 was chosen in part because of the difficulties it presents potentially visible set methods [Airey90, Teller-Sequin91, Teller92], which we reviewed in §2.1.6. Recall that these methods partition model space into disjoint regions and establish the "potentially visible set" of primitives that are visible from each region. Then, only primitives in the potentially visible set for the region that contains the current viewpoint need to be rendered for a given frame. While potentially visible set methods often work well for architectural models, they would not work effectively with our model because within every office cubicle there are viewpoints from which almost every other cubicle on the same floor is visible. As a result, if the cubicles were used as cells, the potentially visible

Figure 3.9 Terrain models rendered with the hierarchical visibility algorithm.

set for each cell would have to include nearly all the cells on its floor and many on other floors. Since each floor contains about four million polygons, potentially visible set methods would probably have to render many more polygons than the hierarchical visibility method.

3.5.4 An Outdoor Scene

Figure 3.9 shows the hierarchical visibility method applied to outdoor scenes consisting of a terrain mesh with vegetation replicated on a two-dimensional grid. The model shown in the left panel consists of approximately 53 million polygons, but only about 25,000 polygons are visible from this point of view. Most of the model is hidden by the hill or is outside the viewing frustum. This frame took approximately 7 seconds to render with software scan conversion on our 50Mhz R4000 SGI Crimson. On the right, we show a model consisting of approximately 5 million polygons. Even though this model has fewer primitives, the image took longer to render because a much larger fraction of the model is visible from this point of view. This image took approximately 40 seconds to render with software scan conversion on the Crimson. These outdoor scenes have very different characteristics from the building interiors shown in figure 3.6 and are poorly suited to potentially visible set methods because cell-to-cell visibility is not nearly as limited as in an architectural interior. Nonetheless, the hierarchical visibility algorithm continues to work effectively.

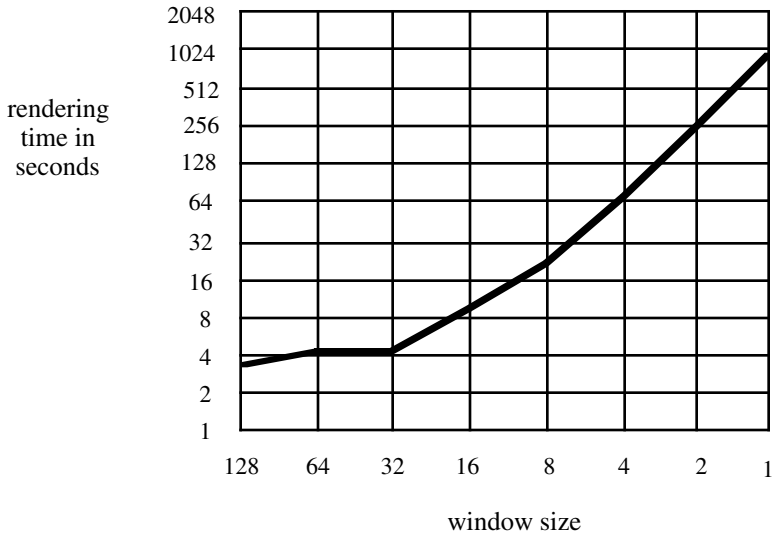


Figure 3.10 Total time in seconds to render all windows of a frame versus window size expressed as the number of pixels on the side of each window.

3.5.5 Parallel Performance

We have made our hierarchical visibility implementation capable of dividing the image into a grid of smaller windows, rendering them individually, and combining them into a final image. The performance of the algorithm as the window size is varied tells us about the parallel performance of the algorithm and the extent to which it is able to exploit image-space coherence. If, like most ray tracers, the algorithm made no use of image-space coherence, we could render each pixel separately at no extra cost. Then it would be fully parallelizable. At the other extreme, if the algorithm made the best possible use of image-space coherence, it would render a sizeable region of pixels with only slightly more computation than required to render a single pixel. Then it would be difficult to parallelize. Note that if we shrink the window size down to a single pixel, the hierarchical visibility algorithm resembles a ray caster using an octree subdivision.

Figure 3.10 graphs the rendering time for a frame from a walk-through of the model shown in figure 3.6 as a function of the window size. For window sizes of 32×32 and larger, the curve is relatively flat, indicating that the algorithm should parallelize fairly well. For window sizes smaller than 32×32, however, the slope of the curve indicates that the time required to render a window is almost independent of its size. For example, note that it only takes about four times longer to render a 32×32 region as it does to ray-cast a single pixel with this algorithm.

3.5.6 Use of Graphics Hardware

In addition to the pure software implementation, we have attempted to modify the algorithm to make effective use of available commercial hardware graphics accelerators. This raises some difficult challenges because the hierarchical visibility algorithm makes somewhat different demands of scan-conversion hardware than traditional z-buffering. In particular, our octree culling procedure depends on being able to determine quickly whether a polygon would be visible if it were scan converted. Unfortunately, the commercial hardware graphics pipelines we have examined are either unable to answer this query at all, or can take milliseconds to answer it. One would certainly expect some delay in getting information back from a graphics pipeline, but hardware designed with this type of query in mind should be able to return a result in microseconds rather than milliseconds.

We have implemented a modified version of the hierarchical visibility algorithm on a Kubota Pacific Titan 3000 workstation with Denali GB graphics hardware. The Denali hardware supports an unusual graphics library call that determines whether or not any pixels in a set of polygons are visible given the current z-buffer. We use this *z-query* feature to determine the visibility of octree cubes. The cost of a z-query depends on the screen size of the cube, and it can take up to several milliseconds to determine whether or not a cube is visible. Our implementation does not use a z-pyramid because it is not supported by the Denali hardware. During walk-through animation of a version of the office environment with 1.9 million polygons, the Titan took an average of .54 seconds per frame to render 512×512 images. Because of the cost of doing the z-query, we only tested visibility of octree cubes containing at least 800 polygons. Even so, 36.5% of the running time was taken up by z-queries. If z-query were faster, we could use it effectively on octree cubes containing many fewer polygons and achieve substantial further acceleration. The Titan implementation has not been fully optimized for the Denali hardware and makes no use of temporal coherence, so these performance figures should be considered only suggestive of the machine's capabilities.

The other implementation we have that makes use of graphics hardware runs on SGI workstations. On these workstations there is no way to inquire whether or not a polygon is visible without actually rendering it, so we use the hybrid hardware/software strategy described in §3.3.3. Reiterating this discussion, we render the first frame of an animation sequence entirely with software scan conversion. Starting with the second frame, we use the hardware pipeline to render the polygons contained in octree nodes that were visible in the preceding frame. Then we read the image and the z-buffer from the hardware, build a z-pyramid, and continue with the second pass of the temporal-coherence procedure, filling

in geometry that has come into view since the last frame with software scan conversion. With this implementation, temporal coherence typically reduces frame time by a factor of approximately two and one-half for animation of the office environment.

In the course of walk-through animation of the office environment, we rendered the frame in the left panel of figure 3.6 without the temporal-coherence procedure, and then rendered the next frame, shown in the right panel, with it. In the temporal-coherence frame, polygons rendered with the hardware pipeline are Gouraud shaded and the remaining polygons rendered with software scan conversion are shown in magenta to distinguish them. For the most part, these are polygons that came into view as a result of panning the camera. These magenta polygons are the only geometry visible in this frame that is not associated with an octree node that was visible in the preceding frame. This high degree of frame-to-frame coherence of visible octree nodes is typical of walking through this environment.

Current graphics accelerators are not designed to support the rapid feedback from the pipeline needed to realize the full potential of octree culling in the hierarchical visibility algorithm. Hardware designed to take full advantage of the algorithm, however, could make it possible to interact very effectively with extremely complex environments as long as only a manageable number of the polygons are visible from any point of view. The octree subdivision, the z-pyramid, and the temporal-coherence procedure are all suitable for hardware implementation.

3.6 Conclusion

Underlying our basic algorithm is a very simple idea: we organize scene geometry in bounding boxes, and before doing any work on the geometry inside a box, we first verify that the box itself is visible by tiling its faces.[4] In the context of z-buffering, visibility of boxes can be established by tiling their faces with ordinary scan conversion, but as we have seen, hierarchical tiling enabled by a z-pyramid permits box-visibility tests to be performed much more efficiently. And if bounding boxes are arranged hierarchically, as they are in an octree, culling of hidden geometry can be performed hierarchically in both object space and image space. Conversely, we find visible geometry by logarithmic search in both object space and image space. This simple strategy encapsulates our basic algorithm. As previously noted, Meagher has applied a similar approach to volume rendering [Meagher82b].

---

[4] No doubt this idea has occurred to many practitioners, but I first heard it articulated by Lance Williams in the early 1980's when we worked together at the NYIT Computer Graphics Lab. When I moved to Apple in 1989, I was reminded of the strategy by Gavin Miller, who had used this method to accelerate z-buffering.

From the standpoint of coherence, the algorithm exploits object-space coherence as effectively as ray casting through a spatial subdivision and it exploits image-space coherence even more effectively than traditional incremental scan conversion. For animation sequences it is also able to exploit frame-to-frame coherence. In fact, the hierarchical visibility algorithm appears to be the first practical visibility algorithm which materially profits from object-space, image-space, and temporal coherence simultaneously. The algorithm has been tested and shown to work effectively on complex, densely occluded indoor and outdoor scenes with up to half a billion polygons. While the algorithm can make use of existing graphics accelerators without modification, small changes in the design of graphics accelerators would dramatically improve the performance of the algorithm. We hope that the appeal of this algorithm will induce hardware designers to adapt future graphics hardware to facilitate hierarchical visibility computations.

Chapter 4

Hierarchical Polygon Tiling with Coverage Masks

4.1 Overview

While the hierarchical z-buffer visibility algorithm has excellent performance, the images that it produces suffer from the usual aliasing problems caused by point-sampling algorithms – jaggies and other disturbing visual artifacts. As discussed in §2.2.2, the most popular approach to antialiasing is oversampling and filtering. Applying this approach with the hierarchical z-buffer algorithm is very straightforward. Tiling of primitives and visibility testing of octree cubes is simply performed on a higher-resolution grid, which entails maintaining a higher-resolution image buffer and z-pyramid. After the scene has been tiled, each pixel in the output image is obtained by filtering a neighborhood of the higher-resolution image. While this approach is reasonably efficient, the cost of tiling and the size of image memory increase in approximate proportion to the degree of oversampling, so image generation is many times slower than with the corresponding point-sampling algorithm. Consequently, we explored alternative methods of performing tiling within the framework of the basic hierarchical visibility algorithm. In doing so, we discovered a novel polygon tiling algorithm which we call *hierarchical polygon tiling* that is capable of producing antialiased images much more efficiently than hierarchical z-buffering, while requiring much less image memory.

Hierarchical tiling traverses convex polygons in front-to-back order, tiling them into an image pyramid, using specially constructed masks that we call *triage coverage masks* to accelerate tiling. Triage masks are tiling patterns represented by a pair of bit masks that facilitate classifying a convex polygon as *inside*, *outside*, or *intersecting* cells in the image pyramid. They permit Warnock-style subdivision of image space with its ability to find visible geometry by logarithmic search to be driven very efficiently by boolean mask operations. The resulting tiling procedure performs subdivision and visibility operations very rapidly while only visiting cells in the image hierarchy that are crossed or nearly crossed by visible edges in the output image. Visible samples are never overwritten. Hierarchical z-buffering, on the other hand, must visit every image sample covered by every visible polygon,[1] and it frequently overwrites image samples. Moreover, when antialiasing

[1] Recall that the hierarchical z-buffer algorithm reverts to traditional z-buffer scan conversion in tiling visible

is performed by oversampling, hierarchical tiling requires only a small fraction of the image memory required by hierarchical z-buffering. As a consequence of these advantages in tiling efficiency and memory usage, hierarchical tiling can generate high-quality antialiased images at approximately the same speed as hierarchical z-buffering generates the corresponding point-sampled images. The main drawback of hierarchical tiling as compared with hierarchical z-buffering is that polygons must be traversed in strict front-to-back order, which for dynamic scenes necessitates additional sorting computations. This is usually not a serious problem for predominantly static scenes, since they can be conveniently maintained in a BSP tree [Fuchs-Kedem-Naylor80, Naylor92a], but it can severely impair performance if a great many polygons are moving independently.

We begin our presentation of the hierarchical polygon tiling algorithm by placing it in the general context of polygon tiling algorithms, and then describe how it can be integrated with the basic hierarchical visibility algorithm.

4.2 Introduction

The purpose of a polygon tiling algorithm is to determine which point samples on an image raster are covered by the visible portions of each of the polygons composing a scene. This has been an important topic in computer image synthesis since the advent of raster graphics some two decades ago. Currently, polygon tiling software running on inexpensive computers can render point-sampled images of simple scenes at interactive rates. The fastest tiling algorithms have been carefully tuned to exploit image-space coherence by using incremental methods wherever possible. However, they fail to exploit opportunities for precomputation and they waste time tiling hidden geometry. There is a need for more efficient tiling algorithms that effectively exploit precomputation and coherence to enable efficient culling of hidden geometry and efficient tiling of visible geometry.

The dominant polygon tiling algorithm in use today is incremental scan conversion. Typically, raster samples on a polygon's perimeter are traversed with an incremental line-tiling algorithm such as the Bresenham algorithm [Bresenham65]. Edge samples on each intersected scanline define spans within a polygon, which are then crossed pixel-by-pixel, permitting incremental update of shading parameters and, in the case of z-buffering, depth values. Visibility of samples can be determined by a) maintaining a z-buffer and performing depth comparisons [Catmull74], b) traversing primitives back to front and writing every pixel tiled [Foley-et-al90], or c) traversing primitives front to back and overwriting only

---

polygons.

vacant pixels [Foley-et-al90]. With incremental scan conversion, the cost per pixel tiled is very low because incremental edge and span traversal effectively exploits image-space coherence. Overall, the cost of tiling a scene is roughly proportional to its depth complexity, so traditional incremental scan conversion is an efficient way to tile shallowly occluded scenes.

One problem with traditional incremental scan conversion is that it must tile every sample on every primitive, whether or not it is visible, and so it wastes time tiling hidden geometry. This is not a big problem for simple scenes, but for densely occluded scenes it severely impairs efficiency. Ideally, a tiling algorithm should cull hidden geometry efficiently so that running time is proportional to the visible complexity of the scene and independent of the complexity of hidden geometry.

As discussed in §2.1.3, the Warnock subdivision algorithm [Warnock69] approaches this goal, performing logarithmic search for visible tiles in the quadtree subdivision of a polygon. If scene primitives are processed front to back, only visible tiles and their children in the quadtree are visited. Although Warnock subdivision satisfies our desire to work only on visible (or nearly visible) regions of primitives, the traditional subdivision procedure is relatively costly and consequently, this approach is slower than incremental scan conversion except for densely occluded scenes. Neither traditional incremental scan conversion nor Warnock subdivision is well suited to tiling scenes of moderate depth complexity.

A second shortcoming of incremental scan conversion is that it spends most of its time tiling edges and spans, traversing these features pixel by pixel, even though all possible tiling patterns for an edge crossing a block of samples can be precomputed and stored as bit masks called *coverage masks*. Then the samples that a convex polygon covers within a block can be quickly found by compositing the coverage masks of its edges. Previously, this technique has been applied to estimating coverage of polygonal fragments within a pixel to accelerate filtering [Carpenter84, Sabella-Wozny83, Fiume-et-al83, Fiume91], as discussed in §2.2.2.

Here we present a polygon tiling algorithm that combines the best features of traditional algorithms. The key innovation that makes this integration possible is the generalization of coverage masks to permit their application to image hierarchies. The generalized masks, which we call *triage coverage masks*, classify cells in the hierarchy as *inside*, *outside*, or *intersecting* an edge. This permits them to drive Warnock-style subdivision. The result is a hierarchical tiling algorithm that finds visible geometry by logarithmic search, as with the Warnock algorithm, that exploits precomputation of tiling patterns, as with filtering with coverage masks, and that also uses incremental methods to exploit image-space coherence, as with incremental scan conversion. The algorithm is well suited to high-resolution tiling

(e.g. 4096×4096), so it naturally supports high-quality antialiasing by oversampling. A-buffer-style antialiasing with coverage masks [Carpenter84] is particularly convenient.

For densely occluded scenes we combine hierarchical tiling with the basic hierarchical visibility algorithm to permit hierarchical culling of hidden regions of object space. This combination of algorithms enables very rapid rendering of complex polygonal scenes with high-quality antialiasing. This method has been tested and shown to work effectively on densely occluded scenes. On a test scene having upwards of 500 million replicated polygons, the algorithm computed visibility on a 4096×4096 grid at roughly the same speed as the hierarchical z-buffer visibility algorithm of chapter 3 tiled a 512×512 grid.

In §4.3, we survey the most relevant previous work on efficient polygon tiling. In §4.4, we introduce triage coverage masks, and in §4.5 we present the hierarchical tiling algorithm in which they are applied. In §4.6, we discuss how the algorithm can be accelerated with object-space culling methods. In §4.7, we describe the implementation and show results for a densely occluded scene. In §4.8, we compare the hierarchical tiling algorithm to the hierarchical z-buffering algorithm of chapter 3. Finally, we present our conclusions in §4.9.

4.3 Previous Work

Previous approaches to accelerating polygon tiling have included exploiting image-space coherence with incremental methods, hierarchical culling with Warnock subdivision, and precomputation of tiling patterns with coverage masks. However, no previous algorithm has combined all of these features.

4.3.1 Warnock Subdivision

Incremental methods that exploit image-space coherence in traversing edges and spans are summarized in the introduction, so we begin our discussion of previous work with the Warnock algorithm [Warnock69]. Reiterating the discussion from §2.1.3, the Warnock algorithm is a recursive subdivision procedure that finds the quadtree subdivision of visible edges in a scene by logarithmic search. Scene primitives are inserted into a quadtree data structure beginning at the root cell. At each level of subdivision, the algorithm classifies the quadrants of the current quadtree cell as inside, outside, or intersecting a primitive, and only intersected quadrants are subdivided. Quadrants which are entirely covered by primitives are identified, permitting hidden geometry within them to be culled. The Warnock algorithm is actually a family of algorithms based on a common subdivision procedure, and the control structure varies from implementation to implementation [Rogers85]. A typical

implementation processes primitives in no particular order, maintains lists of potentially visible primitives at quadtree cells, and expends considerable work performing depth comparisons in order to cull hidden geometry.

