Effective Occlusion Culling for the Interactive Display of Arbitrary Models

by

Hansong Zhang

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1998

Approved by:

Dinesh Manocha, Advisor

Frederick P. Brooks, Jr., Reader

Anselmo Lastra, Reader

Copyright © 1998
Hansong Zhang
ALL RIGHTS RESERVED

ABSTRACT

HANSONG ZHANG: Effective Occlusion Culling for the Interactive Display of Arbitrary Models.
(Under the direction of Dinesh Manocha.)

As an advanced form of visibility culling, occlusion culling detects hidden objects and prevents them from being rendered. An occlusion-culling algorithm that can effectively accelerate interactive graphics must simultaneously satisfy the following criteria:

- Generality. It should be applicable to arbitrary models, not limited to architectural models or models with many large, polygonal occluders.
- Significant Speed-up. It should not only be able to cull away large portions of a model, but do so fast enough to accelerate rendering.
- Portability and Ease of Implementation. It should contain as few assumptions as possible on special hardware support. It must also be robust (i.e. insensitive to floating-point errors).

Based on proper problem decomposition and efficient representations of cumulative occlusion, this dissertation presents algorithms that satisfy all three of the criteria listed above. Occlusion culling is decomposed into two sub-problems—in order for an object to be occluded by the occluders, its screen-space projection must be inside the cumulative projection of the occluders, and it must not occlude any visible parts of the occluders. These two necessary conditions are verified by the overlap tests and the depth tests, respectively. The cumulative projection and the depth of the occluders are represented separately to support these tests.

Hierarchical occlusion maps represent the cumulative projection to multiple resolutions. The overlap tests are performed hierarchically through the pyramid. The multi-resolution representation supports such unique features as *aggressive approximate culling* (i.e. culling away barely-visible objects), and leads to the concept of *levels of visibility*.

Two depth representations, the depth estimation buffer and the no-background Z-buffer, have been developed to store the depth information for the occluders. The former conservatively estimates the far boundary of the occluders; the latter is derived from a conventional Z-buffer and captures the near boundary.

A framework for a two-pass variation of our algorithms is presented. Based on the framework, a system has been implemented on current graphics workstations. Testing of the system on a variety of models (from 300,000 to 15 Million polygons) has demonstrated the effectiveness of our algorithms for the interactive display of arbitrary models.

To Lei

Acknowledgments

I would like to thank:

- My committee members: Dinesh Manocha, my advisor, for suggestions, advice and cheering; Frederick Brooks, Anselmo Lastra, and Turner Whitted, for inspiration and insightful comments on the thesis; Gary Bishop and Nick England, for their encouragement during the course of my research.

- All members of the UNC Walkthrough Project, in particular Daniel Aliaga, Rui Bastos, Jon Cohen, Mike Goslin, Kenneth Hoff, Tom Hudson, and Mark Mine, for the wonderful experience of working together.

- The 1995 Admissions Committee, for admitting me into this paradise for graphics researchers.

- My wife, Lei, for her unconditional love and support.

Contents

Acknowledgments **vi**

List of Figures **x**

1 Introduction **1**

2 Background **6**

2.1 Terminology and Observations 6

2.1.1 Occluders and Occludees 6

2.1.2 The Fundamental Property of Occlusion 7

2.1.3 Occlusion Representation 8

2.1.4 Progressive vs. Multi-pass Occlusion Culling 9

2.2 Related Work 11

2.2.1 Hidden Surface Removal 11

2.2.2 Global Visibility 12

2.2.3 Visibility Culling Algorithms 14

3 Algorithm Outline **22**

3.1 Problem Decomposition 22

3.2 Framework 25

4 Hierarchical Occlusion Maps **28**

4.1 Occlusion Maps 28

4.2 Hierarchical Occlusion Maps (HOM) 31

4.3 Fast Construction of the Hierarchy 32

4.4 Properties of Hierarchical Occlusion Maps 34

5 Overlap Tests Using Hierarchical Occlusion Maps **36**
5.1 Projection Estimation 36
5.2 Hierarchical Overlap Tests 37
5.2.1 The Basic Algorithm 37
5.3 Early Terminations in Overlap Tests 39
5.3.1 Conservative Rejection 40
5.3.2 Aggressive Approximate Culling 40
5.3.3 Computing the Thresholds 43
5.3.4 Predictive Rejection 46
5.4 Summary 47

6 Resolving Depth **49**
6.1 Depth Tests 49
6.2 A Single Plane 51
6.3 Depth Estimation Buffer 52
6.3.1 Updating the Depth Estimation Buffer 53
6.3.2 Depth Tests with the Depth Estimation Buffer 54
6.3.3 Discussions 55
6.4 No-background Z-Buffer 56

7 Occluder Selection **59**
7.1 Static Occluder Selection 59
7.2 Occlusion Preserving Simplification (OPS) 60
7.3 Run-time Occluder Selection 64
7.3.1 Z Plane 64
7.3.2 Distance-Based Selection 65
7.4 Temporal Coherence 66
7.5 Visibility Pre-processing 68
7.6 Discussion 69

8 Implementation and Results **71**
8.1 Pipelining 71
8.2 Test Environments 73
8.2.1 **Scene graphs and the spatial hierarchy** 74
8.2.2 Resolution for occluder rendering. 74
8.2.3 Construction of the occlusion map pyramid. 75

8.2.4 Active levels in the map pyramid. 75
8.3 Performance Measures 75
8.3.1 Graphs 76
8.4 Experimental Results 77
8.4.1 The City Model 77
8.4.2 The Submarine Auxiliary Machine Room (AMR) 79
8.4.3 The Power Plant Model 80

9 Conclusion and Future Work 89

A Incremental Transformation of Axis-Aligned Bounding Boxes 91

Bibliography 93

List of Figures

1.1 Occlusion in the notional model of the auxiliary machine room in a submarine 3

2.1 Cumulative occlusion 7
2.2 A "forest" of smaller objects 8
2.3 Progressive occlusion culling 9
2.4 Multip-pass occlusion culling 10
2.5 Changes in topological appearance 13

3.1 The decomposition of an occlusion test into a 2-D overlap test and a depth test. 23
3.2 Projection and depth representations 24
3.3 An implementation of our algorithms in a two passes 25

4.1 One view of a scene and its corresponding occlusion map. 30
4.2 An example of hierarchical occlusion maps 32
4.3 A pyramid of occlusion maps 33
4.4 Using texture-mapping hardware to accelerate the construction the occlusion map hierarchy. 34

5.1 Approximate culling 42
5.2 Biggest possible square holes given an opacity threshold 44
5.3 Pseudo-code for overlap tests 48

6.1 Definition of depth tests 50
6.2 The single plane depth representation 52
6.3 The depth estimation buffer. 53
6.4 The no-background z-buffer. 56

7.1 View-independent Occlusion-Preserving Simplification 62

7.2 View-dependent Occlusion-Preserving Simplification 63
7.3 Distance-based run-time occluder selection 65
7.4 Temporal coherence hazards in view-frustum culling and occlusion culling 68

8.1 Parallelization using three-stage pipelining 72
8.2 Process scheduling in the *Draw* stage 73
8.3 The city model 82
8.4 A frame from the walkthrough of the city 82
8.5 Frame rate for the city model 83
8.6 Culling in the city model 83
8.7 The auxiliary machine room in a submarine 84
8.8 A frame on the test path for the AMR model 84
8.9 Frame rates for the AMR model 85
8.10 Culling in the AMR model 85
8.11 Aggressive approximate culling on the AMR model 86
8.12 The power plant model 86
8.13 A frame on the test path for the power plant model 87
8.14 The walkway of the 9th floor in the power plant model 87
8.15 Frame rates for the power plant model 88
8.16 Culling in the power plant model 88

Chapter 1

Introduction

A major focus of research in computer graphics has been the interactive display of large 3-D models. Massive models are commonly produced in many applications such as computed-aided design (CAD) of large mechanical systems, architectural visualization, and urban planning. Driven by the need to render large models quickly, the capabilities of hardware graphics systems have been increasing considerably over the years. State-of-the-art high-end commercial systems have peak throughputs of tens of million polygons per second and support advanced rendering features like lighting, texture mapping, and anti-aliasing in real-time. This is a dramatic improvement compared to ten years ago, when high-end systems were capable of only tens of thousands of shaded polygons per second, without texture mapping or anti-aliasing. However, the size and complexity of 3-D models have increased in a more dramatic fashion, as more and more large-scale design and manufacturing projects resort to computer assistance to reduce cost and shorten the development cycle. For example, the model of a Boeing 777 jet passenger airliner has about 2 million parts and a total of 500 million polygons. The brute-force rendering of this model (by sending all the polygons directly to the hardware system) at 20 frames per second would require a graphics system capable of drawing 10 billion polygons per second. Current graphics hardware is not even close to meeting this requirement.

As interactive rendering of large models exceeds the performance limit of hardware systems, measures must be taken to reduce the number of primitives (polygons or curved surfaces) that have to be rendered for each frame. This reduction, however, should not introduce too many visual artifacts, if any at all. Visibility culling, model simplification, and image impostors are among the commonly used primitive reduction techniques. Often, several techniques must be employed at the same time to bring down the primitive count to levels that the graphics hardware can handle at interactive

rates.

Visibility culling is based on the fact that an object does not have to be rendered if the viewer cannot see it; i.e. only visible (or partially visible) objects need to be drawn. Visibility culling algorithms detect objects not visible from the viewer, and prevent them from being rendered. An object in a scene may be non-visible to the viewer for various reasons. For example, objects outside the field of view cannot be seen; removing these objects is called *view-frustum culling*, since the field of view is typically defined by a frustum. If a scene is comprised of closed objects, and the viewer is always on their outside, surfaces facing away from the viewer (back-faces) are always non-visible and can thus be omitted from rendering by *back-face culling*. Furthermore, surfaces inside the field of view and facing the viewer may still not be visible because they can be occluded by other surfaces that are opaque and nearer to the viewer. Detecting such occluded objects and removing them from rendering is called *occlusion culling*. This dissertation presents a new approach to occlusion culling.

Occlusion is one of the most common phenomena in the human visual experience, whether in a virtual environment or in the real world. In typical models, most objects are not visible to the viewer from most viewpoints. This means that, for each frame, only a small portion of the total primitives needs to be rendered. As an example, Figure 1.1 shows the notional model of the auxiliary machine room (AMR) in a submarine, with a total of 632,252 polygons. From the particular view defined by the view-frustum shown in the figure, over 80% of the model (the parts drawn in red) are occluded by the parts shown in blue; in other words, approximately 500K polygons can be culled away with occlusion culling.