When circumstances permit convenient front-to-back traversal of primitives, as with presorted static polygonal scenes, a simpler and more efficient variation of the Warnock algorithm can be employed. In this case, we insert primitives into the quadtree one at a time in front-to-back order. As subdivision proceeds, we mark cells that one or more primitives completely covers as *occupied* and ignore cells that are already occupied, since any geometry that projects to them is known to be hidden. We complete subdivision of one primitive down to the finest level of the quadtree before processing the next. This version of the Warnock algorithm is simpler because it need not maintain lists of primitives or perform depth comparisons. It is more efficient because, unlike the traditional algorithm, it only subdivides cells crossed by edges that are visible in the output image. To the best of our knowledge, this is a novel variation of the Warnock algorithm, although we haven't researched the question in depth. In any case, our tiling algorithm is based on this variation, which we will refer to as the *depth-priority Warnock algorithm.* It should be noted that Meagher's volume rendering algorithm tiles the faces of the cubes of an octree using this procedure [Meagher81].

4.3.2 Coverage Masks

We turn now to reviewing how filtering algorithms exploit precomputation with coverage masks [Carpenter84, Sabella-Wozny83, Fiume-et-al83, Fiume91], which has already been discussed in §2.2.2. The underlying idea is that all possible tiling patterns for a single edge crossing a grid of raster samples within a pixel can be precomputed and later retrieved, indexed by the points where the edge intersects the pixel's border [Fiume-et-al83, Sabella-Wozny83]. These tiling patterns can be stored as bit masks, permitting samples inside a convex polygon to be determined by ANDING together the coverage masks for its edges. Moreover, if polygons are processed front to back or back to front, visible-surface determination within a pixel can also be performed with boolean mask operations. For example, Carpenter's A-buffer algorithm [Carpenter84] clips polygons to pixel borders, sorts the polygonal fragments front to back, and determines the visible samples of each fragment on a 4×8 grid by compositing coverage masks. For each visible fragment, a single shading value is computed, weighted by the bit count of its mask, and added to pixel color. This method is a very efficient way of approximating *area sampling* [Sutherland-et-al74, Catmull78] and effectively antialiases edges. Abram, Westover, and Whitted advance sim-

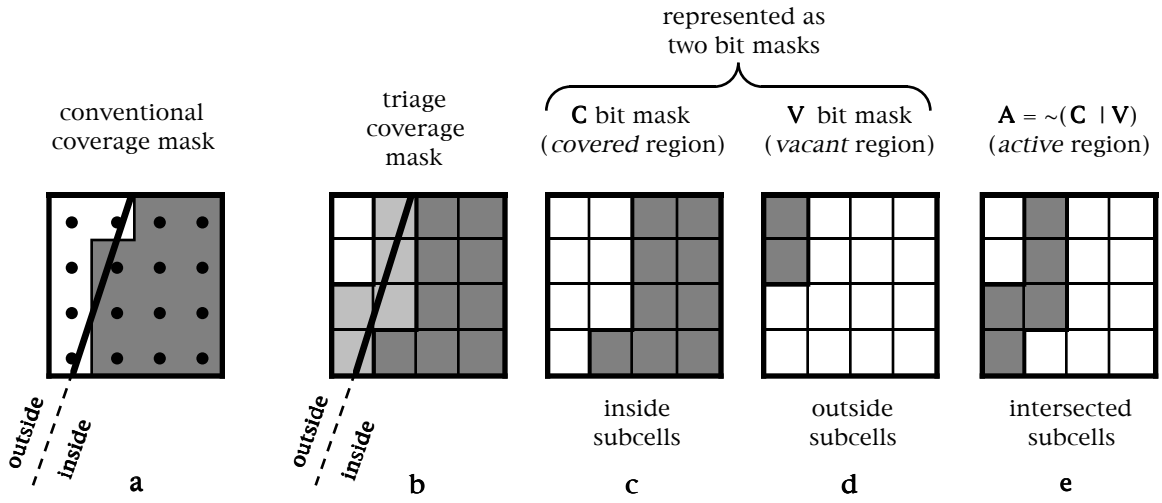


Figure 4.1 A conventional coverage mask classifies grid points as inside or outside an edge (panel a). A *triage* coverage mask classifies subcells as inside, outside, or intersecting an edge (panel b). We refer to these regions as *covered* (panel c), *vacant* (panel d), and *active* (panel e), respectively. We represent triage masks as the pair of bit masks (C,V) indicating the covered and vacant regions. In our illustrations of triage masks (e.g. panel b), the C bit mask corresponds to dark gray subcells and the V bit mask corresponds to white subcells. In practice, we use 8×8 masks rather than 4×4 masks.

ilar methods that permit convolution with arbitrary filter kernels, jitter, and evaluation of simple shading functions by table lookup [Abram-et-al85].

4.4 Triage Coverage Masks

To accelerate polygon tiling, the hierarchical tiling algorithm generalizes coverage masks to operate on image hierarchies, thereby enabling Warnock-style subdivision of image space to be driven by boolean mask operations. We precede the discussion of the tiling algorithm by explaining how these *triage* masks work.

4.4.1 Triage Edge Masks

A conventional coverage mask for an edge classifies each grid point within a square as inside or outside the edge, as shown in figure 4.1a. In the context of Warnock subdivision, the analogous operation is classifying subcells of an image hierarchy as inside, outside, or intersecting an edge, as shown in figure 4.1b for an edge crossing a square containing a 4×4

grid of subcells. We call such masks *triage*[2] *coverage masks* because the three states that they distinguish correspond to trivial rejection, trivial acceptance, and "do further work." We represent each triage mask as a pair of bit masks, one indicating inside subcells, the other indicating outside subcells, as shown in figures 4.1c and 4.1d. We will refer to the bit mask for inside subcells as the "C" mask (for *covered*) and the bit mask for outside subcells as the "V" mask (for *vacant*). We call the intersected subcells the *active* region of the mask, because this is the region that requires further work and will later be subdivided. The bit mask for the active region is $A = \tilde{}(C \mid V)$, as shown in figure 4.1e.[3] Hereafter, we will often refer to triage coverage masks simply as "masks" and we will refer to conventional one-bit masks as "bit masks" or "conventional coverage masks." In practice, we use 8×8 masks rather than the illustrated 4×4 masks.

4.4.2 Compositing Triage Masks

The representation of masks as (C,V) pairs was chosen to facilitate the basic tiling and visibility operations: finding the mask of a convex polygon from the masks of its edges, and compositing a polygon mask together with a mask representing a square region of the screen. Both of these operations can be performed with simple boolean mask operations.

Polygon Masks from Edge Masks

We make a mask for convex polygon P within a square region S of the screen as follows. Assume that we have already created a lookup table for edge masks that is indexed by edge-square intersection points, as we will describe later in §4.4.3. For each edge of P that intersects S, we find its two intersection points on S's perimeter and look up the precomputed edge mask. Suppose that P has N edges which intersect S, and that their edge masks are $(E1_C, E1_V), (E2_C, E2_V), ..., (EN_C, EN_V)$. Then P's mask is simply (P_C, P_V), where $P_C = E1_C \;\&\; E2_C \;\&\; ... \;\&\; EN_C$ and $P_V = E1_V \mid E2_V \mid ... \mid EN_V$. For example, figure 4.2a shows a polygon P having two edges, $E1$ and $E2$, that intersect square S. We ignore the edges of P that don't intersect S's perimeter. Combining the edge masks for $E1$ (figure 4.2b) and $E2$ (figure 4.2c) as just described produces P's mask (figure 4.2d).

---

[2] Pronounced *TREE-ahj*. To allocate scarce resources to best effect, the triage system of disaster management assigns survivors to three categories. Resources are reserved for those who require treatment to survive and withheld from those who would survive in any case and those who would perish in any case.

[3] We use standard notation for boolean mask operations: & for bitwise AND, | for bitwise OR, and ~ for bitwise complement.

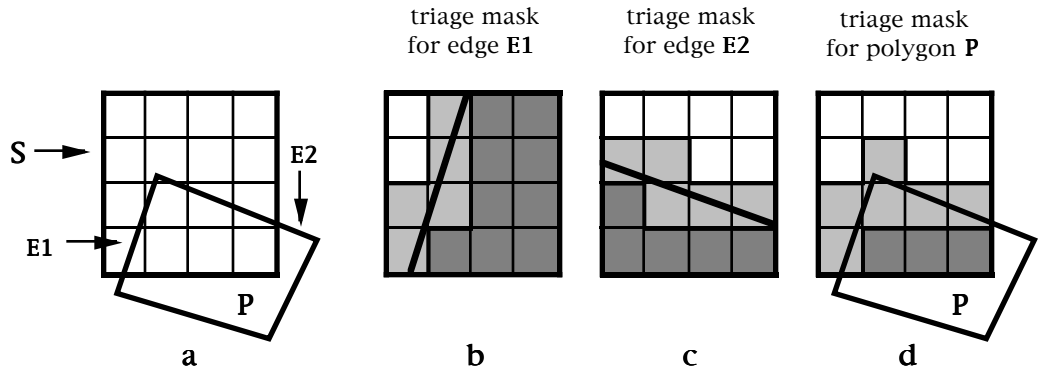


Figure 4.2 An example illustrating how the triage mask of a convex polygon is constructed from the triage masks of its edges that intersect the mask. The polygon's C bit mask is obtained by ANDing together the edges' C bit masks, and the polygon's V bit mask is obtained by ORing together the edges' V bit masks. In this example, the mask of polygon P (panel d) is made from the masks of edges $E1$ (panel b) and $E2$ (panel c) as follows: $P_C = E1_C \;\&\; E2_C$, $P_V = E1_V \mid E2_V$. As in the other illustrations of triage masks, C bit masks correspond to dark gray subcells and V bit masks correspond to white subcells.

Compositing Rules

Now for compositing masks within a square. Let (S_C, S_V) be the mask for a square region S of the screen that is initially vacant with S_C=*all_zeros*, S_V=*all_ones*. We process polygons in front-to-back order, compositing their masks with S's mask. For each polygon, our task is to identify the subcells that it covers and the subcells in which it is active, and then update (S_C, S_V) as necessary. In analyzing this procedure, we first consider the rules for compositing on a subcell-by-subcell basis, even though actual compositing is done with boolean mask operations.

Let s be any subcell of S and let p be the corresponding subcell of the mask for polygon P. Since each mask distinguishes three states, there are nine possible compositing cases which are summarized in table 4.1. With cases 1, 2, and 3, s is already *covered*, so P is not visible and can be ignored. With cases 6 and 9, p is *vacant*, so P is not visible and can be ignored. Thus, only cases 4, 5, 7, and 8 require any action. With case 7, s is *vacant* and p is *covered*, so we know that P is visible at all raster samples within the subcell. Thus, we shade and display them (or tag them as belonging to P) and then change the status of s to *covered*. Cases 4, 5, and 8 require subdivision to determine where P is visible within the subcell, and for case 5, what the new status of s should be. We defer discussion of these cases until the hierarchical tiling algorithm is presented in §4.5, but the general idea is that raster samples covered by P are identified during recursive subdivision, and when

case	screen subcell s	polygon subcell p	resulting screen subcell s'	action
1	covered	covered	covered	do nothing
2	covered	active	covered	do nothing
3	covered	vacant	covered	do nothing
4	active	covered	active*	SUBDIVIDE
5	active	active	active†	SUBDIVIDE
6	active	vacant	active	do nothing
7	vacant	covered	covered	DISPLAY
8	vacant	active	vacant‡	SUBDIVIDE
9	vacant	vacant	vacant	do nothing

* propagation will change status to covered
† propagation may change status to covered
‡ propagation will change status to active

Table 4.1 Rules for front-to-back compositing of triage masks, subcell by subcell. Since polygons are traversed front to back, in compositing a polygon's mask with the existing screen mask, screen subcell s is composited "over" polygon subcell p. The hierarchical tiling algorithm actually performs compositing with boolean mask operations rather than subcell by subcell.

the status of a mask in the finest level of the pyramid changes from *vacant* or *active* to *active* or *covered*, this status information is propagated to coarser levels of the pyramid.

Figure 4.3 presents an example which includes all nine compositing cases. Suppose we have already composited the polygon shown in dashed lines into a vacant square on the screen, creating the screen mask of panel a. Now we composite with this mask the mask for the polygon shown in panel b. Applying the compositing rules of table 4.1, we obtain the new screen mask, shown in panel c, where each subcell is labeled with the number of the corresponding compositing case.

Next we consider how the compositing rules can be efficiently executed with boolean mask operations. Let (S_C, S_V) be the mask for a region of the screen and (P_C, P_V) be the mask for a polygon P that we are compositing. Recall that only compositing cases 4, 5, 7, and 8 require any action. First we handle case 7 by creating a bit mask for the subcells where S is *vacant* and P is *covered*: $W = S_V \,\&\, P_C$. We call this mask "W" because all raster samples in this region of the screen are covered by P and may now be *written* (or tagged). We update the status of the *covered* subcells in S as follows: $S_C = S_C \mid W$, $S_V = S_V \,\&\, \tilde{}W$. Cases 4, 5, and 8 require subdivision and can only be fully described in the context

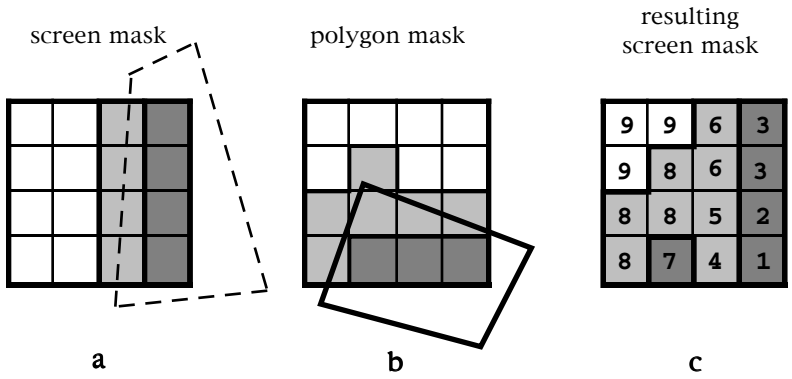


Figure 4.3 Example illustrating compositing of triage masks according to the compositing rules of table 4.1. The screen mask of panel a was produced by the dashed polygon. This screen mask is composited "over" the polygon mask of panel b, producing the new screen mask of panel c. In panel c, each subcell is numbered with the corresponding compositing case of table 4.1.

of the hierarchical tiling algorithm. However, the first step to is construct the bit mask for *active* subcells which require subdivision, which is done as follows: $A = \tilde{}(W \mid P_V \mid S_C)$. As we will see in §4.5, the hierarchical tiling algorithm recursively subdivides *active* subcells and updates the corresponding subcells of S by propagation from finer levels.

4.4.3 Building Lookup Tables

Before presenting the recursive tiling procedure in which these procedures are applied, we describe how lookup tables for triage masks are constructed. Although the method is generally analogous to building tables for conventional coverage masks, there are some important differences. In both cases, we divide the perimeter of a canonical square into some number of equal intervals and create an entry in a two-dimensional table for each pair of intervals not lying on a common edge. Once this table has been constructed, to obtain the mask for an arbitrary edge we determine which two intervals it crosses and look up the corresponding table entry. To conserve storage, we can use the same table entry for edges with opposite directions, since complementing the (C,V) bit masks in a triage mask corresponds to reversing an edge.

With conventional coverage masks, the accuracy of the table depends on the length of the intervals, the shorter the intervals, the more accurate the table. However, there can be classification errors even if intervals are very short. The problem is illustrated in figure 4.4a, where we see that within the quadrilateral defined by the endpoints of a pair of intervals, classification of grid points is ambiguous. Thus, for some edges, some grid

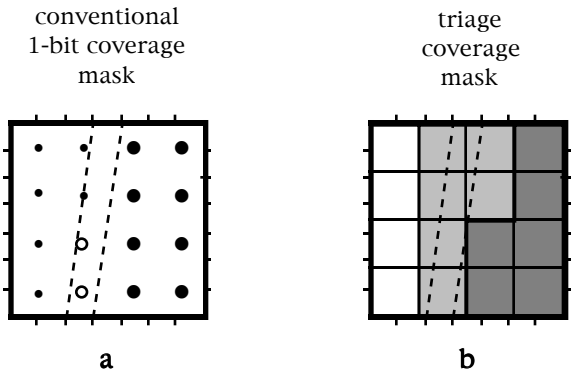


Figure 4.4 The endpoints of the pair of intervals used to index a coverage mask define a quadrilateral. With conventional one-bit coverage masks, the status of gridpoints inside the quadrilateral is ambiguous, for example, the gridpoints marked with small circles in panel a. With triage masks, however, any subcells intersected by the quadrilateral are classified *active*, guaranteeing that cells classified *covered* are completely covered and cells classified *vacant* are completely vacant (panel b). In practice, edges of triage masks are more finely divided than shown here.

points will be mis-classified by the precomputed edge mask in the lookup table. In practice, small errors in classification are not a problem when applying coverage masks to filtering subpixel samples. In fact, caching performance is generally considered more important than accuracy, so typical implementations use low-resolution tables that routinely mis-classify a few samples. For example, Sabella and Wozny use 28×28 tables [Sabella-Wozny83].

The hierarchical tiling algorithm, however, depends on accurate classification of *vacant* and *covered* regions in triage masks. Accordingly, in constructing a mask we classify any subcell which intersects the quadrilateral defined by interval endpoints as *active* (see figure 4.4b). Thus, cells classified as *covered* are completely covered and cells classified as *vacant* are completely vacant. This conservative procedure guarantees accurate results, regardless of interval length. The only advantage of dividing the square's edges into shorter intervals is that, on average, a smaller fraction of subcells will be classified as *active*, which reduces the amount of subdivision the tiling algorithm needs to perform. In addition to creating a table of triage masks, we also create a table of conventional one-bit coverage masks to determine coverage of raster samples at the finest level of the mask pyramid.

4.5 The Hierarchical Tiling Algorithm

As discussed in §4.3.1, the "depth-priority" variation of the Warnock algorithm that processes primitives in front-to-back order is simpler and more efficient than the traditional

algorithm, provided that front-to-back traversal of primitives is convenient. While depth-priority Warnock is a big improvement over the traditional algorithm, it is still relatively slow due to the computational expense of classifying primitives as inside, outside, and intersecting quadrants. We use the basic recursive subdivision procedure of depth-priority Warnock as the core of our hierarchical tiling algorithm and circumvent the classification bottleneck by precomputing classification patterns and storing them as triage coverage masks. The mechanism by which the masks control subdivision is a straightforward application of the compositing rules for masks. We will discuss this procedure after describing the underlying data structures: the mask pyramid, the image array, and the model tree.

4.5.1 Data Structures

We maintain visibility information about previously tiled polygons in an image-space pyramid. The finest level of the pyramid is a one-bit image indicating whether or not each point sample in the image raster has been covered. All other levels of the pyramid are two-bit images, each element of which corresponds to a square region of the screen, indicating whether it is *covered*, *vacant*, or *active*. Similarly, Meagher's volume rendering algorithm represents the screen as a quadtree, each element of which distinguishes these same three states [Meagher82b]. We use a pyramid rather than an adaptively subdivided hierarchy to exploit the efficiency of static memory allocation.

To permit Warnock subdivision to be driven by boolean mask operations, we build the image-space pyramid from coverage masks. As schematically illustrated in figure 4.5, a single triage mask represents the whole screen, triage masks at the next level of the pyramid correspond to subcells in the root mask, and so forth. Thus, this *mask pyramid* is a hierarchical representation of the screen with the C and V bits for each subcell in the triage masks indicating whether a square region of the screen is *covered*, *vacant*, or *active*. At the finest level of the pyramid only, we use conventional one-bit coverage masks to indicate whether or not point samples in the image raster have been covered. If we are performing A-buffer-style filtering, each of these one-bit masks corresponds to the 8×8 grid of raster samples within a pixel. The appropriate pyramid for a 512×512 image with 8×8 oversampling at each pixel has four levels, three arrays of triage masks with dimensions 1×1, 8×8, and 64×64, and one 512×512 array of one-bit masks. Alternatively, if we are point sampling rather than filtering, each one-bit mask corresponds to an 8×8 block of pixels. In this case, the pyramid for a 512×512 image would have two arrays of triage masks with dimensions 1×1 and 8×8, and one 64×64 array of one-bit masks.

Memory requirements for the mask pyramid are very modest. Since the finest level

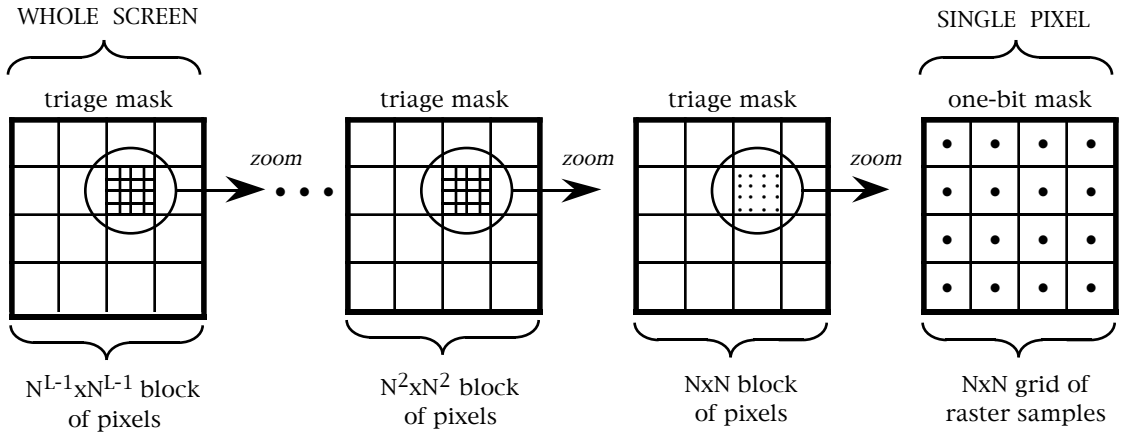


Figure 4.5 Schematic diagram of a pyramid of $N \times N$ masks with L levels for an image with $N \times N$ oversampling at each pixel. This *mask pyramid* is built from triage masks, except at the finest level where a conventional one-bit coverage mask is associated with each pixel. In this hierarchical representation of the screen, the C and V bits for each subcell in triage masks indicate whether a square region of the screen is *covered*, *vacant*, or *active*. At the coarsest level, a single triage mask represents the whole screen (left), and at the finest level, a single one-bit mask represents the raster samples within a pixel (right). The corresponding diagram for a point-sampled image is the same, except that the masks represent an $N \times N$ block of pixels, an $N^2 \times N^2$ block of pixels, and so forth. In practice, we use 8×8 masks. A four-level pyramid of 8×8 masks corresponds to a 512×512 image with 8×8 oversampling at each pixel.

---

requires only one bit per raster sample and the vast majority of cells in the pyramid are in the finest level, total memory requirements are only slightly more than one bit per raster sample. The actual number of bits per raster sample required for an n-level pyramid lies in the range [1 1/32 1 2/63) for n>1. Note that storing depth values for each raster sample in a z-buffer requires a great deal more memory.

The other image-space data structure that our algorithm requires is an image array with an element for each color component at each pixel. If we perform A-buffer-style filtering [Carpenter84], shading contributions from 64 subpixel samples accumulate in each array element. Thus, elements in this *accumulation buffer* require considerable depth. We use 16 bits per pixel per color channel. If no filtering is performed, pixel values do not accumulate, so a conventional image array is employed.

Now for representing the model. Our algorithm requires front-to-back traversal of polygons in the scene, so we represent the scene as a binary space partitioning tree (BSP tree) [Fuchs-Kedem-Naylor80], which permits very efficient traversal in depth order. If some polygons are in motion, it is necessary to update the BSP tree every frame. This does not

require much computation if the scene is predominantly static [Naylor92a], but if a large number of polygons are moving, maintaining the tree can be expensive. In extreme cases, the time spent maintaining the tree can exceed the time saved by our hierarchical tiling procedure, in which case our algorithm offers no advantage over traditional methods such as z-buffering. This potential difficulty with dynamic scenes is the biggest limitation of our algorithm.

4.5.2 The Basic Hierarchical Tiling Algorithm

Now that the primitive mask operations and the underlying data structures have been presented, the procedure for creating a frame is easy to describe. Details of the algorithm vary depending on filtering procedure, if any. First we present the algorithm in the context of A-buffer-style area sampling, saving discussion of other variations until later.

Generating a Frame

In a precomputation step, we build a BSP tree for the model and build lookup tables for both triage and conventional coverage masks. We begin a frame by clearing entries in the accumulation buffer to zero and clearing masks in the pyramid to *vacant*. We traverse polygons in the BSP tree front to back, tiling them one at a time into the mask pyramid, adding color values for each of their visible samples into the accumulation buffer. The tiling procedure also identifies *covered* and *active* regions of the screen and maintains this information in the mask pyramid. After all polygons have been processed, we obtain the output image by scanning through the accumulation buffer and dividing each color value by 64, the number of raster samples per pixel.

Tiling a Polygon with Area Sampling

Within this overall algorithm, the procedure for tiling a polygon with A-buffer-style filtering [Carpenter84] is as follows. We clip each polygon P to the front clipping plane in object space, if necessary, and project its vertices into the image plane. There is no need to preserve depth information. Now we begin the recursive subdivision procedure that tiles P into the mask pyramid, beginning with the root mask which corresponds to the whole screen. The procedure summarized here is also outlined in pseudocode in LISTING 4.1. We create P's mask by finding the intersection points of its edges with the border of the pyramid mask (initially, the root mask), looking up the corresponding edge masks, and combining them as described in §4.4.2. We then composite P's mask with the pyramid mask using the

compositing rules of §4.4.2. At subcells where the "*W*" bit mask (see §4.4.2) indicates that *P* is entirely visible, we evaluate the shading function for each pixel in the corresponding region of the screen and update the accumulation buffer. Since *P* covers all 8×8 samples at these pixels, we add 64 times the evaluated color to the accumulation buffer. No visible edges cross these pixels, so edge filtering is not called for. We also change the status of the corresponding subcells in the mask pyramid to *covered*. At subcells where the "*A*" bit mask (see §4.4.2) indicates that subdivision is required, we call this same tiling procedure recursively with the appropriate pyramid mask and the edges of *P* that intersect it. In regions of the screen crossed by visible edges, recursive subdivision continues down to the pixel level where A-buffer-style filtering is performed on visible samples using conventional one-bit coverage masks. At these pixels, shading is evaluated and color values are multiplied by the bit count of the coverage mask before they are added to the accumulation buffer. In addition to updating the accumulation buffer, we also need to update the mask pyramid by propagating coverage information to coarser levels, which is accomplished easily with recursively applied mask operations (see pseudocode). When this procedure finishes, all visible raster samples covered by *P* have been evaluated and added to the accumulation buffer, and the appropriate cells in the mask pyramid have been updated. Since visibility is computed on an 8×8 grid of raster samples within each pixel, the resulting image has very high-quality antialiasing.

Other Filtering Methods

The filtering method outlined above performs A-buffer-style *area sampling*, a term used to describe convolution of visible samples with a pixel-sized box filter [Catmull78]. As with the original A-buffer algorithm [Carpenter84], to improve efficiency only one shading evaluation is performed for each primitive at each pixel. This generally produces acceptable results, although results are not as accurate as when shading is performed at all subpixel samples. It is interesting to note that for the important special case of Gouraud shading, the contributions of all visible samples within a pixel can be computed all at once using precomputed coefficients stored in lookup tables. This technique was proposed by Abram, Westover, and Whitted, who also experimented with coverage masks for jittered samples and table-driven convolution with arbitrary convolution filters [Abram-et-al85]. All of these refinements are compatible with our algorithm and would be expected to improve accuracy, although we have obtained good results with A-buffer-style area sampling, which is simpler and faster.

Two other variations on the algorithm outlined in pseudocode should be mentioned. First, image resolution doesn't need to be a power of eight. For example, if the resolution of

LISTING 4.1 (PSEUDOCODE)

```
/*
Recursive subdivision procedure for tiling a convex polygon P.

After clipping P to the near clipping plane in object space, if necessary, and
projecting P's vertices into the image plane, we call tile_poly with the root
mask of the mask pyramid, P's edge list, and "level" set to 1.

arguments:
(Sc,Sv): pyramid mask (input and output)
edge_list: P's edges that intersect pyramid mask
level: pyramid level: 1 is root, 2 is next coarsest, etc.
*/

tile_poly((Sc,Sv), edge_list, level)
{
   set active_edge_list to nil

   /* build P's mask (Pc,Pv) */
   Pc = all_ones
   Pv = all_zeros
   for each edge on edge_list {
      find intercepts on square perimeter of mask
      if square is outside edge, then return /* polygon doesn't intersect mask */
      if edge intersects square, then {
         append edge to active_edge_list
         /* Note: at pixel level, Ec is conventional coverage mask, Ev = ~Ec */
         look up edge mask (Ec, Ev)
         Pc = Pc & Ec
         Pv = Pv | Ev
      }
   }

   /* make "write" bit mask and update pyramid mask */
   W = Sv & Pc
   Sc = Sc | W
   Sv = Sv & ~W

   if level is the pixel level, then {
      /* filter pixel with coverage mask W */
      evaluate shading and add bitcount(W)*(pixel color) to accumulation buffer
      return
   }

   for each TRUE bit in W { /* compositing case 7 */
      for each pixel in this square region of screen /* all 64 samples covered */
         evaluate shading and add 64*(pixel color) to accumulation buffer

   /* RECURSION */

   /* make "active" bit mask and subdivide active subcells */
   A = ~(W | Pv | Pc)
   for each TRUE bit in A { /* compositing cases 4, 5, 8 */
      /* call corresponding subcell C; call its pyramid mask (Cc,Cv) */
      copy all edges on active_edge_list that intersect C to C_edge_list
      tile_poly((Cc,Cv), C_edge_list, level+1)
      /* propagate coverage status to coarser levels of mask pyramid */
      if Cc==all_ones then Sc = Sc | active_bit /* set covered status */
      if Cv!=all_ones then Sv = Sv & ~active_bit /* clear vacant status */
   }
}
```

the raster grid is 4096×4096, instead of filtering 8×8 blocks of samples to produce a 512×512 image, we could instead filter 4×4 blocks of a 1024×1024 image. Second, modifying the algorithm to produce point-sampled rather than filtered images is straightforward. In this case, each mask at the finest level of the pyramid corresponds to an 8×8 block of pixels. So for each TRUE subcell in the "W" mask (see pseudocode), we evaluate the shading function at the corresponding pixel and write the result to the image buffer. Since pixel values correspond to point samples, color values do not accumulate, so we use a conventional image array rather than an accumulation buffer. Note that it is not necessary to clear the image array at the beginning of a frame. Instead, after tiling all scene polygons, we composite a screen-sized polygon of the desired background color (or texture) with the root mask, thereby tiling all remaining vacant pixels in the image.

Properties of the Hierarchical Tiling Algorithm

Hierarchical polygon tiling combines desirable features of traditional tiling algorithms: logarithmic search for visible geometry, precomputation of tiling patterns, and incremental evaluation. It is the only tiling algorithm that effectively exploits all of these methods. Since the recursive subdivision procedure only visits cells in the mask pyramid that are crossed (or nearly crossed, in some rare cases) by visible edges in the output image, it finds visible geometry by logarithmic search. Logarithmic search can be extended to finding visible subcells within masks by using bit patterns to control a switch statement. Once visible samples are found, front-to-back traversal of polygons guarantees that they are never overwritten, so shading evaluation is never wasted. Regarding precomputation, since tables of triage tiling patterns are precomputed, the only geometric operations required to classify cells as inside, outside, or intersecting an edge are finding intercepts of edges on mask borders, and this can be done incrementally, except at the root mask. In fact, the only other geometric operations relating to visibility that must be performed on polygons are clipping to the near clipping plane, if necessary, and projecting vertices into the image plane. Polygons are automatically clipped to the screen border when they are composited with the root mask. The algorithm's geometric simplicity makes it an attractive candidate for hardware implementation. Regarding incremental evaluation, the algorithm identifies hierarchical blocks of pixels covered by the same polygon, so it is straightforward to apply standard incremental methods for evaluating shading functions. Moreover, indices into image data structures can be computed incrementally, as can edge-mask intercepts, as mentioned above. Finally, we note that the algorithm is probably well suited to parallel implementation on a moderate scale, since it exploits image-space coherence in much the same way as the hierarchical z-buffer visibility algorithm, which has good parallel performance as discussed in §3.5.5.

4.6 Hierarchical Object-Space Culling

Because of its ability to cull hierarchically in image space, the hierarchical tiling algorithm processes densely occluded scenes much more efficiently than conventional tiling methods, which must traverse hidden geometry pixel by pixel. Nonetheless, it must consider every polygon in a scene, doing some work even on those that are entirely hidden. To avoid this behavior, we integrate our algorithm with the basic hierarchical visibility algorithm to enable object-space culling of hidden regions of the model. This can be done by substituting hierarchical tiling for z-buffering in the hierarchical z-buffer algorithm of chapter 3, which requires some changes in both the object-space and image-space hierarchies. In image space, instead of using a z-pyramid of depth samples to maintain visibility information, we use a mask pyramid. In object space, we modify the octree to permit strict front-to-back traversal of polygons. Note that the z-buffer algorithm traverses octree cubes in front-to-back order, but not the primitives contained in them. And since octree cubes are nested, it is not sufficient to simply organize the primitives inside each cube in a BSP tree. Instead we use the following algorithm for building an *octree of BSP trees* that does permit strict front-to-back traversal.

4.6.1 Building and Maintaining an Octree of BSP Trees

Starting with a root cube which bounds model space, we insert polygons one at a time into the cube. If the polygon count in the cube reaches a specified threshold (e.g. 80), we subdivide the cube into eight octants and insert each of its polygons into each octant that it intersects, clipping to the cube's three median planes. When all polygons in the scene have been inserted into the root cube and propagated through the tree, we have an octree where all polygons are only associated with leaf nodes, thereby circumventing the ordering problem caused by nesting. The last step is to organize the polygons in each leaf node of the octree into a BSP tree [Foley-et-al90]. Now scene polygons can be traversed in strict front-to-back order by traversing octree cubes front to back and traversing their BSP trees front to back.

It should be noted that the cost of maintaining the object-space hierarchy, whether it is a single BSP tree or an octree of BSP trees, can potentially limit the algorithm's performance in processing dynamic scenes. While this is not a major problem with predominantly static scenes, it can seriously erode performance when a great many polygons are moving, and in extreme cases, the cost of maintaining the hierarchy can exceed the time saved by algorithm's tiling efficiency, in which case the algorithm offers no advantage over other methods. For a discussion of strategies for efficiently maintaining octrees of dynamic scenes,

see §3.4. For a discussion of maintaining BSP trees of dynamic scenes, see [Naylor92b]. Note that it is only necessary to maintain the BSP trees in visible octree nodes, since only their polygons are traversed.

4.6.2 Combining Hierarchical Tiling with Hierarchical Visibility

Now that we have established how to organize a scene model into an octree of BSP trees, combining hierarchical tiling with the basic hierarchical visibility algorithm is very straightforward. As with the hierarchical z-buffer visibility algorithm, we traverse octree cubes in front to back order, testing them for visibility and culling those that are hidden. To test cubes for visibility, we modify the tiling procedure of LISTING 4.1 to report visibility status of polygons, returning TRUE whenever a polygon's mask indicates that it covers a subcell or a point in the image raster. Cube-visibility testing can be done much more efficiently with hierarchical tiling than with z-buffering, because tiling is strictly two dimensional, so the visibility of a cube can usually be established by tiling its polygonal silhouette. By comparison, z-buffering often needs to tile three faces of a cube to establish its visibility. Once we have established that an octree cube is visible, we traverse the polygons in its BSP tree front to back, tiling them into the mask pyramid and updating the image buffer when visible samples are encountered. When we finish traversing the octree, all visible polygons have been tiled and the image is complete.

This version of the hierarchical visibility algorithm has very efficient traversal properties in both object-space and image-space. Like the hierarchical z-buffer algorithm of chapter 3, in object space the algorithm only visits visible octree nodes and their children, and it only renders polygons that are in visible octree nodes. In image space, when tiling polygons into the mask pyramid it only visits cells that are crossed by visible edges in the output image. Visible samples are never overwritten. As a result of these properties, this variation of the hierarchical visibility algorithm is very efficient in culling hidden geometry and tiling visible geometry.

4.6.3 Accelerating Image-Space Culling

In addition to enabling object-space culling, organizing polygons in bounding boxes can also accelerate image-space culling. A polygon can only be visible at samples where its bounding box is visible, and this can be exploited in two ways. First, tiling a polygon can begin at the smallest mask in the pyramid that contains all of its bounding box's visible samples. Secondly, there is a simple strategy that avoids attempting to tile any polygon

into any mask in which it has no visible samples, which works as follows. When a cube is tested for visibility, we traverse all visible samples on the cube and store the count of visible samples with each mask in the pyramid that is traversed. When polygons contained in the cube are tiled, this count is decremented when visible samples are covered. If the count for a particular mask falls to zero, we know that no visible samples remain within the cube's silhouette, so the corresponding region of the screen can be ignored. This strategy will often prevent unnecessary subdivision, because masks will often contain visible samples that are outside the cube's silhouette. And if the count falls to zero at the coarsest level of subdivision, indicating that all samples within the cube's silhouette are covered, we know that all remaining polygons in that subtree of the octree are hidden and can be culled.

4.7 Implementation and Results

Our initial implementation of hierarchical tiling is programmed in C and is integrated with the hierarchical visibility algorithm as described in the preceding section. It renders either point-sampled or box-filtered images with either flat or Gouraud shading. Filtering is done using the area-sampling method outlined in pseudocode. The accumulation buffer for filtered images is 16 bits deep in each color channel. Triage masks were constructed with 64 intervals along an edge. We tested our algorithm by generating antialiased motion sequences of a densely occluded scene, and observed good-quality antialiasing. Figure 4.6 is a frame from this animation.