If the AMR model is rendered using a typical hardware graphics system with depth-buffer visibility but without occlusion culling, the non-visible parts will consume many system resources (coordinate transformations, lighting, scan-conversion, etc.) before they are discarded by the depth comparison. Although the depth buffer determines visibility correctly, it is located at the end of the rendering pipeline and cannot prevent unnecessary computation on occluded parts in earlier stages. The aim of occlusion culling is to detect large numbers of non-visible primitives and remove them as early as possible. If done in software, it prevents occluded objects from being sent to the hardware graphics system at all.

Occlusion culling has been successfully applied to some specially structured virtual environments such as building interiors. However, the majority of the 3-D models in

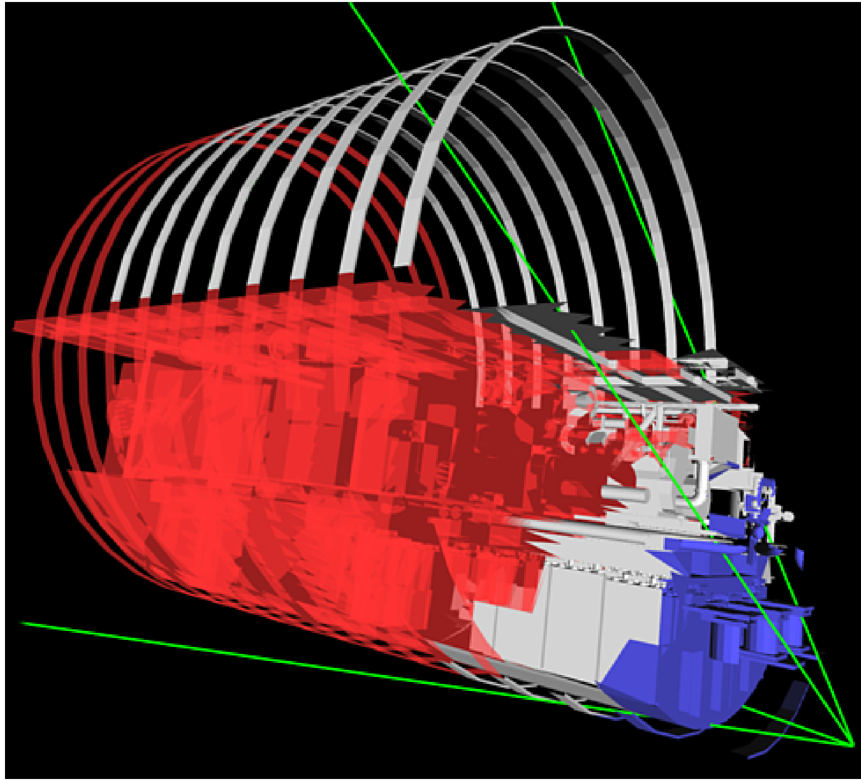


Figure 1.1: Occlusion in the notional model of the auxiliary machine room in a submarine

typical applications (like CAD) are of a general, unrestricted nature. For these models, occlusion culling has not been widely used due to the lack of *effective* occlusion culling algorithms. We believe that an effective occlusion culling algorithm for interactive graphics must satisfy all of the following criteria:

- **Generality.** The algorithm must be applicable to arbitrary models, not limited to architectural models or models with plenty of big, polygonal occluders.

- **Significant Culling and Speed-up.** It must be able to cull away large portions of a model, *and* do so quickly enough to accelerate interactive graphics. The time taken by occlusion culling should not exceed that taken by brute-force rendering of the culled-away portions of scene (although with proper pipelining of culling and rendering, this constraint is no longer necessary.)

- **Portability and Ease of Implementation.** The algorithm should make few assumptions as to graphics hardware features—especially those rarely supported or difficult to implement. It should also be robust, so that implementations are not flawed by floating point errors.

My Thesis and Main contributions. The goal of this dissertation is to provide a new approach to effective occlusion culling, thereby pave the way for the wide-spread use of occlusion culling in interactive 3-D graphics. My thesis is: *By employing a proper decomposition of the occlusion culling problem and efficient representations of occlusion, we can obtain effective occlusion culling algorithms and systems that simultaneously meet all our criteria.* More specifically, our major contributions are as follows:

- We decompose the occlusion culling problem into overlap tests and depth tests. The overlap tests are 2-D, image-space operations and can thus take advantage of image analysis techniques. In our algorithm, occlusion is represented in part by a gray-scale image called an *occlusion map*. The analysis of the occlusion map is based on *hierarchical occlusion maps* (HOM), which form an image pyramid based on the occlusion map.

- The occlusion map pyramid supports fast overlap tests and *high-level occlusion estimation*. The latter has led to the notion of *levels of visibility*, and (among others) the unique feature of *approximate culling* that can remove barely-visible objects.

- For depth tests, we use the *depth estimation buffer* to conservatively estimate the depth ranges beyond which occlusion takes effect. It replaces the z-buffer for occlusion culling purposes and eliminate the need for ordering of objects in depth. As an alternative, we also use a variation of the z-buffer, called the *no-background z-buffer*, when the hardware graphics system supports a user-accessible, conventional z-buffer.

- We have implemented a system that performs effective occlusion culling on real-world models. To the best of our knowledge, this is the first general-purpose occlusion-culling system that runs at interactive frame rates on commercial graphics systems, accelerating the display of arbitrary, large-scale and real-world models.

This dissertation is organized as follows. In Chapter 2 we define the basic terminology, present several of our observations, and review the previous work. In chapter

3, we describe our decomposition of the occlusion culling problem into 2-D overlap tests and depth tests, and give an outline of our algorithms. Chapter 4 introduces hierarchical occlusion maps, the basic data structure on which our algorithm is based. Chapter 5 discusses how the occlusion maps are used in overlap tests, highlighting the special features of our algorithm. We present several depth determination methods in Chapter 6 to round out the occlusion tests. Chapter 7 presents algorithms for occlusion selection, both as pre-processing and at run-time. Finally, Chapter 8 discusses the implementation of our algorithm on commercial graphics platforms, and analyzes the performance we have obtained for different kinds of models. We draw conclusions and propose future work in Chapter 9.

Chapter 2

Background

2.1 Terminology and Observations

In the following discussions we use *objects* to refer to the geometric elements that comprise the scene. A object may be a collection of graphics primitives (polygons or free-form surfaces) or a single primitive; it may or may not correspond to real-world objects such as a table or a chair.

2.1.1 Occluders and Occludees

In the context of occlusion culling, objects in a scene can be classified as occluders and occludees, i.e. those hiding other objects and those hidden. Since visibility changes dynamically as the viewer moves, this classification potentially differs for each frame.

Occlusion culling begins inevitably with selecting some of the objects as occluders, since without these there is no occlusion. The amount of occlusion we will have depends directly on the choice of occluders. The *optimal occluder set* includes only objects that really contribute to occlusion; that is, it should contain only visible objects, and the visible portions of partially visible objects. An occluded object can also occlude other objects, but its occlusion is redundant in the sense that objects it occludes are also hidden by its own occluders. By removing it from the occluder set we do not lose any occlusion.

The problem of selecting an optimal occluder set is therefore the visibility problem (or equivalently, the occlusion culling problem) itself. This is a typical "chicken-and-egg" situation, in which the exact solution to a sub-problem requires a solution to the problem itself. Because of this, we do not seek to compute the optimal occluder set,

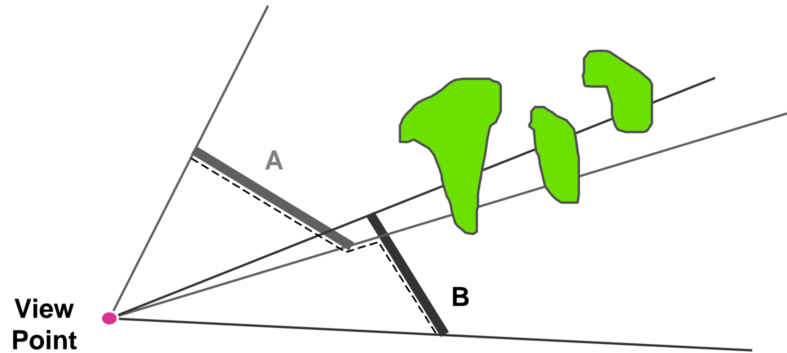


Figure 2.1: Cumulative occlusion

but instead use some heuristics to estimate it. The estimation is made up of *potential occluders*, some of which may be occluded by others.

An object going through occlusion tests is called a *potential occludee* before it is determined to be occluded (or not).

2.1.2 The Fundamental Property of Occlusion

A fundamental fact that must be taken into account in occlusion determination is that the *cumulative occlusion* of multiple objects can be far greater than the sum of what they are able to occlude separately. This is illustrated in 2-D in Figure 2.1 with two occluders A and B. Neither A nor B can individually occlude any of the gray shapes; so if we simply take the union of objects they separately occlude, the gray shapes will still not be considered their occludees. However, A and B do occlude the gray shapes if their occluding effects are combined to produce the cumulative occlusion, which is equivalent to the occlusion of the dotted line from the particular view. A more extreme case is shown in Figure 2.2, where the scene has a dense population of small objects. In this case, the simple sum of the objects' separate occlusions is probably zero since no object completely occludes any other objects. But, in fact, there is good occlusion in this scene—one is not able to see very deep into the scene before the cumulative occluding effects of the small objects completely block the view. Computing cumulative occlusion is called *occluder fusion*.

Note that cumulative occlusion is largely view-dependent, i.e. total occlusion provided by the same set of objects can vary greatly depending on the viewpoint. Thus, in general, cumulative occlusion has to be dynamically computed at each frame.