We tested our implementation on a version of the office-interior model described in chapter 3. We built an octree of BSP trees for the repeating module, each BSP tree containing approximately 16,000 polygons, and replicated the octree in a 33×33×33 grid to construct a densely occluded scene with upwards of 500 million polygons. The hierarchical tiling algorithm with object-space culling rendered 512×512 antialiased frames with flat shading in approximately 11 seconds on a 50Mhz R4000 SGI Crimson. As our area-sampling method calls for, a single shading value was computed for each visible primitive at each pixel, and visibility was determined at all samples on a uniform 4096×4096 grid. Rendering this same scene with Gouraud shading took an additional two seconds per frame. By comparison, the hierarchical z-buffer algorithm of chapter 3 took approximately 5 seconds to render point-sampled flat-shaded 512×512 images of this model. Although this is 2.2 times faster than our implementation of hierarchical tiling, hierarchical tiling determines visibility at 64 times as many samples. When this is taken into consideration, we see that the hierarchical tiling implementation processed visible samples 29 times faster than the z-buffer algorithm. It should also be noted that many more primitives are visible on the finer raster.

Figure 4.6 A frame from a motion sequence through a scene containing upwards of 500 million replicated polygons. Our unoptimized implementation of hierarchical tiling with object-space culling rendered this flat-shaded 512×512 antialiased image in approximately 11 seconds on a 50Mhz R4000 SGI Crimson. Visibility was determined on a 4096×4096 raster to permit A-buffer-style filtering on an 8×8 grid within each pixel.

As previously mentioned, we observed good-quality antialiasing in motion sequences of this scene with our implementation of hierarchical tiling. By comparison, images produced with the z-buffer algorithm aliased badly as expected.

It should be noted that these results were produced by our initial implementation which fails to exploit a number of optimizations and opportunities to exploit incremental methods. For example, polygon tiling always begins at the root mask and we don't use the image-space bounds of octree cubes to prevent unnecessary subdivision as described in §4.6.3. Also, we don't use traditional incremental evaluation of Gouraud-shading values, even though the algorithm facilitates this by identifying blocks of pixels in which a single polygon is visible. In addition, we use floating-point rather than fixed-point arithmetic throughout. All things considered, our preliminary results suggest that a careful implementation of the hierarchical tiling algorithm on the latest generation of microprocessors would be able to render frames of our replicated office model at frame rates that permit convenient interaction.

Unfortunately, we haven't yet tested our algorithm on relatively simple scenes, so we are not able to cite performance figures. However, given the algorithm's exceptional tiling efficiency and its conservative memory usage, we expect that it would also perform very

Thing Being Compared	Hierarchical Tiling	Hierarchical Z-Buffering
object-space hierarchy	BSP tree / octree of BSP trees	octree
image-space hierarchy	pyramid of coverage masks	z-pyramid
front-to-back polygon traversal required?	yes	no
visibility info per raster sample	< 1 2/63 coverage-mask bits	Z (usu. 24-32 bits)
color info per raster sample	none	RGB (usu. 24-36 bits)
type of output-image buffer	accumulation (deep)	standard
need to store triage LUTs?	yes	no
pixel overwrite?	no	yes
support for A-buffer filtering built-in?	yes	no
identifies *covered* image-pyramid cells?	yes	no
able to exploit z-buffer hardware?	no	yes

Table 4.2 Some points of comparison between the hierarchical polygon tiling algorithm and the hierarchical z-buffering algorithm of chapter 3, assuming that we are oversampling the image to enable A-buffer-style filtering.

well in this domain.

4.8 Hierarchical Tiling versus Hierarchical Z-Buffering

Table 4.2 summarizes some points of comparison between the hierarchical polygon tiling algorithm and the hierarchical z-buffer algorithm of chapter 3, assuming that we are oversampling the image to enable A-buffer-style filtering. As this table points out, hierarchical tiling requires strict front-to-back traversal of polygons, which complicates the object-space hierarchy, assuming that we are maintaining an octree of BSP trees to enable object-space culling. Other points in favor of hierarchical z-buffering are that it can exploit hardware z-buffer accelerators with its temporal-coherence procedure and that it doesn't need to build or store lookup tables for triage coverage masks. The other points of comparison strongly favor hierarchical tiling. One big advantage is that its memory requirements are much less. Whereas hierarchical z-buffering needs to store depth and color information with each raster sample, hierarchical tiling only needs to store slightly more than one bit of coverage information for each raster sample. The resulting memory savings can be very substantial. In fact, if we are rendering a 512×512 image with 8×8 oversampling at each pixel, hierarchical tiling requires only about 3.7% of the image memory required by z-buffering. Other points in favor of hierarchical tiling are that it never overwrites visible

samples, it has built-in support for A-buffer-style filtering, and it facilitates exploiting image-space coherence by identifying regions of the image-space hierarchy that are completely covered by individual primitives.

4.9 Conclusion

Hierarchical polygon tiling in combination with the basic hierarchical visibility algorithm simultaneously exploits precomputation, hierarchical culling in both object space and image space, and the parallelism inherent in boolean mask operations. To exploit precomputation, triage tiling patterns are precomputed and stored in lookup tables and the scene model is presorted in an octree of BSP trees. The main limitation of the algorithm is that it can be costly to maintain the octree of BSP trees if numerous primitives are moving independently. These data structures permit tiling and visibility operations to be performed hierarchically, with triage masks driving recursive subdivision of image space. The result is a very efficient subdivision procedure that only visits visible nodes in the octree, only visits cells in the image-space hierarchy that are crossed by visible edges, and never overwrites a visible sample. The algorithm is compact and straightforward to implement, has very modest memory requirements, and is amenable to parallel implementation. In short, hierarchical visibility and tiling methods working together offer the prospect of rendering very complex scenes at rapid frame rates with modest computing resources.

Chapter 5

Error-Bounded Antialiased Rendering

5.1 Overview

In this chapter we present a variation of the basic hierarchical visibility algorithm that is capable of antialiased rendering with guaranteed accuracy. This approach to antialiased rendering is slower but more robust than the hierarchical tiling algorithm of the preceding chapter. If visible geometry within individual pixels is extremely complex, uniform oversampling methods like those of hierarchical tiling may not produce good results. In order to guarantee accurate filtering of arbitrarily complex geometry, the algorithm advanced in this chapter uses interval methods to control recursive subdivision of the image-space hierarchy. This permits identifying regions of the screen where geometry or shading is complex, and then doing as much work as necessary within those regions to produce accurate results. Consequently, each pixel of the output image can be guaranteed to be within a user-specified error tolerance of the filtered underlying continuous image. To the best of my knowledge, the images produced with this algorithm are the only computer-generated images of guaranteed accuracy that have ever been created of extremely complex scenes or scenes rendered with complex shaders.

Our work on this algorithm preceded the development of the hierarchical tiling algorithm of chapter 4. In working on hierarchical tiling, we became aware that strict front-to-back traversal of primitives makes recursive subdivision of image space simpler and more efficient by eliminating the need for depth comparisons, improving culling efficiency, and preventing unnecessary subdivision of the image-space hierarchy. The algorithm described in this chapter would also benefit from this approach, although this would require changing various details of the algorithm as presented. One important difference is that primitives would need to be maintained in an octree of BSP trees rather than in an ordinary octree, as described in §4.6.

5.2 Introduction

The fact that computer-generated images frequently contain step edges and other high-frequency detail has always posed serious aliasing problems for rendering algorithms [Crow77]. Even with special measures to combat aliasing, jaggies and other familiar aliasing

artifacts frequently occur. The aliasing difficulties we have with computer graphics today, however, may seem mild compared to the aliasing problems that will become commonplace in a few years. As inexpensive computers become more and more powerful, computer graphics models will become increasingly detailed, and we will regularly be faced with images that contain large numbers of sub-pixel polygons. These sub-pixel polygons will create a great deal of energy at high spatial frequencies and cause severe aliasing for current algorithms.

The left panel of figure 5.2 shows the severity of aliasing that can result from standard z-buffer rendering of a 167-million polygon model of the Empire State Building, shown antialiased in the right panel of figure 5.2 and also in figure 5.1. The exterior skin of the building is represented by large multi-pixel polygons that cause the usual aliasing artifacts we have come to expect from z-buffer rendering. The interior of the building, however, consists of an extremely large number of polygons, and it exhibits a qualitatively different level of aliasing. Each of the floors of the model is populated with hundreds of office cubicles, each consisting of thousands of polygons. The small image at lower left in figure 5.1 shows a view of the interior. Through some of the windows in figure 5.1, large numbers of polygons are visible within individual pixels. With z-buffer rendering, each pixel is colored using just the front-most polygon that crosses the pixel center, but with so many polygons covering portions of a single pixel, the color that happens to occur at the center is likely to differ greatly from the local average, causing a disturbing visual artifact.

Uniform oversampling is often used to combat aliasing, but with a model of this complexity, no reasonable amount of oversampling can be guaranteed to produce good results. Moreover, for many applications, such as animation, it is important to be able to produce an acceptable antialiased result without user intervention. If the geometry is very complicated within a particular pixel of a frame, the rendering algorithm should detect the problem and work as hard as necessary on that pixel to avoid aliasing.

Here we present the first rendering algorithm capable of dealing with models as complex as our model of the Empire State Building while guaranteeing proper antialiasing to within a desired error tolerance. Our algorithm extends the techniques presented in chapter 3 to cull hidden geometry very quickly. Having discarded most hidden geometry, it uses Warnock-style subdivision [Warnock69] to bound the convolution integral at each pixel to within the desired error tolerance. For flat-shaded or Gouraud-shaded polygons, it is possible to compute the integral exactly. For more complicated shading functions, we rely on interval analysis to provide bounds. In any case, for each pixel, the algorithm terminates when it computes the exact result or when it can prove a sufficiently good error bound that further subdivision is unnecessary.

In §5.3 we discuss the aliasing problem in general and the relationship of our method to

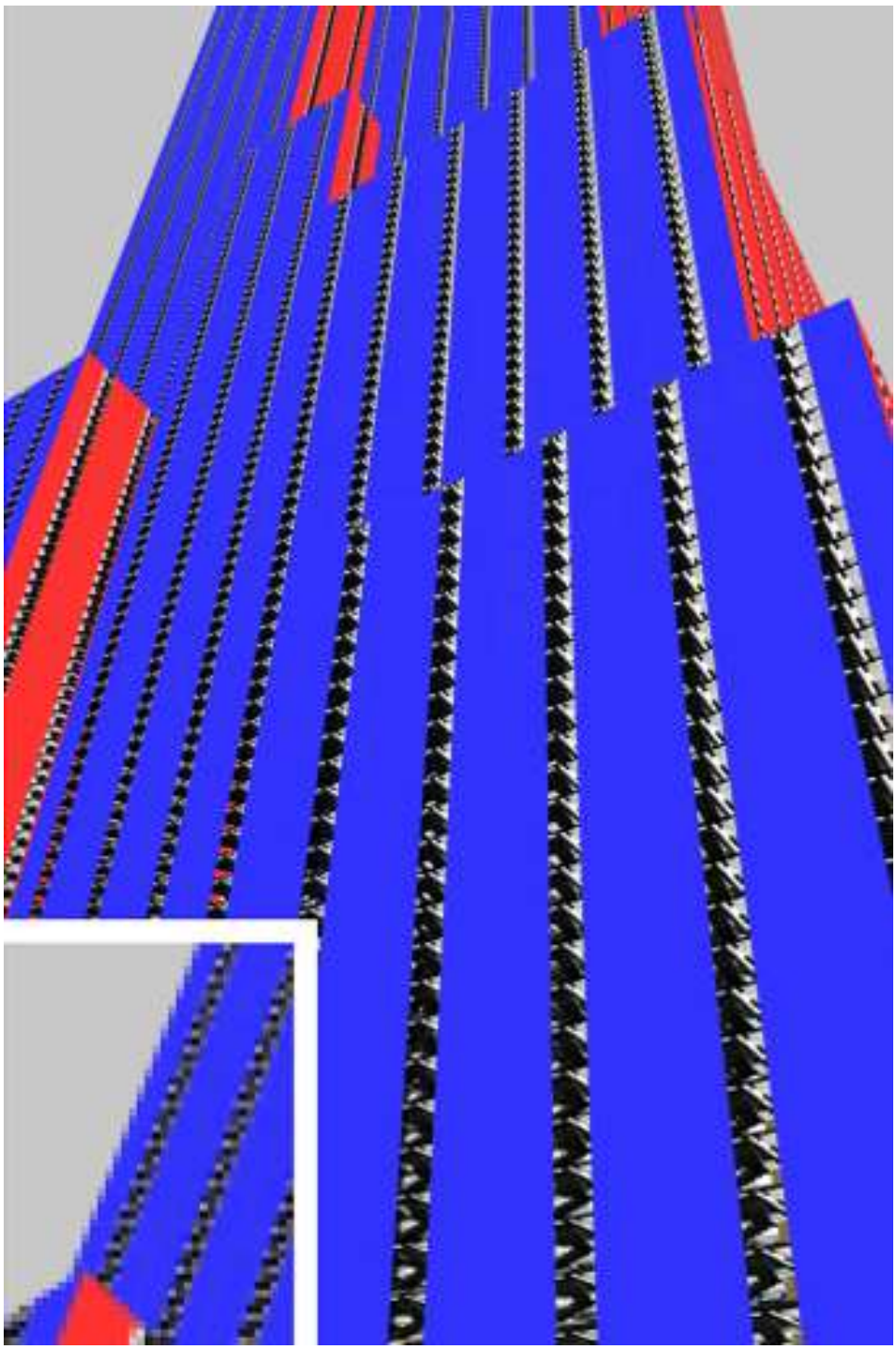


Figure 5.1 A model of the Empire State Building consisting of 167 million polygons rendered with the error-bounded rendering algorithm. The image at lower left is a view of the interior.

previous work. In §5.4 we present the details of our algorithm and describe its two stages. In the first stage, the algorithm culls hidden geometry using an octree, and in the second stage, it uses Warnock-style subdivision to achieve the desired error bound. In §5.5 we describe our implementation of the algorithm and show results. In §5.6 we describe extensions of the algorithm that should make it possible to compute motion blur. Finally, in §5.7 we state our conclusions.

5.3 Aliasing

Reiterating the discussion of aliasing in §2.2.1, the potential for aliasing arises in computer graphics because the mathematical representations that we ordinarily use to describe images (e.g. polygons) contain energy at arbitrarily high spatial frequencies, while the sampled rasters we use to display and print images are limited to a finite range of spatial frequencies. Let $I(x, y)$ be the vector-valued function that gives the color of each point in $\Re^2$ for the idealized mathematical representation of a computer-graphics image. If we compute a raster image by directly sampling $I(x, y)$ at the center of each output pixel, as is done with the hierarchical z-buffer algorithm presented in chapter 3, then any spatial frequency content in $I(x, y)$ beyond half the sampling rate will alias to a lower frequency and cause disturbing visual artifacts [Foley-et-al90].

Following the discussion in §2.2.2, there are three approaches for dealing with this problem: a) adjust the number, locations, or weights of the samples, b) detect aliasing artifacts in the rendered image and remove them by post-processing, and c) compute or approximate the convolution $I(x, y) * f(x, y)$ of the image $I(x, y)$ with a low-pass filter $f(x, y)$ at each output pixel. Approach c), *convolution before sampling*, is the only approach that is capable of guaranteed accuracy.

The problem with the first approach, adjusting the number, locations, or weights of the samples, is that it provides no guarantees about the quality of the result and can produce unacceptable errors in cases where a large number of primitives are visible within a single pixel. Although aliasing can be converted to less disturbing noise by placing samples stochastically, the noise may still be unacceptably high in some areas. The problem is that we do not know in advance what sampling rate will be required for any particular region of the image. Any algorithm that uses a fixed sampling rate, such as the 8×8 oversampling employed by the hierarchical tiling algorithm of chapter 4, will be unable to deal with the extreme geometry in figure 5.1 where large numbers of polygons are frequently visible within a single pixel. If we use the variance of a collection of rays through a pixel to determine a local sampling rate for ray casting [Lee-Redner-Uselton85, Dippe-Wold85], the problem is

the number of samples that must be employed to get good results, since the accuracy of the result is proportional to the square root of the number of samples. It can be shown using analysis methods described by Lee et al. [Lee-Redner-Uselton85] that it may be necessary to cast hundreds of rays through a pixel having very complex visible geometry to achieve high confidence that the pixel value is accurate. Thus, this approach is not an efficient way to make very accurate images. Rendering figure 5.1 by casting hundreds of rays to antialias each of the complex pixels would be prohibitively expensive. Even if we used that many rays, accuracy is measured statistically and some pixels would still have some significant error. In short, traditional oversampling methods can be applied to antialiasing scenes in which numerous primitives are visible within individual pixels, but they have shortcomings in both quality and efficiency.

The second approach to combating aliasing, post-processing [Bloomenthal83, Fujimoto-Iwata83]), is not appropriate for images with numerous sub-pixel polygons, such as figure 5.1, because too much information is lost in the sampling process to allow a post-process to compute an acceptable reconstruction.

The third approach, convolution before sampling [Crow77, Catmull78], is the only technique that is, in principle, capable of eliminating aliasing entirely. Rendering with this approach requires (a) identifying the visible geometric primitives affecting each output pixel and (b) filtering them.

Addressing the visibility component of the problem first, most visible-surface algorithms capable of finding all the geometric primitives potentially affecting a single pixel require examining each primitive in the model (e.g. [Catmull78, Catmull84, Weiler-Atherton77, Warnock69, Sharir-Overmars92, Sechrest-Greenberg82, Sequin-Wensley85, Naylor92b]). For ordinary purposes with moderately complex models, this is not a serious limitation. For a model as complex as that of figure 5.1, however, this poses a major problem. Examining and processing each of the 167 million primitives in the model would take a prohibitively long time on contemporary computers. Instead, we should use a visibility algorithm that can efficiently cull much of the hidden geometry in densely occluded scenes, such as the algorithms discussed in §2.1.6. These algorithms include potentially visible set methods, Naylor's BSP-tree method, and the hierarchical visibility method presented in chapter 3.

As mentioned in §2.1.6, potentially visible set methods [Airey90, Teller-Sequin91, Teller92, Luebke-Georges95] effectively cull hidden regions of architectural models, so within this limited domain, this approach could be adapted to accelerate visibility computations for an antialiased renderer. Another alternative is Naylor's BSP-tree visibility algorithm [Naylor92b]. However, as mentioned in §2.1.6, this algorithm has not actually been demonstrated to work effectively on complex models, and in addition it might need to be modified

to cull enough hidden geometry to work effectively, since the algorithm as presented does not cull all hidden subtrees. In any case, we use the hierarchical visibility algorithm because it culls hidden geometry very effectively in scenes with high depth complexity and is well suited to the type of model in figure 5.1.

This is the visibility algorithm that we use with our antialiased rendering algorithm, although it is necessary to modify the algorithm to support filtering before sampling. With z-buffer rendering, geometry that is hidden on all point samples can be culled whether or not portions of the geometry are visible in between samples. With antialiased rendering, however, geometry should be culled only if it is completely hidden. To facilitate culling, we insert the model geometry into a quadtree data structure. If the geometry in a quadtree cell is simple, we can quickly determine whether or not the geometry completely hides a given primitive. Otherwise, we use Warnock-style subdivision to create simpler children.

Once the visible polygons affecting a pixel have been identified, they need to be convolved with the desired filter and summed. Fast algorithms using lookup tables have been developed to compute the convolution quickly for arbitrary filters with flat-shaded or Gouraud-shaded polygons [Feibush-Levoy-Cook80, Abram-et-al85], and for Gaussian or box-filtered texture-mapped polygons [Williams83, Crow84]. In §5.4.4 we show how to use interval analysis to extend the range of cases that can be accurately filtered to a broad class of shading functions.

5.4 The Rendering Algorithm

Our algorithm employs two key data structures in order to antialias very complex models with guaranteed accuracy. The first data structure is an object-space octree used to organize the model polygons in world space. The second is an image-space quadtree used to organize information about potentially visible polygons in each region of the screen. With these two data structures, the algorithm culls hidden geometry very quickly and establishes color bounds for each output pixel.

5.4.1 Construction and Rendering of the Octree

The first step of the algorithm is to organize the model in an octree. We do this in the same way as with the hierarchical z-buffer visibility algorithm, as described in §3.3.1. After the model has been organized into an octree, our antialiased rendering algorithm operates in two passes. In the first pass, the *tiling pass*, we traverse the octree in front-to-back order, culling sub-octrees that are completely hidden and inserting polygons associated

with potentially visible octree cubes into the image-space quadtree. In the second pass, the *refinement pass*, we compute minimum and maximum color bounds for each quadtree cell. Anywhere the color uncertainty at the pixel level is above the user-specified error tolerance, we subdivide in breadth-first order until the uncertainty drops below the error tolerance. It should be noted these two passes are generally analogous to the tiling and rendering passes of ray casting with a ZZ-buffer [Salesin-Stolfi89], which was reviewed in §2.1.4. Except for the fact that it employs a *uniform* subdivision of image space, the ZZ-buffer's tiling pass is similar to ours, using Warnock subdivision to organize potentially visible primitives [Warnock69]. However, the second passes of the two algorithms are entirely different, since rendering is performed by ray casting in one case and recursive subdivision in the other.

5.4.2 Tiling Pass

As with the hierarchical z-buffer visibility algorithm of chapter 3, we cull hidden cubes before processing the polygons they contain by applying the following observation. Assuming that the viewpoint is outside a cube, if the front faces of the cube are completely hidden, then all of the geometry contained in the cube is hidden and can be ignored. To use this test effectively, we traverse the octree in front-to-back order, because this guarantees that any geometry that can occlude a cube is represented in the quadtree before the cube is considered. Combining front-to-back traversal and cube culling, we have the following overall control structure for the tiling pass:

```
Tile(OctreeNode N)
{
   if CouldCubeBeVisible(N) returns FALSE
   then return
   Q = smallest quadtree cell enclosing N
   for each polygon P on N's list
      InsertPolyIntoQuadtree(P, Q)
   for each child C of N in front-to-back order
      Tile(C)
}
```

```
CouldCubeBeVisible(OctreeNode N)
{
   if N is completely outside viewing frustum /* method: see appendix */
   then return FALSE
   if viewpoint is inside N
   then return TRUE
   for each front face F of N {
      Q = smallest quadtree cell enclosing F
      if CouldCubeFaceBeVisible(F, Q) returns TRUE
      then return TRUE
   }
   return FALSE
}
```

The test CouldCubeFaceBeVisible returns FALSE if it can prove that a cube face is hidden, and TRUE otherwise. With z-buffer rendering, a cube face (or in fact any polygon) is hidden if its depth at each pixel sample is farther than the value stored in the z-buffer. To cull hidden z-buffered cube faces (and model polygons) efficiently, the hierarchical visibility method uses a pyramid of depth values which can often show with a single depth comparison that all the z-buffer values in a region enclosing a cube face are nearer than the face. With antialiased rendering, however, a cube face is hidden only if every point on the face is behind the corresponding point on a model polygon. To establish visibility of faces in this case, we generalize the depth pyramid of the hierarchical visibility method to a quadtree data structure containing a value $zfar$ at each cell that is equal to the depth of the farthest visible point within the quadtree cell and a value $znear$ that is equal to the depth of the nearest visible point within the quadtree cell. If the nearest point on a face within the cell is farther than $zfar$ of an enclosing quadtree cell, then the face is hidden within the cell. Likewise, if the nearest point on a face within the cell is nearer than $znear$ of an enclosing quadtree cell, the face is at least partially visible within the cell. If neither of these two conditions holds, we can obtain a more definitive result by applying the same visibility tests recursively within children of the quadtree cell.

In some cases the amount of computation needed to resolve the visibility of a cube can exceed the cost of rendering the polygons it contains, so we use a computation limit in testing cube faces for visibility. If the number of quadtree cells we visit while testing a cube face for visibility exceeds some constant times the number of polygons within the cell, we stop testing and assume that the cube face is visible. The following pseudocode outlines a visibility test for cube faces that visits quadtree cells in breath-first order.

```
CouldCubeFaceBeVisible(CubeFace F, QuadtreeCell Q)
{
   NumberVisited = 0
   create queue of quadtree cells containing only Q
   while queue is not empty {
      NumberVisited = NumberVisited + 1
      if NumberVisited > ComputationLimit
      then return TRUE
      C = Head(queue)
      remove C from queue
      if F intersects C /* method: see appendix, section A.3 */
      then {
         if nearest point on F within C is nearer than C.znear
         then return TRUE
         if nearest point on F within C is nearer than or equal to C.zfar
         then {
            if C is a leaf cell
            then SubdivideQCell(C)
            add C's children to end of queue
         }
      }
   }
   return FALSE
}

SubdivideQCell(QuadtreeCell Q)
{
   create child for each quadrant of Q
   for each child C of Q {
      add each polygon on Q's list that intersects C to C's polygon list
      find any covering structures formed by polygons on C's list
      if C is covered by polygons on its list
      then {
         find C.zfar
         RemoveHiddenPolys(C)
      }
      else C.zfar = z of far clipping plane
      find C.znear
      PropagateZBounds(C)
   }
   delete Q's polygon list
}
```

```
PropagateZBounds(QuadtreeCell Q)
{
   if Q is the root cell
   then return
   P = Parent(Q)
   P.znear = nearest znear of Q and its siblings
   P.zfar = farthest zfar of Q and its siblings
   PropagateZBounds(P)
}
```

When the above visibility test fails to prove that a cube is hidden, we insert any polygons associated with the cube into the quadtree. First we test each polygon for visibility using $znear$ and $zfar$ comparisons, as we did when testing cube faces for visibility. If the test fails to show that a polygon is hidden, we insert the polygon into the quadtree cell and update $zfar$ if the depth of the farthest visible point within the cell has changed. This only happens if the new polygon, either by itself or in combination with polygons already stored in the cell, forms a geometric structure that completely covers the cell. Easily detected covering structures include (a) a single polygon, (b) two polygons meeting along an edge and (c) a set of polygons meeting at a single vertex. If the number of polygons in a cell is large, testing for coverage becomes expensive, so we subdivide the quadtree cell whenever the number of polygons on its list exceeds a threshold (ten in our current implementation). The following pseudocode outlines this portion of the algorithm.

```
InsertPolyIntoQuadtree(Polygon P, QuadtreeCell Q)
{
   if P does not intersect Q
   then return
   Pznear = z of nearest point on P within Q
   if Pznear is farther than Q.zfar
   then return
   if P covers Q or forms a covering structure with other polygons on Q's list
   then {
      if zfar of covering structure is nearer than Q.zfar
      then {
         Q.zfar = zfar of covering structure
         PropagateZBounds(Q)
         if Q is a leaf cell
         then RemoveHiddenPolys(Q)
         else for each child C of Q {
            if C.znear is farther than Q.zfar
            then delete C and its subtree
         }
      }
   }
   InsertPoly(P, Q)
}
```

```
InsertPoly(Polygon P, QuadtreeCell Q)
{
   if Q is a leaf cell and its polygon list is not full
   then {
      add P to Q's polygon list
      Pznear = z of nearest point on P within Q
      if Pznear is nearer than Q.znear
      then {
         Q.znear = Pznear
         PropagateZBounds(Q)
      }
      return
   }
   if Q is a leaf cell
   then SubdivideQCell(Q)
   for each child cell C of Q
      InsertPolyIntoQuadtree(P, C)
}

RemoveHiddenPolys(QuadtreeCell Q)
{
   for each polygon P on Q's list {
      Pznear = z of nearest point on P within Q
      if Pznear is farther than Q.zfar
      then remove P from Q's list
      if P is behind all planes of a covering structure of Q
      then remove P from Q's list
   }
}
```

The tiling pass traverses all cubes of the octree that it is unable to prove are hidden. When it finishes, it has inserted all potentially visible polygons into the quadtree. The tiling pass never visits an octree node more than once, and for densely occluded scenes, it typically visits only a small subset of all octree nodes.

5.4.3 Refinement Pass

The refinement pass begins after the tiling pass has culled most of the hidden polygons and inserted the remaining polygons into the quadtree. Its task is to evaluate the filtered color $f(x, y) * I(x, y)$ of each output pixel. If the geometry and shading in a quadtree cell are simple enough, the algorithm may be able to compute the required convolution integral exactly or bound it tightly enough to meet the error tolerance. If the geometry or shading is too complex, the quadtree cell is subdivided as in Warnock's algorithm. New color bounds are computed for the children and propagated to coarser levels of the quadtree. Subdivision continues recursively in breadth-first order until the error tolerance is met at the pixel

level. The algorithm converges as long as repeated subdivision ultimately improves the error bounds.

At the beginning of the refinement pass, we subdivide the quadtree if necessary so that all leaf cells are no larger than a single pixel. For the purposes of refinement, we associate with each quadtree cell a minimum and maximum bound on each color component of the portion of the convolution integral $f(x,y) * I(x,y)$ within the cell. If the point-spread function $f(x,y)$ is a pixel-sized box filter, then $f(x,y) * I(x,y)$ is just the average value of $I(x,y)$ in each pixel and the filtering is known as area sampling [Catmull78]. In this case, the refinement pass is relatively simple and is described by the following pseudocode in which the uncertainty of each quadtree cell is difference between min and max bounds of the portion of the convolution integral within the cell.

```
Refine()
{
   place all quadtree leaf cells in a priority queue sorted by uncertainty
   for each cell Q on queue
      PropagateUncertainty(Parent(Q))
   while queue is not empty {
      Q = Head(queue)  /* greatest uncertainty */
      remove Q from queue
      SubdivideQCell(Q)
      for each child C of Q {
         compute C.Uncertainty
         add C to queue
      }
      PropagateUncertainty(Q)
   }
}

PropagateUncertainty(QuadtreeCell Q)
{
   Q.Uncertainty = sum of uncertainty of Q's children
   if Q is sub-pixel
   then PropagateUncertainty(Parent(Q))
   else {
      if Q.Uncertainty < ErrorTolerance
      then {
         /* pixel within error tolerance */
         set color to mean of color interval
         display pixel or write to image buffer
         remove Q and its subtree from the priority queue
      }
   }
}
```

If the point-spread function $f(x,y)$ of the filter extends beyond the boundaries of a pixel, then the algorithm is more complicated. In general, if $f(x,y)$ is zero beyond some fixed

radius, there will be a small maximum number of pixels k that can be affected by the value of $I(x, y)$ in a pixel or sub-pixel region. In this case, with each quadtree cell, we store up to k min and max color bounds corresponding to the portion of the convolution integral $f(x, y) * I(x, y)$ lying inside the quadtree cell for each of the affected pixels. Whenever we subdivide a quadtree cell, for each of the affected pixels, we compute new bounds on $f(x, y) * I(x, y)$ within the children and propagate the changes to the affected pixel-sized quadtree cells. If the point-spread function $f(x, y)$ has negative lobes, we can improve the bounds on $f(x, y) * I(x, y)$ by breaking f into the sum of a positive part $f^+(x, y)$ and a negative part $f^-(x, y)$. Then we can separately bound $f^+(x, y)*I(x, y)$ and $f^-(x, y)*I(x, y)$ and combine them to bound $f(x, y) * I(x, y)$. The control structure for choosing which cell to subdivide next can be the same here as outlined above for area sampling.

5.4.4 Interval Analysis

In order to compute each pixel to within the error tolerance with guaranteed accuracy, the refinement pass of the algorithm must be able to integrate $f(x, y) * I(x, y)$ analytically within a quadtree cell or prove sufficiently tight bounds on the integral. For flat shading, Gouraud shading, and other polynomial shading functions, the integral can be computed analytically with simple filters. For more general shaders, however, analytic integration may not be possible, so we rely on interval analysis [Moore66, Moore79, Alefeld-Herzberger83, Snyder92] to bound the integral. An interval shader receives bounds on input parameters such as u, v coordinates, components of the surface normal, the vector to the eye, etc., and uses these bounds to compute a bound on the output color. Once bounds on color within a quadtree cell have been established, they need to be combined with bounds on $f(x, y)$, the point-spread function of the filter. For efficiency, we can precompute a table of intervals for $f(x, y)$ for subdivisions of a canonical pixel down to some fine resolution. We can then use interval multiplication to find bounds on the required convolution $f(x, y) * I(x, y)$.

In combination with the control structure of §5.4.3, interval shaders drive quadtree subdivision at places like specular highlights because, for this example, the bounds are loose if a polygon contains a highlight and much tighter otherwise. Note that some care is required when constructing interval shaders for use with the algorithm. To guarantee convergence, all that is needed is that the intervals get smaller with subdivision. To achieve rapid convergence, however, the intervals must give reasonably tight bounds on the integral.

It should be noted that Kass has previously constructed interval shaders in a dataflow environment that are able to report bounds on the output color [Kass92], but this idea was not combined with subdivision of image space to create error-bounded images.

5.5 Implementation and Results

We have implemented the algorithm described in §5.4 using the area-sampling technique described in §5.4.3. To avoid using excessive amounts of memory, we have found it necessary to render complex images in subwindows. To demonstrate the progressive-refinement possibilities of the algorithm, our implementation differs from the pseudocode of §5.4.3 in that it does not subdivide down to the pixel level before beginning the refinement pass, and instead of using the priority queue described in §5.4.3, it refines the quadtree one level at a time, doing all subdivision at level k before doing any subdivision at level $k+1$.

The current list of cases that it integrates exactly is as follows. If a single flat-shaded polygon covers a quadtree cell, the implementation computes the integral as the product of the polygon color and the area of the quadtree cell. If two flat-shaded polygons cover a quadtree cell and either (a) one of the polygons is completely in front of the other or (b) the two polygons meet on an edge, the implementation computes the area of each polygon within the cell, multiplies each area by the corresponding color, and sums the results to compute the integral. Additional integration cases would reduce the need for subdivision and improve overall efficiency.

Figure 5.2 shows the 167-million polygon model of the Empire State Building rendered with our error-bounded rendering algorithm (right) and with the hierarchical z-buffer method (left). At the lower left corner of each image, a small region of the scene has been magnified with pixel replication to show detail. Note that the z-buffered image has bright pixels that appear to be randomly scattered along the columns of windows. This severe aliasing artifact is absent from the error-bounded antialiased image on the right. Our unoptimized implementation rendered this 340×512 antialiased image to within an error tolerance of .05 in approximately one hour on a 50Mhz R4000 SGI Crimson.

Figure 5.3 shows a flat-shaded interior view of the Empire State Building model. At top right, we show the result of antialiased rendering within an error tolerance of .05, and at top left we again show an aliased comparison image rendered using the hierarchical z-buffer method. The bottom-center panel of figure 5.3 shows the quadtree subdivision produced in rendering the antialiased image. Note that subdivision is coarse in regions covered by large polygons and becomes very fine along visible edges and in regions of fine geometric detail. The bottom-right panel of figure 5.3 shows quadtree subdivision in a different way. In this image, intensity encodes the log of the number of quadtree cells created at each pixel. Bright regions indicate where the algorithm is working hardest, for example, at the end of the corridor where pixels contain numerous visible polygons. The bottom-left panel of figure 5.3 shows the uncertainty of the antialiased image at each pixel, scaled so that the

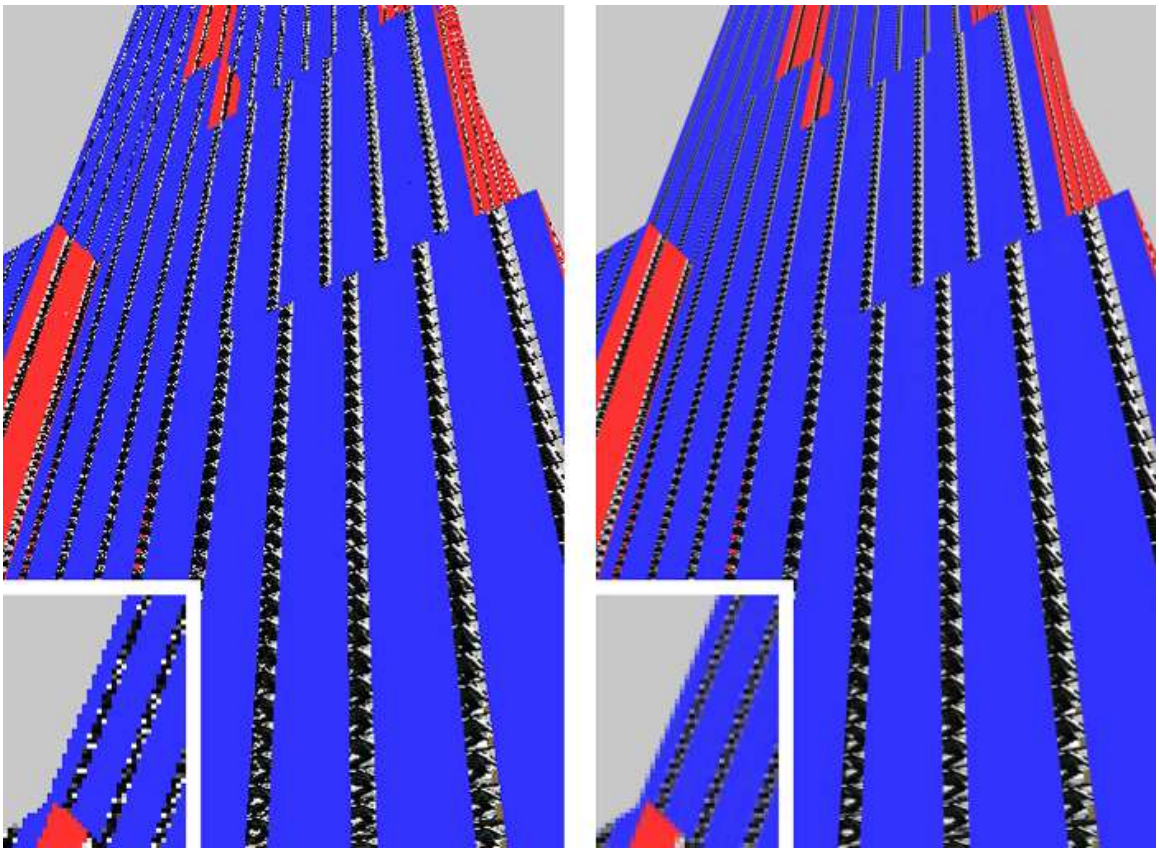


Figure 5.2 A model of the Empire State Building consisting of 167 million polygons rendered with z-buffering (left) and with the error-bounded antialiased rendering algorithm (right). Each box-filtered pixel of the antialiased image is accurate within an error tolerance of .05. The small images in the lower left corners of the panels show a small region of the scene that has been magnified with pixel replication to show detail.

maximum allowable error of .05 is represented as white. Most pixels are black, indicating that the algorithm was able to determine an exact color by integration. Some of the bright lines in the foreground are caused by slight gaps in the model; others are due to simple cases that our implementation does not yet integrate. Our unoptimized implementation rendered this 512×512 antialiased image in approximately 5 minutes on a 50Mhz R4000 SGI Crimson.