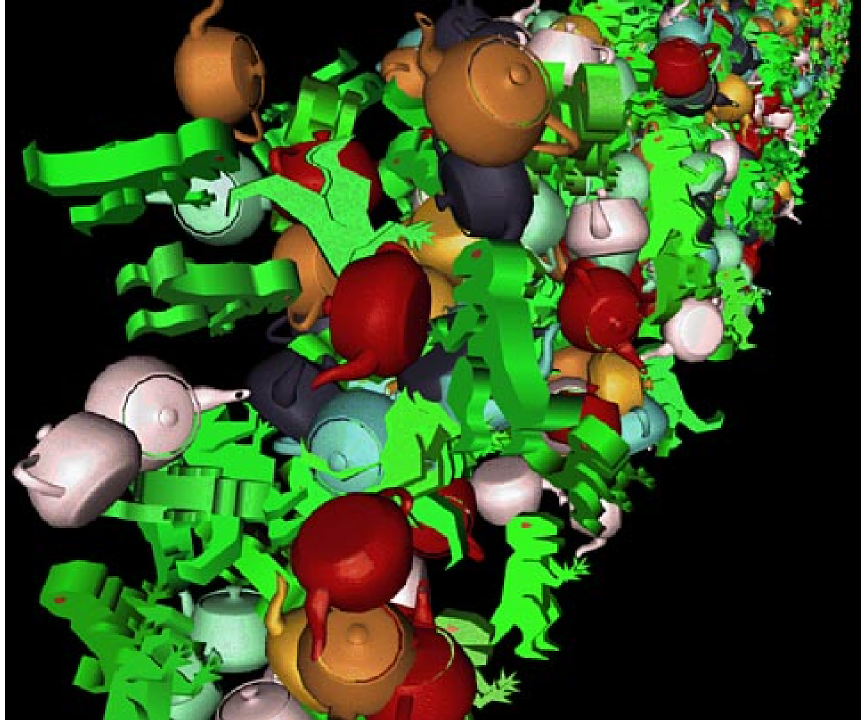


Figure 2.2: A "forest" of smaller objects

An occlusion culling algorithm must be able to perform occluder fusion dynamically and efficiently to be effective in general environments. In fact, their not being able to do so is a major reason why most object-space occlusion culling algorithms are limited to special environments, where occluder fusion is less important due to the abundance of large polygonal occluders (see section 2.2, Previous Work, for details).

2.1.3 Occlusion Representation

The fundamental property of occlusion discussed in the previous section immediately points to a need for a separate representation of cumulative occlusion from multiple objects. The scene representation itself usually does not contain this view-dependent information. The representation of individual objects cannot well accommodate it either, since it is of a more global nature and at the multi-object level. A good representation of cumulative occlusion must be easy to compute from given occluders, and it must be easy to use for occlusion tests.

An occlusion representation can be in object space or image space. In object space, the result of occluder fusion for the scene and view shown in Figure 2.1 would be represented by the dotted poly-line. Analogously, in 3-D, the fusion could be

```
Initialize occlusion representation (OR) to empty;
For each object*
    Perform occlusion test(s) against OR;
    If occluded
        Discard object (cull);
    else
        Render object;
        Update OR;
```

*The objects have to be traversed in roughly front-to-back order in order for objects traversed later to take advantage of occlusion from objects traversed earlier.

Figure 2.3: Progressive occlusion culling

represented by a polygonal mesh. However, computing such representations geometrically for arbitrary object configurations is a very difficult problem. An image-space representation is easier to obtain. For example, the depth buffer from the rendering of occluders represents their cumulative occlusion; actually, it can be regarded as a discrete representation of the polygonal mesh we have just mentioned. This process is robust and easily accelerated by ordinary graphics hardware.

The representation of occlusion is the most important aspect of an occlusion culling algorithm, and largely decides its capabilities. As will be seen in the following chapters, the advantages of our approach to occlusion culling is rooted in our new occlusion representations that treat the occluders' screen projection and depth separately.

2.1.4 Progressive vs. Multi-pass Occlusion Culling

Occlusion determination is by nature progressive in the sense that any object, once determined visible, becomes an occluder. Its contribution should be accumulated in the occlusion representation before the next object goes through occlusion tests, since the next object might be occluded because of the newly added occlusion. The process of progressive occlusion culling is shown in Figure 2.3. Note that to cull as much as possible, the objects have to be traversed in roughly front-to-back order, so that any given object is likely to be farther away than previously traversed objects, and thus likely to be able to take advantage of the cumulative occlusion.

A problem with this approach is that each update to the occlusion representation

```
OR: the occlusion representation
PO: the set of potential occluders to be accumulated into OR
Initialize OR to empty;
Initialize PO to empty;
For each object
    Perform occlusion test(s) against OR;
    If occluded (culled)
        Discard object;
    else
        Render object;
        Add object to PO;
        If PO is large enough
            Update OR with objects in PO;
            Reset PO to empty;
```

Figure 2.4: Multip-pass occlusion culling

may have some constant but non-trivial overhead which makes per-object updates too expensive to implement. For example, in the algorithm proposed in this dissertation, updating the occlusion representation involves reading part of the frame buffer. Each read takes substantial time to set up on typical graphics hardware. In this case, multi-pass occlusion culling has to be used instead of progressive culling to reduce the number of updates.

When an object is determined to be visible, its contribution to occlusion is not accumulated to the occlusion representation immediately. Rather, it is put into a potential-occluder set. Contributions from objects in the set are merged into the representation in a single update in the future. That is, an update to the occlusion representation in a multi-pass algorithm is performed for the multiple objects in the potential occluder set. The set contains only *potential* occluders because an object in the set may well have been detected as occluded, had the occlusion of some other objects in the set been accumulated promptly in the occlusion representation (as in the case of progressive culling). But since the update is not progressive, no object in the set can be occluded due to the contributions of other objects in the set. Consequently, an object may be used to update the occlusion representation even if it is occluded. This wasted computation, which does not exist in progressive occlusion culling, is a trade-off in an effort to reduce the number of updates.

An update can be triggered by a variety of criteria. For example, we may set a limit on the total number of polygons in the potential occluder set. Once the limit is exceeded, an update is performed and the set is emptied. An update to the occlusion representation corresponds to a pass in multi-pass occlusion culling. Progressive occlusion culling can be conveniently viewed as multip-pass culling with one pass per object. The general multi-pass algorithm is outlined in Figure 2.4.

We have found that one-pass occlusion culling (in which the occlusion representation is updated only once per frame) has often to be used to minimize the overhead of updating the occlusion representation. This special case of the multi-pass algorithm is executed in the following steps:

1. Select occluders;
2. Build occlusion representation;
3. Occlusion culling;
4. Final Rendering;

2.2 Related Work

The fields of visibility determination, and in particular occlusion culling, have been areas of active research ever since the early days of computer graphics. We now review some of the previous work, classified into three categories:

- Hidden surface removal algorithms, for computing the visible portions of a collection of geometric surfaces.
- Global visibility algorithms, which compute and store visibility information to support visibility queries between pairs of primitives, or track visibility changes as the viewer moves
- Occlusion culling algorithms, which detect and discard the hidden portions of a model in large groups to speed up rendering.

We will mention important work in the first two categories quickly, and have a detailed look at the previous occlusion culling algorithms.

2.2.1 Hidden Surface Removal

At the very least, computer graphics systems should be able to display surfaces with correct visibility. This makes hidden surface removal (or equivalently, visible sur-

face determination) a fundamental problem in computer graphics. Given a geometric model (most often a collection of polygons) and the viewing parameters, hidden surface algorithms find out which surfaces or parts of surfaces are visible to the viewer. Sutherland et. al. [SSS74] presents a characterization of the algorithms. The book by Foley et. al. [FDFH90] has a chapter on visible-surface determination that includes most of the algorithms developed so far. Specifically, algorithms for hidden surface removal include visible-line determination [Rob63, App67], the z-buffer (or depth-buffer) algorithm [Cat74], the depth-sort algorithm [NNS72], scan-line algorithms [WREE67, BK70, Bou70, Wat70], area-subdivision algorithms [War69, WA77] and ray-tracing [App68].

Visibility determination is closely related to depth sorting of primitives. Once the primitives are ordered in depth, they can be rendered back-to-front for correct visibility. The binary space partitioning (BSP) algorithms [FKN80] produce an organization of polygons from which their depth ordering can be quickly derived. More recently, Naylor ([Nay92]) described a algorithm which projects a 3-D BSP tree in object space into a 2D BSP tree in screen space for depth ordering and occlusion culling. In general, the problem with BSP-trees is that polygons frequently have to be split as they are registered in the tree, and, for an n-polygon scene, the splitting can generate $O(n^2)$ new polygons in the worst case. Although the worst case hardly ever happens in practice, splitting the polygons involves computing intersections of the polygons, which is not numerically robust. Also, dynamic model changes are a challenge for BSP algorithms.

There has been significant research in visible surface computation in computational geometry, and many algorithms have been proposed [Mul89, McK87] (see [Dor94] for a recent survey). However, the practical utility of these algorithms is unclear at the moment.

2.2.2 Global Visibility

Global visibility algorithms pre-compute visibility information for arbitrary viewer positions in the 3-D space and store it in special data structures. At run-time, correct visibility is retrieved from the data structures according to the current view point.

Aspect graphs have been extensively investigated in computer vision for applications like object recognition for robots [GM90, GCS91, PD90, CH92]. The goal for this research is to characterize changes in *aspect*, i.e. topological appearance, of surfaces as the viewer moves in space. The changes are called *visual events*. As an

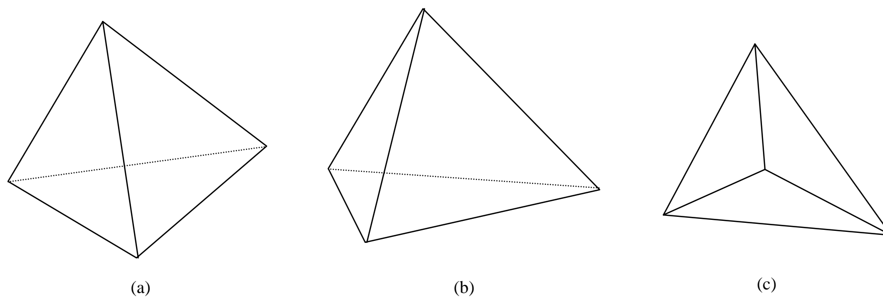


Figure 2.5: Changes in topological appearance

example, Figure 2.5 shows three views of the same tetrahedron. The first two views, (a) and (b), share the same topological appearance, whereas view (c) is topologically different from (a) and (b). The 3-D space is subdivided into volumes of constant aspect (i.e. the same topological appearance), separated by boundaries across which visual events occur. The dual of this subdivision is called the *aspect graph*. [PD90] presented algorithms for generating the aspect graph under parallel or perspective projections, for convex or non-convex polyhedra. They also introduced a complex data structure named *asp* to store the results. [CH92] presented a different aspect graph algorithm for non-convex polyhedra under perspective projections.

Form-factor computation in radiosity algorithms (for simulating global diffuse illumination) requires global visibility information among pairs of patches. Solutions to this problem bear much resemblance to the construction of aspect graphs. [TH93] introduced an algorithm which conservatively supports visibility queries between two patches, as well as providing a list of blocking objects between them. The visibility complex [Poc92, PV95] stores 2-D global visibility information for "flatland" radiosity (i.e. the imaginary "radiosity" computation among lines on a plane). Its 3-D extension, the 3-D visibility complex [DDP96], is similar to the *asp*. More recent research along this line includes the visibility skeleton [DDP97], which aims to simplify the intricate data structures of the visibility complex.