Figure 5.4 illustrates the progressive-refinement nature of the algorithm. The images from left to right show the regions of the image known to be within the error tolerance after four consecutive refinement passes. Pixels whose uncertainty exceed the error tolerance are shown in green. As the algorithm progresses, it produces an increasingly accurate image, computing more and more pixels to within the error tolerance. If it were necessary to

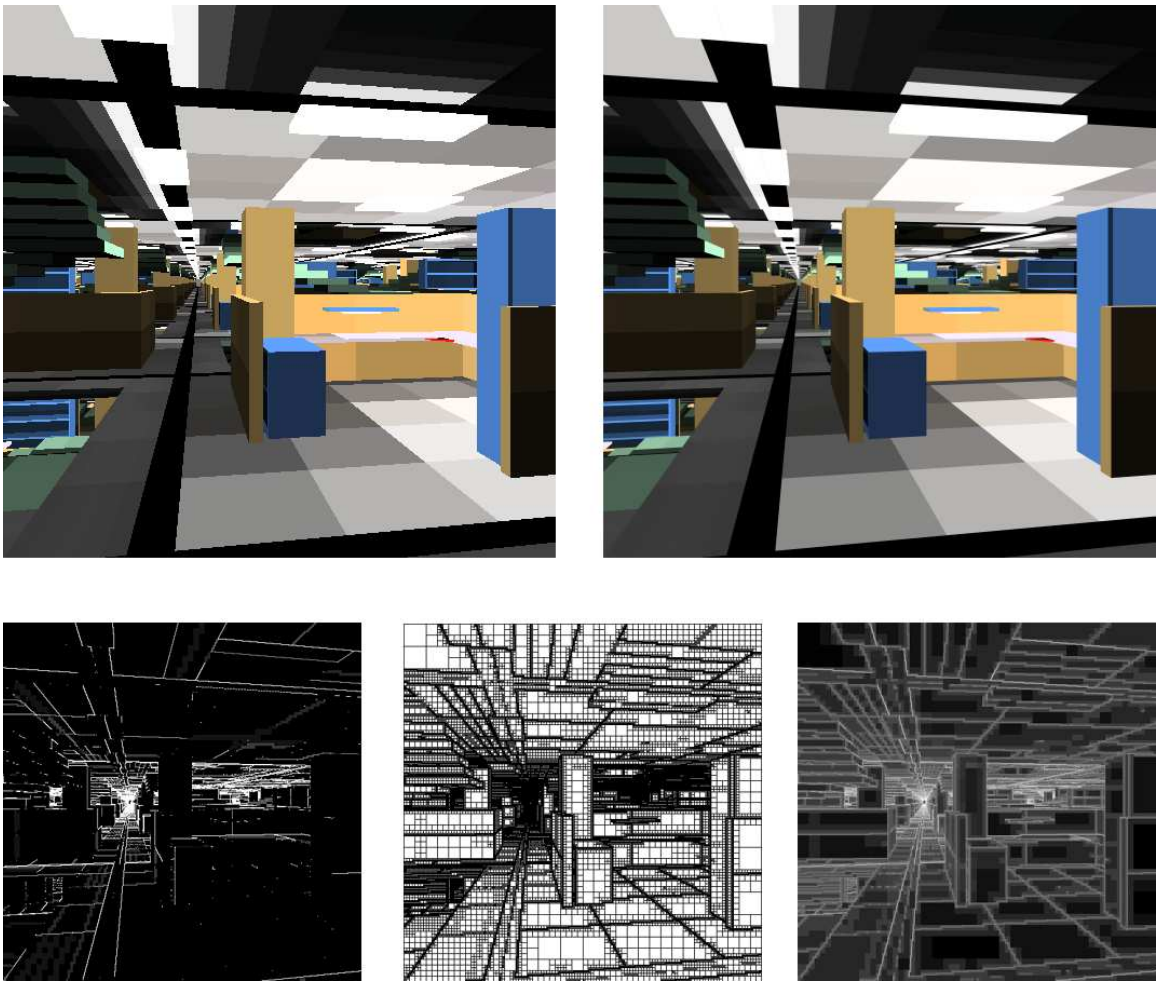


Figure 5.3 An interior view of the Empire State Building model. Top tier: Flat-shaded z-buffer image (left) and corresponding error-bounded antialiased image (right). Bottom tier: Uncertainty image (left), quadtree subdivision (center), and log-scale subdivision image (right). The uncertainty image is scaled so that white corresponds to the error tolerance of .05.

stop rendering at some point due to a limit on frame time, the remaining pixels could be interpolated quickly from their neighbors.

Our model of the Empire State Building is composed of flat-shaded polygons, so in figures 5.2 and 5.3 we were able to analytically integrate most of the polygonal fragments in leaf cells of the quadtree to obtain an exact result. For more complex shaders, we rely on interval analysis as described in §5.4.4. Figure 5.5 shows an example in which an interval shader projects a checkerboard texture map and a cycloidal bump map onto a polygon.

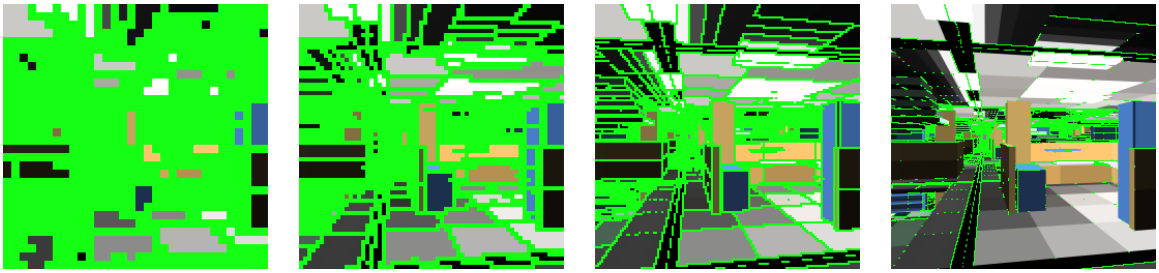


Figure 5.4 Regions known to be within the error tolerance after four consecutive refinement passes for the scene of figure 5.3. Green indicates pixels with color uncertainty above the error tolerance.

The bumps refract the texture map and reflect both Phong highlights and a "sky ramp" reflection map indexed by elevation. Each box-filtered pixel in this 512×512 antialiased image is accurate to within an error tolerance of .025. The left panel of figure 5.5 is a subdivision image in which, as before, intensity encodes the log of the number of quadtree cells created at each pixel. As expected, this image indicates that the algorithm works hardest in high-frequency regions of the output image and does comparatively little work in low-frequency regions.

5.6 Further Directions

As mentioned in §5.1, traversing polygons in strict front-to-back order would simplify and accelerate the algorithm because this strategy eliminates the need for depth comparisons, improves culling efficiency, and prevents unnecessary subdivision of the image-space quadtree. Front-to-back traversal is straightforward if, instead of organizing polygons in an ordinary octree, we organize them in an octree of BSP trees constructed as described in §4.6. Then, front-to-back traversal requires changing a single step in subroutine `Tile()`. We simply change `"for each polygon P on N's list"` to `"for each polygon P in N's BSP tree, in front-to-back order."` This single modification would make the algorithm run substantially faster. Of course numerous other changes should also be made to take full advantage of the fact that the depth order of polygons is known. Modified to exploit front-to-back traversal, the algorithm would run faster and use less memory, and it would be easier to implement.

In principle, it should be possible to extend the algorithm we have presented to do temporal as well as spatial antialiasing (i.e. motion blur). To do this, it would be necessary to add a time dimension to both the object-space octree and the image-space quadtree.

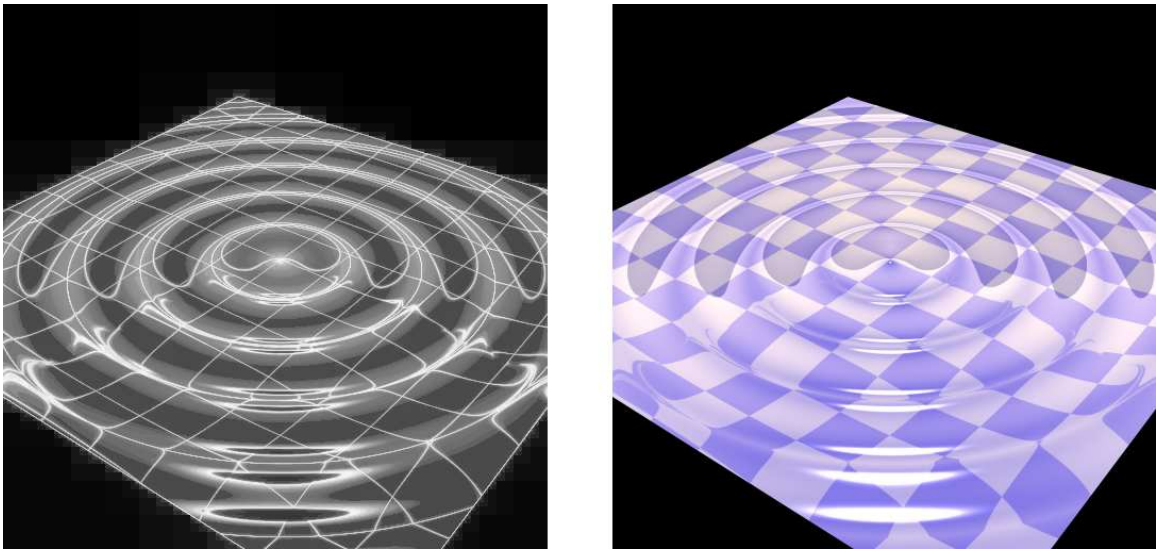


Figure 5.5 Antialiased image of a single polygon (right) rendered with an interval shader using texture mapping, bump mapping, reflection mapping, and Phong highlights. Each box-filtered pixel is accurate to within an error tolerance of .025. In the image on the left, intensity encodes the log of the number of quadtree cells created for each pixel. The algorithm works hardest in high-frequency regions of the output image and does comparatively little work in low-frequency regions.

Nodes in the new object-space tree would represent cubes of space for intervals of time. Nodes in the new image-space tree would represent regions of screen-space for intervals of time. For visibility purposes, each polygon and cube face would be treated as a space-time polyhedron. Culling would be done in space-time; a polygon would be culled from an image-space cell if no part of its space-time polyhedron was visible in the cell during any of the time interval. Subdivision to meet the error tolerance could take place either in space or time as appropriate. While the details of such an approach remain to be worked out, we are optimistic that it is possible to build a practical and guaranteed spatio-temporal antialiasing algorithm with this approach.

5.7 Conclusion

We have presented an antialiased rendering algorithm capable of rendering models of enormous complexity while producing results that are guaranteed to be correctly filtered to within a user-specified error tolerance. We have implemented the algorithm, shown that it is practical, and used it create accurately filtered images of scenes containing over one hundred million polygons. In addition, we have shown that interval analysis can be used to

guarantee that a wide class of shading functions is accurately filtered.

Chapter 6

Conclusion

6.1 Overview

As more and more complex models become commonplace in computer graphics, it becomes increasingly important to devise more efficient visibility algorithms. Ideally, the amount of work expended on visibility operations during image generation should correspond to visible complexity rather than scene complexity. We have presented three related algorithms that approach this goal by applying the same basic hierarchical methods to exploit coherence in both object space and image space and find visible geometry by logarithmic search. We also apply hierarchical methods to accelerate tiling and filtering. We have tested the algorithms on densely occluded scenes having very complex occlusion relationships and shown that they work very effectively, spending most of their time processing visible or nearly visible geometry.[1]

A basic shortcoming of traditional visibility algorithms for scenes represented as geometric models is that they do not fully exploit both object-space and image-space coherence. Generally speaking, methods which effectively exploit object-space coherence, such as ray casting through an octree, are not amenable to exploiting image-space coherence, and methods which effectively exploit image-space coherence, such as z-buffer scan conversion, are not amenable to exploiting object-space coherence. Although visibility algorithms have been devised which exploit both of these forms of coherence, all have important limitations. Of such prior algorithms, *potentially visible set* methods have been most successfully applied to rendering densely occluded scenes at rapid frame rates. However, this approach requires partitioning the model into mostly enclosed, disjoint cells, which can be difficult to automate, and it has only been demonstrated to work effectively for architectural environments. Our approach is simpler and more general, since the model can be organized with a simple, automatic procedure, and there are no geometric restrictions on the model.

The three algorithms that we have presented, *hierarchical z-buffer visibility*, *hierarchical polygon tiling* (with object-space culling), and *error-bounded antialiased rendering*, all share the same basic structure designed to enable very efficient visibility operations. We refer to

[1] In rendering a frame, only primitives contained in visible or nearly visible bounding volumes of the object-space hierarchy are even traversed.

this common structure as the *basic hierarchical visibility algorithm* which may be summarized as follows. We maintain hierarchical data structures in both object space and image space which facilitate hierarchical culling of hidden geometry. To enable hierarchical culling in object space, we organize scene geometry in an object-space octree. Using a simple recursive subdivision procedure, we traverse octree nodes in front-to-back order, testing them for visibility, culling nodes that are hidden and rendering the geometry contained in visible nodes. This object-space culling procedure is very efficient because it traverses only visible or nearly visible octree nodes and their children, and only primitives contained in visible or nearly visible octree nodes need to be rendered[2]. Thus, it very effectively culls most hidden geometry in densely occluded scenes. However, some geometry will still overlap on the screen when primitives are projected. To cull remaining hidden geometry we perform Warnock-style hierarchical culling in image space [Warnock69], using an image-space hierarchy to maintain visibility information about previously rendered geometry. Since the recursive subdivision procedures of this basic algorithm find visible geometry by logarithmic search, the algorithm is very efficient and approaches our goal of doing work proportional to visible complexity rather than scene complexity. It should be noted that our use of object-space and image-space hierarchies is very similar to Meagher's [Meagher82b], even though his methods are only directly applicable to volume models rather than geometric models.

As mentioned in §3.6, underlying the basic algorithm is a very simple idea: we organize scene geometry in bounding boxes, and before doing any work on the geometry inside a box, we first verify that the box itself is visible by tiling its faces.[3] Maintaining information about previously rendered geometry in an image-space hierarchy enables us to perform this tiling hierarchically, and if bounding boxes are nested in an object-space hierarchy, culling of hidden geometry can be performed hierarchically in both object space and image space. Conversely, we find visible geometry by logarithmic search in both object space and image space. This simple strategy encapsulates the basic algorithm.

The main limitation of the basic algorithm is its performance in processing dynamic scenes. When some scene primitives are in motion, the object-space hierarchy must be maintained during frame generation. If the scene is predominantly static, the frame-to-

[2] In the case of hierarchical z-buffer visibility, only visible octree nodes and their children are traversed and only primitives contained in visible nodes are rendered.

[3] As mentioned in §3.6, this idea has undoubtedly occurred to many practitioners, but I first heard it articulated by Lance Williams in the early 1980's when we worked together at the NYIT Computer Graphics Lab. When I moved to Apple in 1989, I was reminded of the strategy by Gavin Miller, who had used this method to accelerate z-buffering.

frame cost of updating the hierarchy is relatively modest and may not seriously impair performance. But if a great many primitives are in motion, the cost of maintaining the hierarchy may be high, perhaps even dominating overall computation. In §3.4 we suggest various methods for exploiting available coherence in dynamic scenes to reduce the cost of maintaining the hierarchy. However, it should be acknowledged that in extreme cases there may be little coherence to exploit, and if the time spent maintaining the hierarchy exceeds the time saved by object-space culling, our basic algorithm offers no advantage over traditional methods.

6.2 Historical Development of the Ideas

We originally developed the basic hierarchical visibility algorithm to accelerate z-buffer rendering of densely occluded scenes as described in chapter 3. As mentioned in the introduction, this algorithm was developed jointly with Michael Kass and Gavin Miller, colleagues at Apple Computer. With the hierarchical z-buffer algorithm, the image-space hierarchy is a pyramidal representation of a conventional z-buffer. Using this *z-pyramid*, culling a hidden cube or hidden primitive often requires only a single depth comparison. Aside from its value in the basic algorithm, a z-pyramid can also improve the performance of traditional z-buffering and z-buffer hardware accelerators. The hierarchical z-buffer algorithm effectively exploits both object-space and image-space coherence as described above. In addition, it can also exploit temporal coherence by using the octree nodes that were visible in the preceding frame of animation to construct a starting point for the algorithm. The hierarchical z-buffer algorithm appears to be the first visibility algorithm that materially profits from object-space, image-space, and temporal coherence simultaneously, resulting in very high performance. The algorithm has been tested and shown to work effectively on indoor and outdoor scenes having hundreds of millions of polygons. While the algorithm can make use of existing z-buffer hardware accelerators, modest changes in the design of accelerators designed to facilitate hierarchical visibility operations could improve performance dramatically.

While the hierarchical z-buffer algorithm has excellent performance, it suffers from the usual aliasing problems of point-sampling algorithms – jaggies and other disturbing visual artifacts. In exploring ways of adapting the basic algorithm to antialiased rendering, it occurred to us that interval methods could be employed to bound error. We observed that for a broad class of shading functions, arbitrarily accurate images could be generated by controlling adaptive subdivision of the image-space hierarchy with interval methods, doing as much work as necessary in regions where geometry or shading is complex to obtain

the desired degree of accuracy. As described in chapter 5, the resulting error-bounded antialiased rendering algorithm guarantees that each pixel of the output image is accurate within a user-specified error tolerance of the filtered underlying continuous image. We have demonstrated that this algorithm can create accurately filtered images of extremely complex scenes in which numerous primitives are visible within some pixels. We have also shown that interval analysis can be used to guarantee that a broad class of shading functions are accurately filtered. To the best of our knowledge, the images produced with this algorithm are the only computer-generated images of guaranteed accuracy that have ever been created of extremely complex scenes or scenes rendered with complex shaders. As mentioned in the introduction, this algorithm was developed jointly with Michael Kass.