The application of global visibility algorithms in interactive display of complex models is limited by the fact that the worst-case complexity is these algorithms is as high as n^6. Although the worst case is seldom encountered in practice, interactive graphics has had difficulty handling even the much lower complexity of n^2. In general, computing and storing global visibility into a database as preprocessing and then querying the visibility database at run-time do not constitute a feasible solution to the visibility determination problem for large models.

2.2.3 Visibility Culling Algorithms

The traditional hidden-surface removal algorithms focus on generating images with correct visibility relationships among objects. They are inefficient for a scene with many polygons because they have to explicitly traverse and process every polygon in the scene to generate a visibility solution. Visibility culling algorithms, on the other hand, seek to quickly detect and remove large portions of a scene not seen by the viewer, and thus accelerate image generation. Visibility culling algorithms do not replace traditional hidden surface algorithms, because they do not resolve visibility at the fine level (e.g. at the pixel level for raster graphics) required for the final graphics output. The two types of algorithms must work together to generate a rendered frame: visibility culling first removes large groups of hidden polygons quickly, and traditional hidden-surface algorithms will then process the remaining polygons and generate the final output.

Visibility culling is based on two key ideas: *hierarchical data structures* and *potentially visible sets*.

2.2.3.1 Hierarchical Data Structures

Geometric models often contain millions of primitives. Traversing all the primitives to compute visibility for each of them is too expensive to do in real-time. Hierarchical organization of primitives helps to reduce this per-primitive operation to an efficient logarithmic search.

Clark [Cla76] proposed a tree-structured *bounding-volume hierarchy* to accelerate the rendering of complex models. Every object has a bounding volume, which is simple in shape and spatially contains the object. For instance, the axis-aligned bounding box is one of the most commonly used bounding volumes. Using a regularly-shaped bounding volume instead of the object itself greatly accelerates visibility determination. An object cannot possibly be visible if its bounding volume is not. However, there is no guarantee that the object will be partially visible if the bounding volume is partially visible, since the bounding volume generally encloses more space than the object itself. So if we determine an object to be partially visible because of its partially visible bounding volume, and thus decide to render the object, we may actually end up rendering a non-visible object. Thus the use of bounding volumes actually introduces the concept of *conservative culling*, which will be further discussed when we review the notion of *potentially visible sets* in the next section.

From bounding volumes of objects, higher-level bounding volumes can be built by grouping adjacent volumes into larger ones. This process can be applied recursively to construct in a tree of bounding volumes called the *bounding-volume hierarchy*. In the hierarchy, the leaves are the bounding volumes of objects and the root bounding volume encloses the whole scene. The main purpose of having a bounding volume hierarchy is to take advantage of the *spatial coherence* of the model: objects close to each other tend all to be either visible or non-visible at the same time. The bounding volume hierarchy collects and bounds spatially adjacent objects so that spatial coherence can be exploited, in a hierarchical fashion, in visibility tests. If a non-leaf bounding volume is determined to be totally non-visible, so must all its descendant bounding volumes; thus they can be culled away without being tested individually for visibility. On the other hand, if the non-leaf bounding volume is only partially visible, then its child bounding volumes may have a chance to be completely hidden due to their smaller spatial extent. So we descend into the child bounding volumes to perform further visibility tests. This process of *hierarchical visibility culling* is an efficient logarithmic search for non-visible objects that removes large groups of non-visible geometry without inspecting individual primitives. Clark proposed hierarchical view-frustum culling, which has since been widely used. However, the "recursive descent visible-surface algorithm" he proposed is unlikely to be efficient for lack of a representation of cumulative occlusion.

Spatial proximity determines how objects are (recursively) clustered in forming a bounding volume hierarchy, but it is not the only possible criterion for grouping object in a hierarchical model organization. For example, the special-purpose hierarchy used in hierarchical back-face culling ([KMGL96] has been built with both spatial and normal distributions in mind.

Hierarchical data structures for model representation exploit the spatial coherence among 3-D primitives. For 2-D images, data structures have also been proposed to take advantage of *image-space coherence*, the fact that pixels near to each other tend to have the same properties (color, opacity, etc.). This type of coherence means blocks of pixels with similar properties can be considered as a whole without each being inspected separately. Such blocks are often formed hierarchically in a quad-tree structure, with tree nodes at different levels representing different block sizes. The quad-tree is commonly called an *image pyramid* [TP75].

Image pyramids are typically employed for analyzing and accessing images in *multiple resolutions*. The pyramid supports viewing of the scene at a higher, more

general level (i.e. through a larger aperture or at a greater distance), with the details "smoothed out" but global features preserved. An examples of this is the use of mip-maps[Wil83], or hierarchical textures, to alleviate aliasing in texturing an object at different distances. Burt [Bur88] designed and implemented the Pyramid Vision Machine based on the hierarchical (pyramidal) representation of images. The image pyramid was used to support coarse-to-fine searches and fine-to-coarse measurement of the images, which facilitates the author's efforts to achieve "smart sensing" (i.e. selectively gathering visual information critical to the task at hand).

As we will demonstrate in this dissertation, image pyramids are useful tools in the analysis of occlusion. In particular, we use an *occlusion-map pyramid* as part of our representation of occlusion.

2.2.3.2 Potentially Visible Set

The ultimate goal of occlusion culling is to increase frame rates. Consequently, the time taken by identifying visible or hidden objects must be well controlled. Finding the exact set of visible objects often takes much more time than that saved by not rendering non-visible objects, in which case visibility culling actually leads to *slower* frame rates. So most algorithms seek to find only the *potentially visible set* (PVS), proposed by [ARB90], which is a superset of the exact set of visible objects; in other words, a typical PVS contains all the visible objects and some non-visible objects. The PVS is less expensive to find than the exact set and can still be significantly smaller than the original model. Finding a PVS is often called *conservative* visibility culling. The quality of a PVS is measured by how conservative it is, i.e. the degree to which it approximates the minimal visible set. However, it should be emphasized that the effectiveness of occlusion culling in rendering acceleration depends not only on the quality of a PVS, but also on the time taken to compute it. It is often worthwhile to build a more conservative PVS in exchange for less time overhead, in an effort to reduce the total frame time.

The notion of a PVS nicely sums up the goal of all visibility culling algorithms: to produce a PVS. Two simple forms of visibility culling are view-frustum culling and back-face culling. View-frustum culling removes objects outside the field of view. Back-face culling removes primitives facing away from the viewer, which cannot possibly be seen if the scene is made up of closed objects and the viewer remains outside them. Each method can be accelerated by a hierarchical data structure. They produce a very conservative PVSs, but due to their low overhead (relative to the

amount of geometry they can typically cull away) they are among the most widely-used rendering-acceleration techniques.

As a more sophisticated form of visibility culling, occlusion culling detects and discards objects which, due to occlusion from other objects, are not visible from the viewer. This is much more difficult than view-frustum or back-face culling. Below we review some major occlusion culling algorithms.

2.2.3.3 Cells and Portals

The terms *cells* and *portals* were first used by Jones [Jon71] in his algorithm for hidden line removal. In his algorithm, the model geometry is subdivided into convex polyhedral cells and convex polygonal portals, so that every polygon in the model belongs to the face of one or more cells. This subdivision is represented by a cell adjacency graph, in which any two cells that share a portal are adjacent. For rendering with hidden-line removal, the graph is traversed in depth-first order. The traversal begins by drawing the faces and portals of the cells containing the view point. After each portal is drawn, the cell on the other side of the portal is recursively traversed. The *portal sequence*, i.e. all the portals on the path from the cell containing the viewer to the cell that is being traversed, forms a mask to which the faces of the current cell are clipped. The mask represents the intersection of the infinite frusta defined by the viewpoint and each portal in the sequence. If the a portal does not intersect the mask, it must be hidden from the viewer, and the cell on the other side need not be traversed.

Airey [ARB90] proposed the notion of *densely occluded environments*—scenes only a small fraction of which can be seen from most viewpoints. He further studied one type of such environments suitable for a portal-based treatment, namely the architectural environment. The interior of a building is subdivided into rooms, which are mutually occluded from one another except through doors and windows. The rooms correspond to cells, and doors and windows to portals. In such environments, the visibility problem is reduced to computing cell-to-cell visibility, by considering sequences of portals. Airey suggested multiple way to precompute cell-to-cell visibility, including shooting random rays from the portals and using shadow volumes. Teller et. al. [TS91] characterized cell-to-cell visibility as a linear programming problem and gave a closed-form analytical solution. In both approaches, cell-to-cell visibility is computed and stored as preprocessing; the time and storage space required by the preprocessing tend to be excessive for large models. To eliminate the expensive

preprocessing, Luebke et. al.[LG95] proposed a new approach to evaluating portal sequences, resulting in fast, dynamic, but more conservative determination of cell-to-cell visibility. Due to the dynamic nature of their method, interactive modifications to the environment (e.g adding, moving, or resizing portals) are supported.

The above techniques are restricted to environments that can be divided into cells and portals. They are not very effective for outdoor, open-space scenes and other non-architectural models.

Other algorithms for densely-occluded but somewhat less-structured models have been proposed by Yagel and Ray [YR96]. They used regular spatial subdivision to partition the model into cells. However, the resulting algorithm are very memory-intensive and does not scale well to large models.

2.2.3.4 Other Object-Space Approaches

Other object-space approaches to occlusion culling have been developed for environments more general than building interiors. Coorg and Teller [CT97] and Hudson et al. [HMC+97]. proposed object-space occlusion culling algorithms for environments with many of large polygonal occluders. These algorithms dynamically choose a subset of polygons as occluders and use them for occlusion culling. [CT97] computed an arrangement corresponding to a linearized portion of an aspect graph; at run-time, they tracked the viewpoint with respect to the arrangement to check for occlusion. [HMC+97] made use of shadow frusta formed by the occluder polygons and the viewpoint. Objects completely inside one of the shadow frusta are culled away.

These algorithms are not restricted to indoor, architectural models; however, the choice of occluders is limited to polygons, convex objects, or simple combination of convex objects (e.g. two convex polytopes sharing an edge). They do not combine "forests" of small, non-convex, and disjoint occluders for significant cumulative occlusion, and are thus effective only in scenes with big, well-shaped occluders, e.g. city models. In other words, the general effectiveness of these algorithms is limited due to the lack of a representation for cumulative occlusion.

2.2.3.5 Hierarchical Z-Buffer

The hierarchical Z-buffer algorithm was presented and improved by Greene in [GK93, GK94]. In our terminology, cumulative occlusion is represented by an ordinary z-buffer, which forms the finest level of the Z-pyramid. Other levels of the pyramid

are built by recursive filtering with a maximum operator on 2×2 blocks of pixels; that is, at all other levels, each entry is the farthest z value in the corresponding 2×2 block of the next finer level. Whenever the z-buffer is modified during scan-conversion of the primitives, the new z values are propagated to coarser levels. The scene is organized into an octree to facilitate hierarchical culling and front-to-back traversal of the nodes.

The algorithm follows the same steps as the typical progressive occlusion culling algorithm shown in Figure 2.3. For each frame, the octree nodes are processed in front-to-back order. Each node (a cube) is first scan-converted and tested against the Z-pyramid to see if any part of it is visible (i.e. if any visible pixels are created in the scan-conversion of the cube). If the box is non-visible, so must be the geometry inside. Otherwise, the geometry is rendered, and the Z-pyramid is updated. The scan-conversion of the cube is done hierarchically from coarse to fine resolutions corresponding to the levels of the z-pyramid. If, at a coarse level, a pixel of the cube already has a greater z value than the pixel in the z-pyramid, then the corresponding part of the cube is determined to be hidden. Otherwise, the algorithm goes into the next finer level for further tests. In so doing, the algorithm takes advantage of image-space coherence to reduce the number of depth comparisons.

The algorithm also exploits temporal coherence by first rendering visible primitives from the previous frame. The image-space occlusion representation by the Z-pyramid makes the algorithm well suited for general models.

However, using this algorithm for interactive rendering involves many hardware assumptions. Maintaining the Z-pyramid in real-time is expensive (in terms of the size and bandwidth of the frame buffer memory) and not supported by current hardware. The only purpose of the Z-pyramid is for hierarchical depth comparison, thus it requires careful analysis to ensure that the overhead of pyramid generation pays off in the reduced number of depth comparisons (otherwise a plain Z-buffer should be used instead). Hardware Z-queries (i.e. whether scan-converting certain primitives yields any visible pixels) are also rarely supported in hardware and can be expensive to perform in a pipelined architecture.

[Geo95] described an implementation of the above algorithm (but without the z-buffer hierarchy) on a parallel graphics computer (Pixel-Planes 5).

2.2.3.6 Hierarchical Polygon Tiling

The hierarchical z-buffer algorithm is prone to aliasing problems (i.e. zig-zag polygon edges) because the point sampling used in scan conversion. In an effort to reduce aliasing, Greene [Gre96] presented a hierarchical tiling algorithm using coverage masks. The algorithm combines mask-based hierarchical polygon scan conversion with hierarchical occlusion culling. It employs a hierarchy of tri-valued masks, known as the *triage mask pyramid*, to hierarchically represent the screen projections of primitives as they are rendered. Each entry in a mask represents a rectangular region in the image; the value of the entry indicates whether the region is vacant, covered or active (partially covered). An entry in the mask pyramid is split into 4 quadrants in the next finer level of the pyramid. The entries are updated during the hierarchical scan conversion of the polygons.

Without other means to resolve depth, the algorithm organizes the model as an octree of BSP-trees, in order to achieve strict front-to-back polygon traversal and hierarchical visibility culling at the same time. With guaranteed depth ordering, the coverage pyramid suffices as an occlusion representation. Similar to the hierarchical z-buffer approach, the algorithm traverses the nodes in the octree in front-to-back order. The octree node is hierarchically scan-converted and compared to the coverage pyramid. If it is in a covered region, then it is occluded, and consequently all geometry in it is culled away. Otherwise, polygons are tiled into the mask pyramid using hierarchical polygon scan-conversion (the final image is also updated); the BSP tree in each octree node provides front-to-back ordering of the polygons.

The application of this algorithm for interactive purposes requires support for hierarchical scan-conversion in hardware; it remains to be seen whether such scan conversion is friendly to hardware implementations. The use of BSP-trees seriously limits its ability to handle large, arbitrary models, e.g. those from CAD applications. Also, it is currently limited to polygonal models only.

2.2.3.7 Other Algorithms

There is substantial literature on the visibility problem from the flight simulator community. Their publications have described algorithms that often bear strong resemblance to those developed separately (and often later) in the computer graphics community. An overview of flight simulator architectures is given in [Mue95]. Most notably, the Singer Company's Modular Digital Image Generator [Lat94] renders

polygons in front-to-back order using a hierarchy of mask buffers to skip over already covered spans, segments or rows in the image. General Electric's COMPU-SCENE PT2000 [BE89] uses a similar algorithm but does not require the input polygons to be rendered in front-to-back order, and the mask buffer is not hierarchical. The Loral GT200 [Lor94] first renders near objects and fills in a (possibly hierarchical) mask buffer, which is used to cull away far objects. This is similar to the hierarchical coverage masks used in the hierarchical polygon tiling algorithm. The SOGITEC APOGEE system [(SO94] uses the Meta-z-buffer, which is similar to hierarchical z-buffer.

Chapter 3

Algorithm Outline

In this chapter, we present a high-level description of our occlusion culling algorithm, focusing on the fundamental ideas and a framework for its implementation. Details will be presented in in the following chapters.

3.1 Problem Decomposition

Occlusion representations largely determine the capabilities and features of an occlusion culling algorithm. In our approach, cumulative occlusion is represented in two parts. First, an occlusion map captures the *cumulative projection* of multiple arbitrary occluders. The occlusion map is a gray-scale image, which is analyzed, in multiple resolutions, through the construction of an occlusion map pyramid (Chapter 4). Occlusion maps do not contain depth information; separate depth representations are computed either by estimating the occluders' depths using a software *depth estimation buffer*, or, if viable, by deriving a *no-background Z-buffer* from a conventional depth buffer (Chapter 5).

Our two-part representation of occlusion reflects a *decomposition* of visibility determination (or equivalently, occlusion determination) into two sub-problems: a two-dimensional overlap test and a depth test. The former decides whether the screen-space projection of a potential occludee lies completely within the union of the screen space projections of all the occluders, while the latter determines whether a potential occludee is behind the occluders.[1] Occlusion maps are used for the overlap tests, and

[1] On a side note, when a conventional depth-buffer is used to resolve visibility, the overlap test is implicitly performed as a side effect of depth comparisons by initializing the Z-buffer with large numbers.

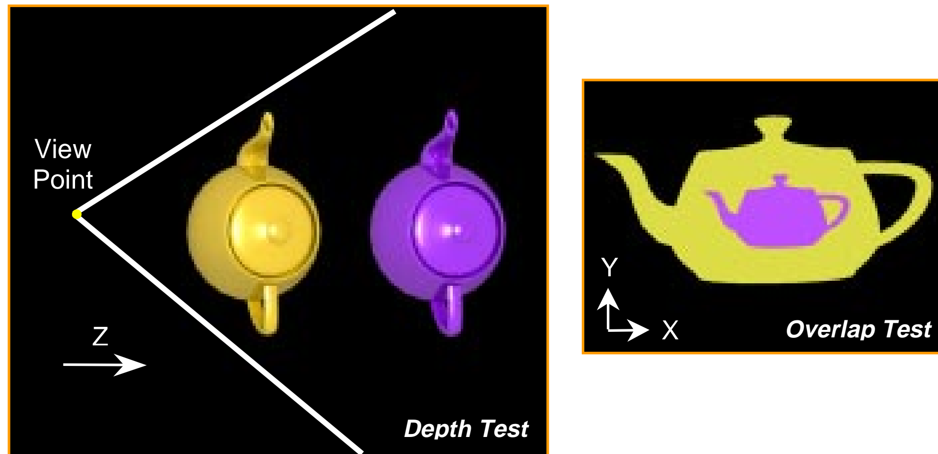


Figure 3.1: The decomposition of an occlusion test into a 2-D overlap test and a depth test.

the depth representations support depth tests. These two tests correspond to two necessary conditions that combine to be a sufficient condition for occlusion. This decomposition is qualitatively illustrated in Figure 3.1. In order for the darker teapot to be occluded by the light-colored one, its screen projection must lie inside that of the latter, and it must be farther away.

This decomposition is desirable for several reasons. First, it is backed by the observation that the amount of occlusion we get is more sensitive to approximations in the cumulative projection than to the approximations in determining depth. This is illustrated in 2-D in Figure 3.2. In 3.2(a), the thick line segment corresponds to the cumulative projection of the occluders. In 3.2(b) we conservatively estimate the cumulative projection, and the estimation is shown by a shorter line as compared to (a). The gray regions in (b) show the loss of occlusion due to the conservative estimation. Depth representations are shown in (c) and (d). In both cases, the occluders' depth is conservatively estimated by a line (shown dotted). The depth test is then to determine whether the tested object is behind the line (relative to the viewer); if it is, then it must be behind the occluders. In (d), we use a more conservative estimation than (c), and the loss of occlusion is indicated by the gray area. Now, as we have greater and greater depth in the scene (imagine the scene extends to the right of the figures), the gray area in (d) remains unchanged and thus becomes less and less significant as compared to the total volume in the view frustum. On the other hand, the gray area in (b) extends and grows in proportion to the depth