Although the error-bounded antialiased rendering algorithm is of considerable theoretical interest, it is not very practical on contemporary computers. Interval methods are often slow to converge, so the algorithm requires a great deal more computation and memory than traditional oversampling and filtering methods for antialiasing. Currently, the algorithm's high cost can only be justified for applications that demand exceptionally high image quality. Consequently, we explored more practical means of adapting the basic algorithm to antialiased rendering. One obvious approach was to create oversampled images with the hierarchical z-buffer algorithm and then filter the subpixel samples. Although the performance of this algorithm would be reasonably good, tiling computations and the size of image memory would be expected to increase in proportion to the degree of oversampling, so the algorithm would be many times slower than the original point-sampling z-buffer algorithm.

In exploring alternatives to z-buffer tiling, we discovered a novel polygon tiling algorithm which we call *hierarchical polygon tiling* that is capable of producing antialiased images much more efficiently than hierarchical z-buffering, while requiring much less image memory. Hierarchical tiling traverses polygons in front-to-back order, tiling them into an image hierarchy using *triage coverage masks*. Triage masks permit Warnock-style subdivision of image space with its logarithmic search properties to be driven very efficiently by boolean mask operations. The resulting tiling procedure performs subdivision and visibility operations very rapidly while only visiting cells in the image hierarchy that are crossed or nearly crossed by visible edges in the output image. Visible samples are never overwritten. Hierarchical z-buffering, on the other hand, must visit every image sample covered by every visible polygon[4] and it frequently overwrites image samples. Moreover, when antialiasing is performed by oversampling, hierarchical tiling requires only a small fraction of the image

[4] Recall that our hierarchical z-buffer algorithm reverts to traditional z-buffer scan conversion in tiling visible polygons.

memory required by hierarchical z-buffering. Whereas z-buffering requires storing color and depth at each subpixel image sample, hierarchical tiling only requires storing slightly more than one bit of occupancy information for each sample. As a consequence of these advantages in tiling efficiency and memory usage, hierarchical tiling can generate high-quality antialiased images at roughly the same speed as hierarchical z-buffering generates the corresponding point-sampled images. The main drawback of hierarchical tiling as compared with hierarchical z-buffering is that polygons must be traversed in strict front-to-back order, which for dynamic scenes necessitates additional sorting computations.

6.3 Conclusion

We have presented three algorithms for rendering complex, densely occluded scenes that all share the same basic structure. Their ability to find visible geometry by logarithmic search in both object space and image space enables them to process densely occluded scenes very efficiently, spending most of their time processing visible or nearly visible geometry. Two of the algorithms, *hierarchical z-buffer visibility* and *hierarchical polygon tiling*, have exceptional performance and are appropriate for interactive applications. The third algorithm, *error-bounded antialiased rendering*, is much slower, but is unique in its ability to produce antialiased images of guaranteed accuracy.

Of the three algorithms, hierarchical polygon tiling combined with the basic hierarchical visibility algorithm usually offers the best combination of speed and image quality. The hierarchical methods we advance for finding visible geometry by logarithmic search have their clearest expression in this algorithm. Only primitives contained in visible or nearly visible nodes in the object-space octree need to be tiled, and tiling only traverses cells in the image-space hierarchy that are crossed or nearly crossed by visible edges in the output image. In addition, the algorithm has very modest memory requirements, it never overwrites image samples, it exploits precomputation by presorting scene primitives and precomputing tiling patterns, and it exploits the parallelism that is inherent in boolean mask operations in visibility, tiling, and filtering operations. These properties give the algorithm exceptional performance in rendering predominantly static polygonal environments with high-quality antialiasing. Quite aside from the processing of densely occluded scenes, hierarchical polygon tiling is an attractive alternative to z-buffering, even when scene geometry is very simple. When generating antialiased images of predominantly static scenes, hierarchical tiling offers a substantial advantage over z-buffering in both speed and memory usage. Aside from the role they play in this algorithm, triage coverage masks can accelerate a wide variety of algorithms that employ Warnock-style subdivision by reducing the computational

overhead of performing subdivision.

The hierarchical z-buffer visibility algorithm also has very good performance, and for some applications offers advantages over hierarchical tiling. Since the algorithm maintains a conventional z-buffer, primitives can tiled in any order, which simplifies processing of dynamic scenes, and results can be composited with images produced with z-buffer hardware and other z-buffer renderers. Our temporal-coherence scheme exploits this compatibility. The algorithm is also well suited to implementation in hardware. In fact, we believe that the performance of z-buffer hardware accelerators could be improved dramatically by incorporating the algorithm's z-pyramid and object-space culling capabilities. We hope that the appeal of this algorithm will induce hardware designers to adapt future graphics hardware to facilitate hierarchical visibility operations.

Although the error-bounded antialiased rendering algorithm is not nearly as fast as the other two algorithms, its ability to render with guaranteed accuracy is an important advantage for applications that demand very high image quality, such as animation for entertainment. The cost of automating the production of high-quality animation with this algorithm is currently high, but the practicality of this approach will improve over time as the cost of computation diminishes.

In conclusion, very complex scenes pose difficult challenges to the performance and accuracy of rendering algorithms. We have presented a general-purpose hierarchical visibility algorithm which, in combination with novel hierarchical tiling and filtering methods, extends the capability of computers to animate complex environments, thereby enhancing the power and richness of the computer-graphics medium.

Appendix A

Detecting Intersection of a Rectangular Solid and a Convex Polyhedron

chaptermark

In this appendix we present a fast method for determining whether a rectangular solid intersects a convex polyhedron. Our implementations of the rendering algorithms presented in chapters 3, 4, and 5 use this method to determine whether an octree cube intersects the viewing frustum (see §3.3.1, §5.4.2). The method can also be applied to detecting the intersection of a rectangular solid and a 3D planar polygon. We use this variant of the algorithm to perform cube-polygon intersection tests during construction of an octree for a polygonal scene, as discussed in §3.3.1. This article appeared in slightly different form in *Graphics Gems IV* [Greene94].

Our algorithm is based on the observation that a rectangular solid R intersects a convex polyhedron P if and only if (a) the projections of R and P intersect in all three axis-aligned orthographic views (i.e. front, side, and top views), and (b) R does not lie entirely "outside" the plane of any face of P. After finding the equations of certain lines and planes associated with P, we can determine whether these criteria are satisfied by simply evaluating inequalities. Determining whether an octree cube intersects a viewing frustum requires evaluating between one and thirty inequalities.

A.1 Introduction

Beginning with some definitions, we will refer to a convex polyhedron simply as a polyhedron and to an axis-aligned rectangular solid simply as a box. The term "bounding box" will specifically refer to a "tight," axis-aligned rectangle in 2D or rectangular solid in 3D. Axis-aligned orthographic views, the familiar front, side, and top views of engineering drawings, will be referred to simply as orthographic views.

Various computer-graphics techniques associate bounding boxes in the shape of rectangular solids with clusters of geometric primitives to enhance efficiency, the idea being that performing a single operation on a bounding box often eliminates the need to process primitives inside the box individually. When culling a geometric model to a viewing frustum, for example, if a bounding box lies outside the frustum, the primitives it contains also lie

outside and need not be considered individually. This observation underlies simple, fast procedures for culling models represented in spatial hierarchies. For example, if geometric primitives are organized in an octree of bounding boxes, parts of the model that lie entirely outside a viewing frustum can be efficiently culled by testing for box-frustum intersection during recursive subdivision of the octree [Clark76, Garlick-et-al90].

Since box-polyhedron intersection tests may be performed hundreds or thousands of times in the course of producing a single image of a scene, devising an efficient intersection algorithm is worthwhile. One intuitive approach to the problem combines a test for intersecting bounding boxes, tests to see if a vertex of one solid lies inside the other, and tests to see if an edge of one solid intersects a face of the other. While this method is very straightforward, the expense of finding geometric intersections makes it quite costly. Alternatively, we could employ algorithms for detecting intersection of convex polyhedra that are described in the computational geometry literature, for example the method of Chazelle and Dobkin [Chazelle-Dobkin87]. However, such methods are typically complicated, designed with asymptotic performance in mind, and poorly suited to simple problems like deciding whether a box intersects a viewing frustum.

Our approach is both simple and efficient. It does not require finding geometric intersections, but rather, after finding the equations of certain lines and planes associated with a polyhedron, box-polyhedron intersection can be determined by simply comparing bounding boxes and evaluating inequalities. The algorithm performs most efficiently when comparing numerous bounding boxes to the same polyhedron, but even when performing a single intersection test, efficiency compares favorably to other methods.

We precede our analysis of box-polyhedron intersection with a discussion of primitive operations that our algorithm performs – determining whether a box intersects a plane, whether a rectangle intersects a line, and whether a rectangle intersects a polygon.

A.2 Box-Plane and Rectangle-Line Intersection

One of the fundamental operations performed by our box-polyhedron intersection algorithm is determining whether a box intersects a plane, and if not, which side of the plane the box lies on. The problem is illustrated in figure A.1, an orthographic projection in which the viewpoint has been chosen so that plane P is perpendicular to the screen. Vector N is normal to P, pointing into P's positive half-space.[1] Box B1 lies in P's negative half-space,

[1] Recall that a plane defined by equation Ax+By+Cz+D=0 has normal (A,B,C) and divides space into a positive half-space satisfying the equation Ax+By+Cz+D>0 and a negative half-space satisfying the equation Ax+By+Cz+D<0.

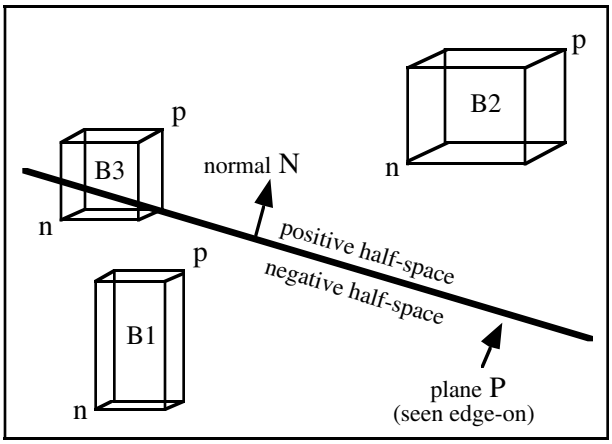


Figure A.1 P- and n-vertices of various boxes with respect to plane P.

box B2 lies in P's positive half-space, and box B3 intersects P. These are the three cases that we wish to distinguish in general.

The first step in distinguishing these cases is to establish which of a box's vertices lies farthest in the "positive direction" (the direction of normal N), call this the *p-vertex*, and which of the box's vertices lies farthest in the "negative direction," call this the *n-vertex*. P- and n-vertices are easily identified, since the p-vertex corresponds to the octant of the plane's normal,[2] and the n-vertex lies at the opposite corner. When an edge or face of the box is parallel to the plane, the octant rule does not specify a unique vertex, but in these cases it doesn't matter which of the obvious candidates is selected. In figure A.1, p-vertices are labeled p and n-vertices are labeled n. Note that the p- and n-vertices associated with a particular plane have the same relative positions on all axis-aligned bounding boxes, so they need to be identified only once.

Once p- and n-vertices have been identified, distinguishing the three cases of box-plane intersection is very simple. Box B lies entirely in plane P's negative half-space if and only if its p-vertex lies in P's negative half-space, B lies entirely in P's positive half-space if and only if its n-vertex lies in P's positive half-space, and if neither of these relationships holds, B intersects P [Haines-Wallace91]. It follows that we can determine whether a box lies in a particular half-space by evaluating one plane equation, and that the three cases of box-plane intersection can be distinguished by evaluating one or two plane equations as follows:

[2] For example, if the coefficients of the plane's normal are all positive, this corresponds to the $+x/+y/+z$ octant of the Cartesian axes and also the $+x/+y/+z$ vertex of an axis-aligned box.

```
Given an axis-aligned rectangular solid R with n-vertex (xn,yn,zn) and
p-vertex (xp,yp,zp), and plane P with equation Ax+By+Cz+D=0,

if (A*xp + B*yp + C*zp + D < 0) { R lies entirely in P's negative half-space }
else if (A*xn + B*yn + C*zn + D > 0) { R lies entirely in P's positive half-space }
else { R intersects P }
```

The two-dimensional problem of determining whether an axis-aligned rectangle intersects a line, lies entirely in the line's negative half-plane, or lies entirely in the line's positive half-plane is entirely analogous, requiring the evaluation of one or two line equations.

```
Given an axis-aligned rectangle R with n-vertex (xn,yn) and p-vertex (xp,yp),
and line L with equation Ax+By+C=0,

if (A*xp + B*yp + C < 0) { R lies entirely in L's negative half-plane }
else if (A*xn + B*yn + C > 0) { R lies entirely in L's positive half-plane }
else { R intersects L }
```

A.3 Rectangle-Polygon Intersection

We now consider another subproblem, determining whether an axis-aligned rectangle R and a convex polygon P, both lying in the same plane, intersect. As Meagher has observed [Meagher81], R intersects P if and only if (a) R intersects P's bounding box and (b) R does not lie entirely "outside"[3] any of the infinite lines defined by P's edges, which we will refer to as *edge-lines*. The problem is illustrated in figure A.2 where P's bounding box B is drawn in dashed lines and P's edge-lines are drawn in dotted lines (e.g. E). Applying the intersection criteria to rectangles R1, R2, and R3 of figure A.2, R1 is found not to intersect P because it does not intersect P's bounding box, R2 is found not to intersect P because it lies outside of edge-line E, and R3 is found to intersect P because it satisfies both intersection criteria.

The rectangle-line intersection method described in the preceding section is an efficient way to evaluate criterion b). Using this method we can determine whether a rectangle lies

[3] By "outside" we mean on the side opposite polygon P.

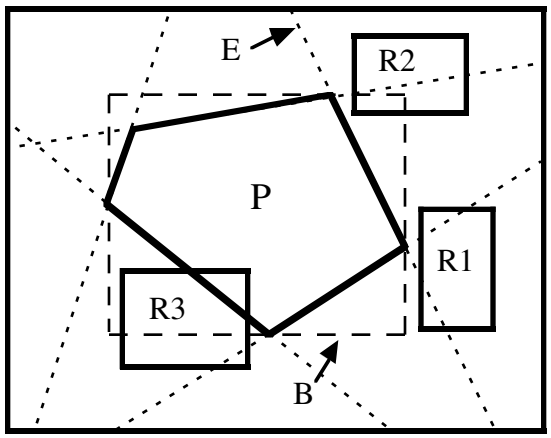


Figure A.2 Testing for intersection between a polygon P and various rectangles. A rectangle R intersects P if and only if (a) R intersects P's bounding box and (b) R does not lie entirely "outside" any of the infinite lines defined by P's edges.

outside an edge-line by substituting the coordinates of the rectangle's n-vertex into the edge-line's equation. For example, to show that R2 lies outside edge-line E we would substitute the coordinates of R2's lower left vertex (the n-vertex for E) into the line equation for E.

The rectangle-polygon intersection problem is germane because our algorithm looks for box-polyhedron intersection in orthographic views[4], and in these views a box always projects to an axis-aligned rectangle and the silhouette of a convex polyhedron is always a convex polygon. These conditions apply, for example, in figure A.3 where the panels labeled "front," "side," and "top" are orthographic views of a box and a polyhedron in the shape of a viewing frustum. Once the polygonal silhouette of the polyhedron has been identified in an orthographic view[5], determining whether box-polyhedron intersection occurs in that view reduces to the rectangle-polygon intersection problem described above. Incidentally, figure A.3 illustrates that intersection of a box and a polyhedron in all three orthographic views does not guarantee intersection in 3D, as is apparent in the upper right panel.

It follows from the rules for rectangle-polygon intersection that the projections of a box B and a polyhedron P intersect in all three orthographic views if and only if (1) B intersects P's bounding box and (2) the projection of B does not lie outside any of the edge-lines of P's silhouette in any of the three orthographic views. Our box-polyhedron intersection

[4] Remember that axis-aligned is implied in this term.

[5] Finding a convex polyhedron's silhouette edges is particularly straightforward in an orthographic view. In a view down an axis, call it the u-axis, if the u components of the outward-pointing normals of two adjacent faces have opposite signs, their common edge is a silhouette edge.

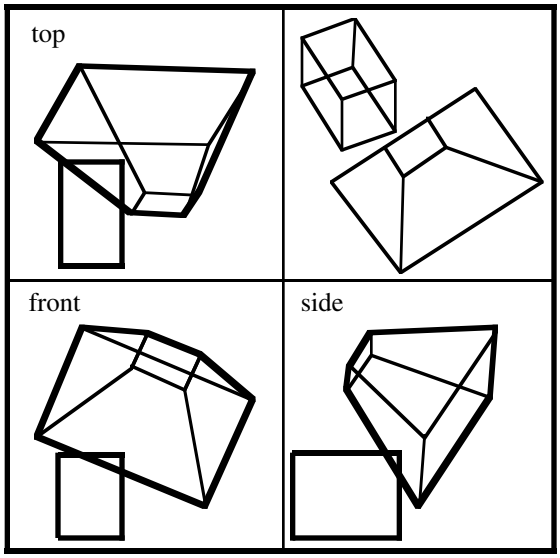


Figure A.3 Four views of a rectangular solid and a polyhedron in the shape of a viewing frustum. The front, side, and top views are axis-aligned orthographic views.

algorithm uses this formulation to establish intersection in orthographic views, as elaborated in the following section.

A.4 Box-Polyhedron Intersection

Our box-polyhedron intersection algorithm is based on the observation that for any two convex polyhedra A and B that do not intersect, there exists a separating plane lying between them that is (1) parallel to a face of A, (2) parallel to a face of B, or (3) parallel to an edge of A and an edge of B.[6]

Applying this observation to a rectangular solid R and a convex polyhedron P, we will say that a separating plane that is parallel to a face of R is of *type 1*, a separating plane that is parallel to a face of P is of *type 2*, and a separating plane that is parallel to an edge of R and an edge of P is of *type 3*. If a separating plane of type 1, 2, or 3 exists, R and P do not intersect; otherwise they do.

[6] My thanks to an anonymous reviewer for *Graphics Gems IV* for pointing this out and suggesting the ensuing line of exposition. As a starting point for a proof, consider two non-interpenetrating convex polyhedra whose surfaces are in contact, e.g., a vertex touches a face, an edge touches an edge, an edge touches a face, etc. For any possible configuration, a plane of type 1, 2, or 3 separates the polyhedra except for the point, line segment, or polygon of contact.