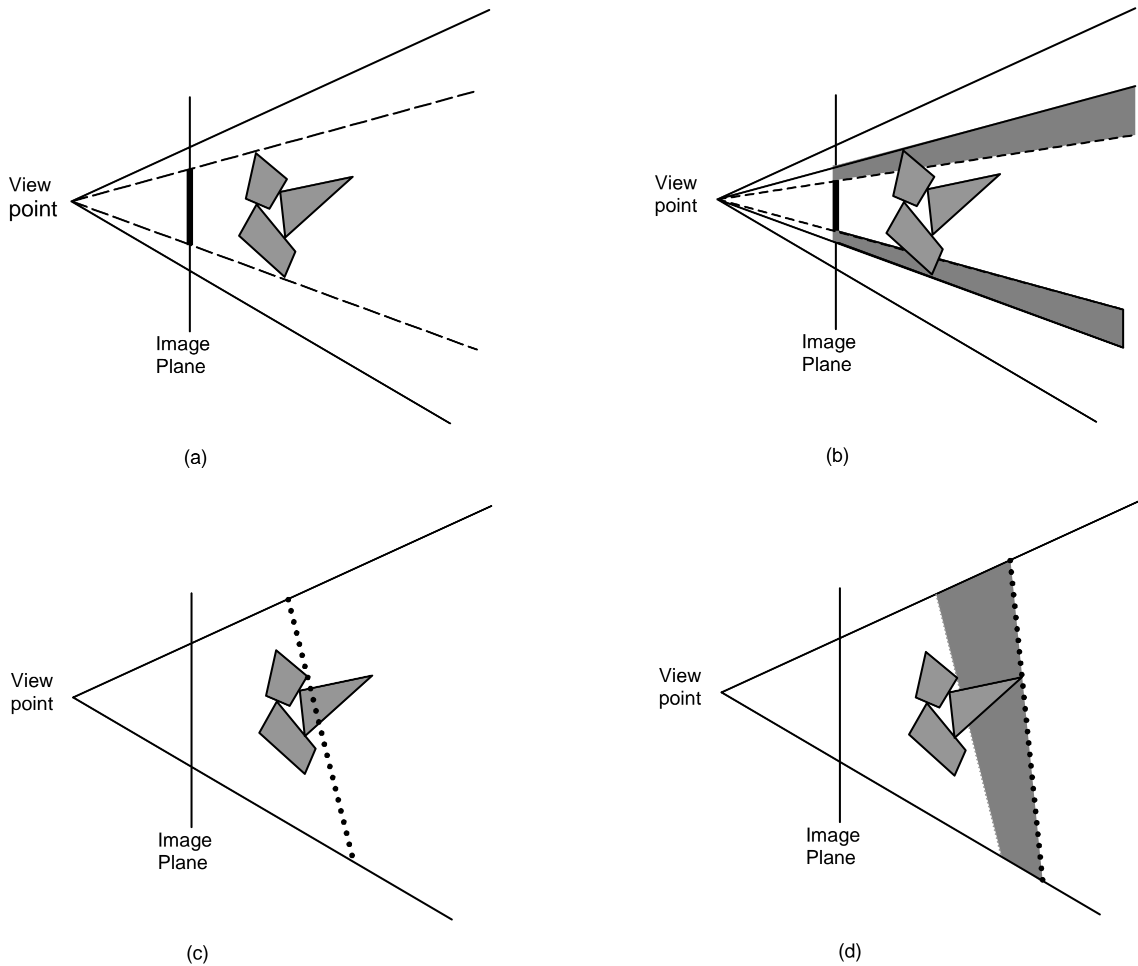


Figure 3.2: Projection and depth representations

of the scene.

This implies that it is beneficial to consider projections and depth separately: we can afford to be more conservative and more approximate with depth than we can with projections. Separating the two leads to more flexibility and leverage in algorithm design.

Another major advantage of our decomposition is that, since the cumulative projection can be represented by a gray-scale image (the occlusion map), 2-D image processing techniques can be applied to analyze it. In particular, we recursively low-pass filter the occlusion map into an image pyramid. This has led to fast overlap tests, the notion of *levels of visibility*, and such unique features as *aggressive approximate culling* (i.e. the removal of "barely visible" objects).

The third benefit of our decomposition is that it enhances the portability of our

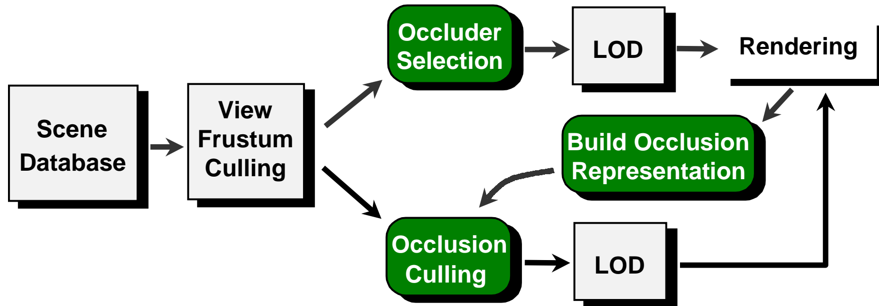


Figure 3.3: An implementation of our algorithms in a two passes

algorithm. All hardware graphics systems by definition produce images, and thus occlusion maps. However, how depth information is managed varies greatly. Some systems do not have a depth buffer. Some have a depth buffer but it is not accessible to the user or too slow to access in real-time. Others have easily accessible depth buffers. By using a separate depth representation, we are able to employ different ways of depth management as appropriate for the underlining graphics system. Different depth representations can be plugged in without changing other parts of our algorithm.

3.2 Framework

Here we outline the two-pass version of our occlusion culling algorithm. It actually implements the one-pass occlusion culling algorithm presented in section 2.1.4, in which the occlusion representation is updated once for each frame. The number of passes of an algorithm is conventionally regarded as the number of accesses to the hardware graphics system, not the number of updates to the occlusion representation. We access the hardware system twice per frame: once for building occlusion representations, and once for final rendering. So, by convention, ours is a two-pass algorithm. The rendering pipeline with our algorithm incorporated is illustrated in Figure 3.3. The shaded blocks indicate components unique to our algorithm. For each frame, the pipeline executes in two major phases:

1. **Building the occlusion representation:** The occluders are selected from the scene database and rendered to build the occlusion map hierarchy. The depth representation is computed as well.

(a) **Occluder selection:** The bounding volume hierarchy of the occluder database is traversed to find objects lying (totally or partially) inside the viewing frustum. A subset of these objects is then selected as occluders, based on a distance-based criterion and temporal coherence (Chapter 7).

(b) **Rendering occluders and building the depth representation:** The selected occluders are rendered at proper levels of detail to form an image in the framebuffer that is the original occlusion map (Chapter 4). If the depth estimation buffer is chosen as the depth representation, it is updated for each occluder; or, if the no-background z-buffer is used, it is derived from the conventional z-buffer after occluder rendering (6).

(c) **Building the occlusion map hierarchy:** The original occlusion map is recursively filtered to generate an occlusion map hierarchy (an image pyramid.) This process can be accelerated by texture mapping with bilinear interpolation (Chapter 4).

2. **Visibility culling and final rendering:** Having built the occlusion representation in the first pass, the algorithm now traverses the bounding volume hierarchy of the model database to perform visibility culling. Objects not culled away are then rendered to produce the final image.

   (a) **View-frustum culling:** Standard view-frustum culling is applied to the model database.

   (b) **Occlusion culling:** Each object in the view frustum is considered a potential occludee and subject to occlusion culling. The following two tests are performed for each potential occludee:

      i. **Overlap tests:** The screen projection of the potential occludee is tested against the occlusion map hierarchy to see whether it is completely within the opaque area of the cumulative projections of the occluders (Chapter 5).

      ii. **Depth tests:** The potential occludee is tested against the depth representation to determine whether it is behind the occluders; or more precisely, if it is behind a boundary beyond which occlusion takes effect (Chapter 6).

   (c) **Final rendering:** Object passing the two tests in (b) are determined to be occluded and culled. Other objects are rendered at their proper levels

of detail to produce the final image.

The above framework is tailored for implementation on currently available commercial hardware graphics platforms. It should be noted that our occlusion representation and techniques for overlap and depth tests are not restricted to this particular two-pass framework in any way. Straightforward modifications to the above framework can yield a progressive algorithm or algorithms with more than two passes, if they are favored by the underlying hardware graphics system. Also, the framework implies a software system running on "black-box" graphics hardware that performs rendering, but the occlusion representations and culling algorithms may very well be implemented in hardware, if occlusion culling is to be supported in hardware directly.

Chapter 4

Hierarchical Occlusion Maps

Hierarchical occlusion maps are an important part of our occlusion representation. They form an image pyramid on which our occlusion culling algorithms are based. In this chapter, we define what occlusion maps are and discuss how a base map, and then the map hierarchy, are built.

4.1 Occlusion Maps

When an opaque object is projected to the screen, the region covered by its projection is made opaque. The screen opacity means another object will not be visible if it lies farther away from the viewer and projects onto the same region. Similarly, when a translucent object is projected, the region of its projection is made semi-opaque. More precisely, the screen opacity is defined by a real function $\rho(x,y) : R^2 \rightarrow [0,1]$, where values 0 and 1 indicates complete transparency and opacity, respectively. Further, we define the opacity, ρ_R, of a region on the screen, R, to be the average opacity of the region, i.e.,

$$\rho_R = \frac{1}{A_R} \int\int_R \rho(x,y)dxdy$$

where A_R is the area of the region. If all objects are completely opaque, then $\rho(x,y)$ has values of either 0 or 1. Let the opaque sub-regions within R be R_o, then

$$\rho(x,y) = \begin{cases} 1 & \text{if } (x,y) \in R_o \\ 0 & \text{if } (x,y) \in R - R_o \end{cases}$$

It follows immediately that $\rho_R = A_{R_o}/A_R$, with A_{R_o} being the area of R_o. Thus,

when all objects are opaque, the opacity of a region is the proportion of the sub-region covered by the objects' projections.

An *occlusion map* is a gray-scale image that corresponds to a uniform subdivision of the screen into rectangular regions. Each pixel in the occlusion map represents one of the regions, recording its opacity. An occlusion map is simply an opacity map used for occlusion culling.

In raster graphics, the screen is always uniformly subdivided into a 2-D array of screen pixels (small rectangular regions) to form the screen image. The opacity of a screen pixel can be computed using the definition above. In particular, the pixel opacity is the same as *pixel coverage* when all objects are opaque. There is a correspondence between an occlusion map and the screen image, both being subdivisions of the screen. For convenience, we always use occlusion maps whose pixels correspond to a $m \times n$ block of screen pixels, m and n being positive integers. It is obvious that the opacity of a pixel in such an occlusion map is the average of the opacities of the screen pixels to which it corresponds.

An occlusion map can be generated by rendering the objects at the same resolution as the map. This is very important since it means map generation can be performed quickly and efficiently by utilizing existing graphics hardware. To generate an occlusion map, the objects are rendered in white color (intensity 1.0), as pure geometry (i.e. with no shading or texturing), and with box-filtered anti-aliasing. Ideal anti-aliasing requires identifying all the primitives visible through each pixel and then filtering them to band-limit the light function sampled by the screen image. In practice, however, complex filters are computationally expensive for real-time applications, and we often employ a simple box filter which spans a pixel. Box-filtered anti-aliasing calculates the percentage of a pixel covered by a primitive visible through the pixel. The contribution of the primitive to the pixel is then computed as the primitive color attenuated by the percentage of coverage. More clearly, suppose there are N primitives visible through a certain pixel, each with visible area A_k (normalized to the area of the pixel) and color C_k, $0 \leq k \leq N-1$. There are normally multiple channels of colors, here C_k refers to any one of the them. The pixel color C_p, then, can be computed as $C_p = \sum_0^{N-1} A_k C_k$. Therefore, if $C_k = 1$ for $0 \leq k \leq N-1$, we have $C_p = \sum_0^{N-1} A_k$. That is, when primitives are rendered in white color, the color of the pixels in the final image reflect the total coverage of all the primitives. Note that this result holds only for the box filter, by which a primitive's contribution is weighted only by its area, regardless of its sub-pixel location.