We can easily see whether a type 1 plane exists by determining whether P's bounding box intersects R. If so, a type 1 plane does not exist; if not, a type 1 plane does exist demonstrating that R and P do not intersect, and we are done.

We can see whether a type 2 plane exists by comparing each face-plane of P with the corresponding n-vertex of R using the method presented in the section on box-plane intersection. If any of these n-vertices is outside of its respective face-plane, there exists a type 2 separating plane demonstrating that R and P do not intersect and we are done.

The remaining problem is to determine whether a separating plane of type 3 exists. Since a type 3 plane is parallel to an edge of R, it projects to a "separating line" lying between the projections of R and P in one of the orthographic views. It follows that a type 3 plane exists if and only if the projections of R and P do not intersect in at least one of the three orthographic views. Combining this observation with intersection criterion (2) from the preceding section, the existence of a type 3 plane can be decided by determining whether the projection of R lies outside an edge-line of P in at least one orthographic view.[7] If so, a type 3 plane exists, establishing that R and P do not intersect; if not, a type 3 plane does not exist and we conclude that R and P intersect because no type 1, 2, or 3 separating plane exists.

Incidentally, the conditions for box-polyhedron intersection can be stated very concisely as follows: Box R intersects polyhedron P if and only if (a) the projections of R and P intersect in all three orthographic views, and (b) R does not lie entirely outside the plane of any face of P. In this formulation the requirement that R intersects P's bounding box is subsumed by condition (a). The analogous intersection conditions for a box R and a 3D planar polygon P are as follows: R and P intersect if and only if the projections of R and P intersect in all three orthographic views and R intersects the plane of P.

A.5 Summary of the Algorithm

Let's reiterate the algorithm as it is applied in practice with efficient execution in mind. Assuming for the moment that polyhedron P will be compared to numerous boxes, we begin with the following preliminary steps: 1) for each face of P we find the plane equation and identify which box corner is the n-vertex; 2) for each silhouette edge of P in the three orthographic views we find the line equation and identify which box corner is the n-vertex[8];

[7] Note that we have already established that R intersects P's bounding box, satisfying intersection criterion (1) from the preceding section.

[8] Pick either of the two vertices that coincide in the orthographic view.

and 3) we find P's bounding box. If P will be compared to only a small number of boxes, lazy evaluation of this information may be more efficient.

Now box-polyhedron intersection testing proceeds as follows. If box R does not intersect P's bounding box, we conclude that R does not intersect P and we are done. Next we consider P's face-planes one by one, seeing if R lies entirely outside any of these planes. If so, we conclude that R does not intersect P and we are done. Finally, we consider P's edge-lines[9] one by one, seeing if R's projection lies entirely outside any of these lines in the appropriate orthographic view. If so, we conclude that R does not intersect P and we are done. Otherwise, it has been established that intersection does occur.

To estimate the computational expense of the algorithm we examine the cost of the primitive operations. We can determine whether two 3D bounding boxes intersect by evaluating between one and six simple inequalities of the form "is a<b?" Determining whether a box lies entirely outside a face-plane requires evaluating one plane equation. Determining whether a box's projection lies entirely outside an edge-line requires evaluating one line equation. So when comparing a box to a polyhedron with F faces and E silhouette edges in orthographic views, each box-polyhedron intersection test requires evaluating between one and six simple inequalities, evaluating between zero and F plane equations, and evaluating between zero and E line equations. Summing up, between 1 and 6+E+F inequalities must be evaluated to show that B and P do not intersect, and all 6+E+F inequalities must be evaluated to show that B and P do intersect. For a viewing frustum, E is at most 18 and F is 6, so the maximum number of inequalities that must be evaluated to decide box-frustum intersection is 30. Our cost estimate should also amortize the cost of finding line and plane equations and identifying n-vertices over the number of intersection tests performed.

There are many variations on this basic algorithm. To avoid evaluating all 6+E+F inequalities whenever intersection occurs, a test can be added to see if the polyhedron lies entirely inside the box or vice versa. When culling geometry to a viewing frustum it may be useful to know which of the frustum's face-planes a box intersects, because they correspond to clipping planes. These refinements are included in the procedure outlined in the following section which we use to cull an octree to a viewing frustum.

---

[9] That is, the lines defined by P's silhouette edges in the three orthographic views.

A.6 Pseudo-Code

Given a collection of axis-aligned rectangular solids and a convex polyhedron P, the procedure of LISTING A.1 classifies each rectangular solid R as entirely outside, entirely inside, or partially inside P, and in the latter case reports which face-planes of P are intersected by R. The procedure can be streamlined for applications that only need to detect intersection.

```
LISTING A.1   (PSEUDOCODE)

/* Routine for Detecting Box-Polyhedron Intersection */

/* Preliminary Step */
Determine P's bounding box.
Find the plane equation of each face of P (these are face-planes).
Determine which vertex of an axis-aligned rectangular solid is the n-vertex and
   which is the p-vertex for each face-plane of P.
Determine the silhouette edges of P's projection in the three orthographic views,
   and find the line equation of each of these edges (these are edge-lines).
Determine which vertex of an axis-aligned rectangular solid is the n-vertex for
   each edge-line of P.

For each rectangular solid R {
   /* Bounding Box Tests */
   if R does not intersect P's bounding box, R is entirely outside P, done
      with R
   if P's bounding box is entirely inside R, R is partially inside P and R
      intersects all face-planes, done with R

   /* Face-Plane Tests */
   for each face-plane F {
      if R is entirely outside F, R is entirely outside P, done with R
      if R is entirely inside F, set a flag indicating that R does not intersect F
      else R intersects F, set a flag indicating that R intersects F
   }
   if R was entirely inside all face-planes, R is entirely inside P, done with R

   /* Edge-Line Tests */
   for each edge-line E
      if the projection of R is entirely outside E (in the appropriate
         orthographic projection), R is entirely outside P, done with R

   R is partially inside P and it is known which face-planes R intersects
}
```

References

[Abram-et-al85] G. Abram, L. Westover, and T. Whitted, "Efficient Alias-Free Rendering using Bit-Masks and Look-Up Tables," *Proceedings of SIGGRAPH '85*, July 1985, 53–59.

[Aho-et-al83] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[Airey90] J. M. Airey, "Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations," PhD Thesis, Technical Report TR90-027, Computer Science Dept., U.N.C. Chapel Hill, 1990.

[Airey-Rohlf-Brooks90] J. M. Airey, J. H. Rohlf, and F. P. Brooks Jr., "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.

[Alefeld-Herzberger83] G. Alefeld and J. Herzberger, *Introduction to Interval Computations*, Academic Press, 1983.

[Appel68] A. Appel, "Some Techniques for Shading Machine Renderings of Solids," *AFIPS Conference Proceedings*, vol. 32, 1968, 37–45.

[Badt88] S. Badt Jr., "Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing," *The Visual Computer*, 4(3), Sept. 1988, 123–132.

[Burt-Adelson83] P. J. Burt and E. H. Adelson, "A Multiresolution Spline with Applications to Image Mosaics," *ACM Transactions on Graphics*, 2(4), Oct. 1983, 217–236.

[Bloomenthal83] J. Bloomenthal, "Edge Inference with Applications to Antialiasing," *Proceedings of SIGGRAPH '83*, July 1983, 157–162.

[Bresenham65] J. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal*, 4(1), 1965, 25–30.

[Carpenter84] L. Carpenter, "The A-buffer, an Antialiased Hidden Surface Method," *Proceedings of SIGGRAPH '84*, July 1984, 103–108.

[Catmull74] E. Catmull, "A Subdivision Algorithm for Computer Display of Curved Surfaces," PhD Thesis, Report UTEC-CSc-74-133, Computer Science Dept., University of Utah, Salt Lake City, Utah, Dec. 1974.

[Catmull78] E. Catmull, "A Hidden-Surface Algorithm with Anti-Aliasing," *Proceedings of SIGGRAPH '78*, Aug. 1978, 6–11.

[Catmull84] E. Catmull, "An Analytic Visible Surface Algorithm for Independent Pixel Processing," *Proceedings of SIGGRAPH '84*, July 1984, 109–115.

[Chapman-et-al91] J. Chapman, T. W. Calvert, and J. Dill, "Spatio-Temporal Coherence in Ray Tracing," *Proceedings of Graphics Interface '91*, June 1991, 101–108.

[Chazelle-Dobkin87] B. Chazelle and D. Dobkin, "Intersection of Convex Objects in Two and Three Dimensions," *Journal of the ACM*, 34(1), Jan. 1987, 1–27.

[Clark76] J. H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM*, 19(10), Oct. 1976, 547–554.

[Cook-et-al84] R. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," *Proceedings of SIGGRAPH '84*, July 1984, 137–146.

[Cook86] R. Cook, "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics*, Jan. 1986, 51–72.

[Crow77] F. Crow, "The Aliasing Problem in Computer-Generated Shaded Images," *Communications of the ACM*, Nov. 1977, 799–805.

[Crow84] F. Crow, "Summed-Area Tables for Texture Mapping," *Proceedings of SIGGRAPH '84*, July 1984, 207–212.

[Dippe-Wold85] M. Dippé and E. Wold, "Antialiasing through Stochastic Sampling," *Proceedings of SIGGRAPH '85*, July 1985, 69–78.

[Feibush-Levoy-Cook80] E. Feibush, M. Levoy, and R. Cook, "Synthetic Texturing using Digital Filters," *Proceedings of SIGGRAPH '80*, July 1980, 294–301.

[Fiume-et-al83] E. Fiume, A. Fournier, and L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer," *Proceedings of SIGGRAPH '83*, July 1983, 141–150.

[Fiume91] E. Fiume, "Coverage Masks and Convolution Tables for Fast Area Sampling," *Graphical Models and Image Processing*, 53(1), Jan. 1991, 25–30.

[Foley-et-al90] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics, Principles and Practice*, 2nd edition, Addison-Wesley, Reading, MA, 1990.

[Fuchs-Kedem-Naylor80] H. Fuchs, J. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," *Proceedings of SIGGRAPH '80*, June 1980, 124–133.

[Fujimoto-Iwata83] A. Fujimoto and K. Iwata, "Jag-Free Images on Raster Displays," *IEEE Computer Graphics and Applications*, (3)9, Dec. 1983, 26–34.

[Garlick-et-al90] B. Garlick, D. Baum, and J. Winget, "Interactive Viewing of Large Geometric Databases using Multiprocessor Graphics Workstations," *Siggraph '90 Course Notes #28 (Parallel Algorithms and Architectures for 3D Image Generation)*, Aug. 1990, 239–245.

[Glassner84] A. S. Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, 4(10), Oct. 1984, 15–22.

[Glassner88] A. S. Glassner, "Spacetime Ray Tracing for Animation," *IEEE Computer Graphics and Applications*, 8(2), March 1988, 60–70.

[Goldstein-Nagel71] R. A. Goldstein and R. Nagel, "3-D Visual Simulation," *Simulation*, Jan. 1971, 25–31.

[Greene-Kass-Miller93] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," *Proceedings of SIGGRAPH '93*, July 1993, 231–238.

[Greene94] N. Greene, "Detecting Intersection of a Rectangular Solid and a Convex Polyhedron," *Graphics Gems IV*, Ed: P. Heckbert, 1994, 71–79.

[Greene-Kass94] N. Greene and M. Kass, "Error-Bounded Antialiased Rendering of Complex Environments," *Proceedings of SIGGRAPH '94*, July 1994, 59–66.

[Greene95] N. Greene, "Hierarchical Polygon Tiling with Coverage Masks," *Siggraph '95 Course Notes #32 (Interactive Walk-Through of Large Geometric Datasets)*, Aug. 1995, C.27–C.49.

[Haines-Wallace91] E. Haines and J. Wallace, "Shaft Culling for Efficient Ray-Traced Radiosity," *Proceedings of SIGGRAPH '91*, July 1994, 59–66.

[Hubschman-Zucker82] H. Hubschman and S. W. Zucker, "Frame to Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World," *ACM Transactions on Graphics*, 1(2), April 1982, 129–162.

[Jevans-Wyvill89] D. Jevans and B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing," *Proceedings of Graphics Interface '89*, June 1989, 164–172.

[Jevans92] D. Jevans, "Object Space Temporal Coherence for Ray Tracing," *Proceedings of Graphics Interface '92*, May 1992, 176–183.

[Kaplan87] M. R. Kaplan, "The Use of Spatial Coherence in Ray Tracing," in *Techniques for Computer Graphics*, Ed: D. F. Rogers and R. A. Earnshaw, Springer-Verlag, New York, 1987, 173–193.

[Kass92] M. Kass, "CONDOR: Constraint-Based Dataflow," *Proceedings of SIGGRAPH '92*, July 1992, 321–330.

[Kay-Greenberg79] D. Kay and D. P. Greenberg, "Transparency for Computer Synthesized Images," *Proceedings of SIGGRAPH '79*, Aug. 1979, 158–164.

[Kay-Kajiya86] T. Kay and J. Kajiya, "Ray Tracing Complex Scenes," *Proceedings of SIGGRAPH '86*, Aug. 1986, 269–278.

[Kaufman86] A. Kaufman, "3D Scan Conversion Algorithms for Voxel-Based Graphics," *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, Oct. 1986, 45–75.

[Lee-Redner-Uselton85] M. Lee, R. Redner, and S. Uselton, "Statistically Optimized Sampling for Distributed Ray Tracing," *Proceedings of SIGGRAPH '85*, July 1985, 61–68.

[Luebke-Georges95] D. Luebke and C. Georges, "Simple, Fast Evaluation of Potentially Visible Sets," *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, April 1995, 105–106.

[Meagher81] D. Meagher, "High Speed Display of 3-D Medical Images Using Octree Encoding," Technical Report IPL-TR-021, Image Processing Laboratory, Rensselaer Polytechnic Institute, Sept. 1981.

[Meagher82a] D. Meagher, "Efficient Synthetic Image Generation of Arbitrary 3-D Objects," *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, June 1982, 473–478.

[Meagher82b] D. Meagher, "The Octree Encoding Method for Efficient Solid Modeling," PhD Thesis, Electrical Engineering Dept., Rensselaer Polytechnic Institute, Troy, New York, Aug. 1982.

[Meagher85] D. Meagher, "The Application of Octree Techniques to 3-D Medical Imaging," *Proceedings of the Seventh Annual Conference of the IEEE Engineering in Medicine and Biology Society*, Sept. 1985, 612–615.

[Meagher91] D. Meagher, "Fourth-Generation Computer Graphics Hardware Using Octrees," *Proceedings of NCGA '91*, April 1991, 316–325.

[Mitchell91] D. P. Mitchell, "Three Applications of Interval Analysis in Computer Graphics," *Siggraph '91 Course Notes #12 (Frontiers in Rendering)*, July 1991, 14.1–14.13.

[Moore66] R. E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1966.

[Moore79] R. E. Moore, *Methods and Applications of Interval Analysis,* SIAM, Philadelphia, 1979.

[Naylor92a] B. Naylor, "Interactive Solid Geometry Via Partitioning Trees," *Proceedings of Graphics Interface '92*, May 1992, 11–18.

[Naylor92b] B. Naylor, "Partitioning Tree Image Representation and Generation from 3D Geometric Models," *Proceedings of Graphics Interface '92*, May 1992, 201–212.

[Reddy-Rubin78] D. R. Reddy and S. M. Rubin, "Representation of Three-Dimensional Objects," Technical Report CMU-CS-78-113, Computer Science Dept., Carnegie-Mellon University, April 1978.

[Rogers85] D. Rogers, *Procedural Elements for Computer Graphics,* McGraw-Hill, New York, 1985.

[Rubin-Whitted80] S. M. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Proceedings of SIGGRAPH '80*, July 1980, 110–116.

[Sabella-Wozny83] P. Sabella and M. Wozny, "Toward Fast Color-Shaded Images of CAD/CAM Geometry," *IEEE Computer Graphics and Applications*, 3(8), Nov. 1983, 60–71.

[Salesin-Stolfi89] D. Salesin and J. Stolfi, "The ZZ-buffer: A Simple and Efficient Rendering Algorithm with Reliable Antialiasing," *Proceedings of the PIXIM '89 Conference*, Hermes Editions, Paris, Sept. 1989, 451–466.

[Salesin-Stolfi90] D. Salesin and J. Stolfi, "Rendering CSG Models with a ZZ-Buffer," *Proceedings of SIGGRAPH '90*, Aug. 1990, 67–76.

[Samet90] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS,* Addison-Wesley, Reading, MA, 1990.

[Sequin-Wensley85] C. Séquin and P. R. Wensley, "Visible Feature Return at Object Resolution," *IEEE Computer Graphics and Applications*, 5(5), May 1985, 37–50.

[Sechrest-Greenberg82] S. Sechrest and D. P. Greenberg, "A Visible Polygon Reconstruction Algorithm," *ACM Transactions on Graphics*, 1(1), Jan. 1982, 25–42.

[Sharir-Overmars92] M. Sharir and M. Overmars, "A Simple Output-Sensitive Algorithm for Hidden Surface Removal," *ACM Transactions on Graphics*, 11(1), Jan. 1992, 1–11.

[Snyder92] J. Snyder, "Interval Analysis for Computer Graphics," *Proceedings of SIGGRAPH '92*, July 1992, 121–130.

[Sutherland-et-al74] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," ACM Computing Surveys, 6(1), March 1974, 1–55.

[Teller-Sequin91] S. Teller and C. Séquin, "Visibility Preprocessing for Interactive Walkthroughs," *Proceedings of SIGGRAPH '91*, July 1991, 61–69.

[Teller-Sequin92] S. Teller and C. Séquin, "Visibility Computations in Polyhedral Three-Dimensional Environments," Univ. of California at Berkeley, Report No. UCB/CSD 92/680, April 1992.

[Teller92] S. Teller, "Visibility Computations in Densely Occluded Polyhedral Environments," PhD Thesis, Univ. of California at Berkeley, Report No. UCB/CSD 92/708, Oct. 1992.

[Warnock69] J. Warnock, "A Hidden Surface Algorithm for Computer Generated Halftone Pictures," PhD Thesis, Computer Science Dept., University of Utah, TR 4-15, June 1969.

[Watkins70] G. S. Watkins, "A Real Time Visible Surface Algorithm," PhD Thesis, Technical Report UTEC-CSc-70-101, NTIS AD-762 004, Computer Science Dept., University of Utah, Salt Lake City, Utah, June 1970.

[Weiler-Atherton77] K. Weiler and P. Atherton, "Hidden Surface Removal using Polygon Area Sorting," *Proceedings of SIGGRAPH '77*, 1977, 214–222.

[Whitted80] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, 23(6), June 1980, 343–349.

[Williams83] L. Williams, "Pyramidal Parametrics," *Proceedings of SIGGRAPH '83*, July 1983, 1–11.