Figure 4.1: One view of a scene and its corresponding occlusion map.

Computing the exact percentage of coverage is an expensive operation. In practice, anti-aliasing algorithms such as supersampling [FDFH90] and the A-buffer [Car84] seek to approximate pixel coverage instead of computing it exactly. For example, if a 4×4 sub-pixel mask is used, there are only 16 possible values for A_k, and at most 16 primitives can make contributions ($N < 16$). In fact, the rendering described above produces an occlusion map even if there is no anti-aliasing (i.e. $N = 1, A_0 = 1$), in which case each map pixel has an opacity value of either 0 or 1. Obviously, for some pixels (along the boundary between the primitives and the background) opacity is overestimated and for some others it is underestimated; the lower the resolution, the greater the error. Thus, if an occlusion map is to be built by rendering primitives without anti-aliasing (or with only highly approximate anti-aliasing), it must have enough resolution to avoid excessive errors in opacity values.

Figure 4.1 shows a rendered image and a corresponding occlusion map of the same resolution. Intuitively, the screen projections of the small polygons that comprise the teapots are merged into the occlusion map. That is, the occlusion map represents the cumulative projection of small primitives from multiple objects. In order for any object to be occluded by the three teapots, its screen projection has to lie within the cumulative projection. An occlusion map represents a *fusion* of occluders in image space.

If not rendered directly as proposed above, the occlusion map can be obtained as a by-product of normal rendering. For example, it can be the α*-channel* [PD84] of a rendered image if *alpha* values are properly set for the primitives. However, this method results in maps that are at the same resolution of the screen image (e.g. 1280×1024) which is usually too high to process in real time. On the other

hand, if we render a map separately, map generation is no longer a by-product and becomes pure overhead. Despite this, the latter approach proves much more flexible, and the overhead can be well controlled through careful selection of occluders and occlusion-preserving simplification (see Section 7.2).

Note that in general the depth of primitives need not be considered when rendering an occlusion map (unless we need depth for other purposes). None of lighting, color interpolation, or texturing are performed, either, and only one "color" channel is needed. Furthermore, the rendering resolution is usually much coarser than that of the scene image. Therefore, occlusion map rendering is much less expensive than general-purpose rendering.

4.2 Hierarchical Occlusion Maps (HOM)

From one occlusion map, we can build a pyramid or hierarchy of occlusion maps by recursively averaging blocks of occlusion map pixels. We follow the convention that the original, i.e. *highest* resolution occlusion map of a hierarchy is at *level* 0, and lower-resolution maps have larger level numbers. We refer to a map with a large level number (and thus low resolution) as high-level maps.

Let the resolution of the n-th level occlusion map be $X_n \times Y_n$. By averaging $p \times q$ blocks of pixels in the n-th map, we generate the $(n+1)$-th level map whose resolution is $\frac{X_n}{p} \times \frac{Y_n}{q}$. That is, higher level maps have lower resolution. Clearly, the pixels in the pyramid form a pixel tree in which each pixel, except for the leaves, has $p \times q$ *child* pixels. All the pixels in the sub-tree rooted at a pixel are called the pixel's *descendants*.

P and q can be arbitrary; however, in practice we favor 2×2 blocks to take advantage of hardware acceleration (see the following section). Figure 4.2 illustrates hierarchy construction with 2×2 blocks. Figure 4.3 shows another example of an occlusion map pyramid and the numbering of levels; the pyramid is created by recursively averaging over 2×2 blocks of pixels. The outlined square marks the correspondence of one top-level pixel to pixels in the other levels. The normal rendering to which the hierarchy corresponds is shown in the upper-right corner.

For each frame, the original (0-th level), and finest occlusion map is obtained by rendering the occluders into an image, as described earlier. The occlusion map hierarchy is then built by recursive filtering, which stops after reaching some minimal map resolution (e.g. 4×4).

Figure 4.2: An example of hierarchical occlusion maps

It is important to note that the resolution at which occluders are rendered (i.e. the resolution of the original map) need not match that of the scene image. As mentioned in the previous section, the latter is often large, making generation of the image pyramid rather expensive in real-time. Using a lower image resolution may lead to inaccuracies in occlusion culling near the silhouette of the occluders, but this is a typical trade-off between precision and speed. Furthermore, if fast anti-aliasing is available for occluder rendering, the original occlusion map has more accuracy than its apparent resolution.

In the occlusion map pyramid, pixels in a higher-level (lower-resolution) map correspond to larger regions on the screen. So as we go from higher to lower resolution, we get a more global, higher-level view of the available occlusion. The hierarchy actually represents cumulative occlusion at *multiple resolutions*. Its application in occlusion culling is rooted in the fact that it allows for examination/estimation of occlusion at different resolutions.

4.3 Fast Construction of the Hierarchy

When filtering is performed on 2×2 blocks of pixels, hierarchy construction can be accelerated by graphics hardware that supports bilinear interpolation of texture maps. The averaging operator for 2×2 blocks is actually a special case of bilinear interpolation. More precisely, the bilinear interpolation of four scalars or vectors $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ is:

$$(1-\alpha)(1-\beta)\mathbf{v}_0 + \alpha(1-\beta)\mathbf{v}_1 + \alpha\beta\mathbf{v}_2 + (1-\alpha)\beta\mathbf{v}_3,$$

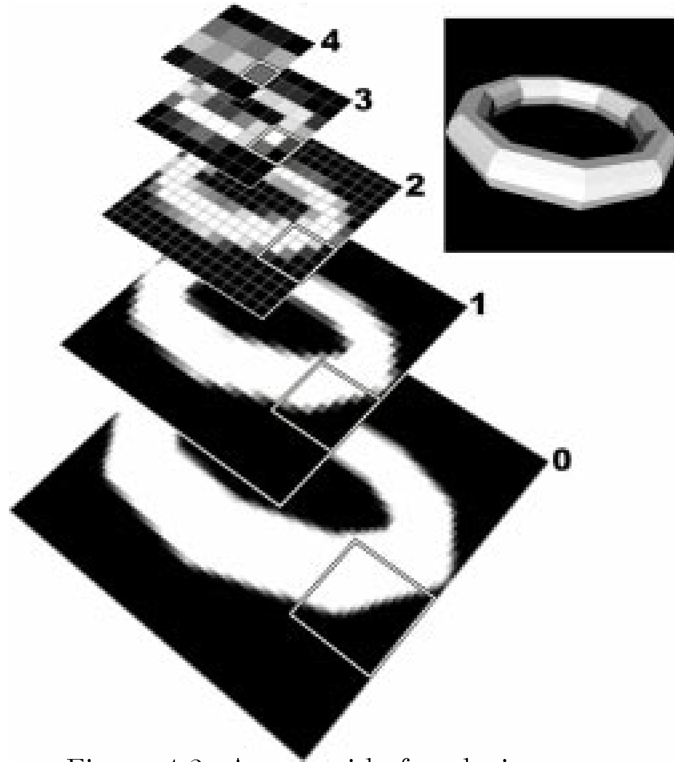


Figure 4.3: A pyramid of occlusion maps

where $0 \leq \alpha \leq 1$, $0 \leq \beta \leq 1$ are the weights. If we set $\alpha = \beta = 0.5$ then the formula produces the average of the four values. By carefully assigning texture coordinates to ensure that $\alpha = \beta = 0.5$, we can filter a $2N \times 2N$ occlusion map to $N \times N$ by drawing a two dimensional square of size $N \times N$ with the $2N \times 2N$ occlusion map as a texture. Figure 4.4 illustrates this process for graphics architectures with a conventional framebuffer and texture memory. The process is repeated to generate all the occlusion maps in the pyramid.

The graphics hardware typically needs some setup time for the required operations. When the size of the map to be filtered is relatively small, setup time may dominate the computation. In such cases, the use of texture mapping hardware may actually slow down the construction of occlusion maps rather than accelerating it, which means we can do better by using the host CPU. For this reason, we take advantage of hardware texturing mapping only at levels where maps are large enough. Then at a certain level in the pyramid where the map drops below a threshold size, we turn to software. The break-even point between hardware and software computation, represented by the threshold map size, varies with different graphics systems.

[BM96] presents a technique for generating mipmaps using a hardware accumulation buffer. We did not use this method because the accumulation buffer is less

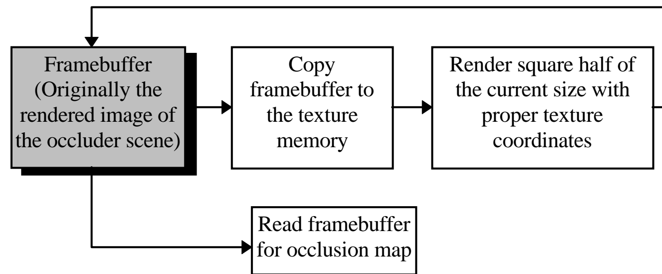


Figure 4.4: Using texture-mapping hardware to accelerate the construction the occlusion map hierarchy.

commonly supported in hardware graphics systems than is texture mapping.

4.4 Properties of Hierarchical Occlusion Maps

Below we highlight the properties of occlusion maps and occlusion map pyramids that facilitate occlusion culling.

1. **Occluder fusion:** Occlusion maps represent the fusion of small and/or disjoint occluders. They can be generated quickly and efficiently using conventional graphics hardware. As is stated in Chapter 2, an efficient representation of cumulative occlusion is crucial to a general-purpose occlusion culling algorithm.

   Occlusion fusion has another aspect regarding occlusion from semi-transparent surfaces: several of these can combine to become practically completely opaque, and thus provide good occlusion. We can easily handle the fusion of semi-transparent surfaces by using a proper blending function to accumulate opacity when rendering the occluders to generate the original occlusion map,

2. **Generality:** No assumptions are made regarding the shape, size, or type of the occluders. Any object that can be rendered can serve as an occluder, whose occlusion can be combined with that of any other occluders.

3. **Fast Construction of the Hierarchy:** When filtering is performed on 2×2 blocks of pixels, construction of the occlusion map hierarchy is readily supported by graphics hardware with bilinear texture mapping.

4. **The averaging operator:** The averaging operator used in building the occlusion map pyramid is responsible for all the pyramid's desirable features in

occlusion culling. These will be discussed in detail throughout the next chapter. In particular, the pyramid supports:

- **Hierarchical overlap test:** The hierarchy allows for fast 2-D overlap tests between the occluders' cumulative projection and a potential occludee, based on the fact that a pixel is completely opaque if and only if all its descendant pixels are completely opaque.
- **High-level opacity estimation:** The opacity values in a low-resolution occlusion map give an estimate of the opacity values in higher-resolution maps. This is because the averaging operator preserves much information so that one can get a "feel" of opacity distributions by looking at low-resolution maps.

Chapter 5

Overlap Tests Using Hierarchical Occlusion Maps

In this chapter, we shall discuss algorithms for verifying the *overlap* condition of occlusion, i.e. the screen projection of an object (a potential occludee) has to lie within, i.e., be completely overlapped by, the cumulative projection of the occluders. This verification is called the *overlap test*, which returns true if the condition is satisfied and false if not. If the overlap test returns false, the tested object is considered visible and subsequently rendered; if it returns true, the depth test (next chapter) is then performed to see if the object is indeed occluded.

Since in our case the cumulative projection is represented by occlusion maps, the overlap test is reduced to testing whether the occlusion-map pixels intersected or overlapped by the projection of the potential occludee are fully opaque.

5.1 Projection Estimation

For overlap tests we first have to compute the screen projection of a tested object, i.e. the potential occludee. More precisely, we have to identify the pixels in an occlusion map that are intersected or overlapped by the projection of the potential occludee. This is, by definition, a scan-conversion of the potential occludee at the resolution of the occlusion map. However, actual scan-conversion of the object is usually not an option since it is computationally expensive. Instead, we seek to compute a simple *overestimation* of the object's projection, which fully encloses the exact projection. The overestimation guarantees *conservative* overlap tests. That is, if the overestimated projection touches only opaque occlusion map pixels, so must

the actual projection. The projection of an object's bounding box always bounds the projection of the object, and thus qualifies as such an overestimation. The cost of scan-converting the box faces (which project to general convex triangles), however, is still too high, especially for software implementations.

The estimation we use is the screen-space bounding rectangle of the projection of the bounding box. Identifying the pixels touched by the rectangle is very simple—or in other words, a rectangle can be trivially scan-converted. Computing the bounding rectangle involves only projecting the eight corner vertices to the screen and finding their extent. For an axis-aligned bounding box, this process can be accelerated by exploiting the fact that projecting the eight corners shares much common computation. (More details are in Appendix A).

5.2 Hierarchical Overlap Tests

As described in the preceding chapter, the original, level-0 occlusion map is obtained by rendering the occluders. The overlap test is to determine whether the bounding rectangle of the potential occludee is enclosed by the cumulative projection of the occluders—i.e. whether the pixels touched by the rectangle in the original, level-0 occlusion map are fully opaque (opacity 1.0). If so, the test object itself must also be enclosed by the cumulative projection, and it thus successfully passes the overlap test.

A simple way of performing the above test is to traverse the relevant pixels in the level-0 occlusion map one by one, checking the opacity values. With the occlusion map pyramid, however, we can perform faster hierarchical overlap tests. We will initially describe a straightforward basic algorithm for hierarchical overlap tests. Then, we will improve its performance and introduce special features by generalizing the basic algorithm.

5.2.1 The Basic Algorithm

Let us assume for the moment that we have a complete occlusion map pyramid with a highest-level map of resolution 1×1.[1] The hierarchical overlap test is a traversal of a portion of the occlusion map pyramid. It begins at the coarsest (i.e. highest) level

[1] Recall that we stop the construction of the pyramid at some minimal resolution, which is not necessarily 1×1.

where a pixel, P_0, fully encloses the bounding rectangle.[2] In Figure 4.3, the square box on the level-4 map shows such a pixel, and the boxes at the other levels highlight the descendant pixels that correspond to P_0. If P_0 is fully opaque (with opacity value 1.0), then we know immediately that the rectangle falls into an fully opaque area in the cumulative projection, and the overlap test returns true. This is because the averaging operator (with which the pyramid is built) guarantees that if a pixel in a coarse map has opacity 1.0, then all the descendant pixels in the finer maps, and in particular those in the original map, must also have opacity (1.0). In many cases, the test on P_0 is the only opacity check we have to perform before returning true.

Also because of the averaging operator, all descendant pixels must have opacity 0.0 if P_0 is 0.0. In this case, we conclude that the rectangle lies in a fully transparent area and that the overlap test should return false.

If P_0 is semi-opaque, i.e. its opacity being neither 1.0 nor 0.0, we descend to the next finer level in the pyramid and find which of P_0's child pixels are touched by the bounding rectangle. If all the children have opacity 1.0, then the overlap test returns true. If at least one of them has opacity 0.0, the test returns false. Otherwise, we recursively descend further into the finer levels for further checking. If we reach the original occlusion map (level-0 in the pyramid) and find any pixels touched by the rectangle have opacity 0, the overlap test returns false.

Note that in the recursive overlap test through the pyramid, at some level the bounding rectangle starts to enclose entire occlusion map pixels. For such an enclosed pixel, all its corresponding pixels in the finer levels in the pyramid must also be enclosed by the bounding rectangle. If the pixel does not have full opacity (1.0), the opacity of some of its corresponding pixels (particularly those in the level-0) must be less than 1.0. It follows that some level-0 pixels covered by the bounding rectangle do not have opacity 1.0, and we immediately conclude that the overlap test should return false. In other words, only when a semi-opaque pixel is partially covered by the bounding rectangle need we recurse to its finer-level corresponding pixels; any fully-covered semi-opaque pixel causes the overlap test to return false immediately.

By beginning at a level where a pixel (P_0) fully encloses the bounding rectangle, we can potentially determine the result of the overlap test by checking the opacity of P_0 only. However, such a beginning level relies much on the position of the bounding

[2] More precisely, what we mean is that the screen region to which the map pixel corresponds encloses the rectangle. For brevity, however, we will often use the map pixel to refer to the screen region to which it corresponds.

rectangle. For example, in a pyramid comprised of maps of resolutions $2^k \times 2^k, k = 0, 1, 2, ...$, a bounding rectangle that covers the center of the screen can only be fully covered by the single pixel in the 1×1 map. This means we have always to start with the 1×1 map for such rectangles. But very coarse maps (like 1×1, 2×2) seldom have fully opaque pixels, meaning we almost always have to do recursive tests. Consequently, it is frequently a waste of time to start at very coarse levels. Moreover, each recursion has an overhead, and the total overhead tends to increase as we begin the test at higher levels. On the other hand, if we start at a level that is too fine, we potentially increase the number of opacity checks we need to perform. In practice, we have found it a good compromise to begin at the finest level where the size of the pixels is greater the rectangle's smaller dimension. Intuitively, this criterion is biased towards beginning at fine levels to avoid recursions, at the expense of potentially more tests.

For simplicity in our description of the hierarchy overlap test, we have used opacity values 1.0 and 0.0 to indicate fully opaque or transparent pixels, respectively. However, the unique features of the occlusion map pyramid are exploited when we have more general and flexible definitions of full opacity or transparency. In the next section, we introduce these definitions and discuss their application to the overlap tests.

5.3 Early Terminations in Overlap Tests

The occlusion map pyramid, built as a result of recursive low-pass filtering, represents the occluders' cumulative projection at multiple scales. Thus, the hierarchical overlap test is an examination of the cumulative projection in different (coarse-to-fine) scales. The previous section describes general recursive overlap tests through the map pyramid, but often the recursive evaluation steps can be omitted, and test results returned, before the basic algorithm is fully carried out. This happens when a decision can already be made without further descending into finer levels of the pyramid, in which case the test returns to the preceding level of recursion or exits entirely. We discuss three types of early terminations: (a) conservative rejection, which makes the overlap test return false immediately; (b) aggressive approximate culling, which regards high-opacity pixels as fully opaque and prevents further recursion; and (c) predictive rejection, in which case the overlap test returns false knowing that the test has to fail somewhere at a finer level in the pyramid.

5.3.1 Conservative Rejection

Conservative rejection pessimistically terminates overlap tests and returns false. If a pixel touched by the bounding rectangle has quite low opacity, we know that, even if we descend to finer-level maps, there is only small probability that we will find many high opacity pixels—if there were, the pixel being examined (whose opacity is their average) would have high opacity. So we may decide to terminate the overlap test immediately and return false. Recall that the hierarchical overlap test we described in the preceding section terminates when a pixel is transparent (opacity 0.0). Now we are actually regarding the pixel with quite low opacity as being completely transparent, even though its opacity is not 0.0. The "quite-low" opacity is defined by the *transparency threshold*, T_t, below which a pixel is considered *fully* transparent. In other words, full transparency has become an opacity range $[0, T_t)$, instead of a single value (0.0).

Intuitively, a low-opacity pixel in a coarse occlusion map can be the result of low-pass filtering an original (level-0) occlusion map with high-opacity pixels scattered around in a majority of black (0- or low-opacity) pixels. Such an original map means overlap tests will tend to fail, since it is hard for a bounding rectangle to touch only the pixels with full opacity. So, if we find a coarse-level pixel with low opacity, we may choose to assume that it is a result of such level-0 distribution and subsequently terminate the overlap test.

5.3.2 Aggressive Approximate Culling

Consider an original (level-0) occlusion map with small holes of black pixels scattered in a majority of white pixels. An occlusion map with this characteristics often results from a set of small, irregular occluders, such as the leaves of a group of trees or the complex mechanical parts in a CAD model. Due to the scattered-around distribution of the black pixels, most bounding rectangles will likely cover some of them, which causes the overlap test to return false. However, the viewer normally does not expect to see much through the small holes anyway. So, if we ignore the holes and discard objects visible only through these holes, we hopefully will not introduce too many visual artifacts. Ignoring small holes to cull away barely-visible objects is called *aggressive approximate culling*, which is achieved by cutting off further recursions in the overlap test based on the *opacity threshold*.

Approximate culling is considered aggressive because contrary to the conservative

acceptance introduced earlier, approximate culling is not conservative, i.e. it can cull away visible geometry and thus introduce visual artifacts. However, the artifacts can be well controlled while a larger portion of the model can be culled away.

Recall that in hierarchical overlap tests, we recurse into the next finer (lower) level for further opacity checking only when an examined pixel is not fully opaque (opacity value 1.0). We now redefine "full opacity" by introducing an opacity threshold, T_Ω: any pixel with opacity greater than T_Ω is considered fully opaque, so that full opacity corresponds to a range of opacity values $(T_\Omega, 1]$. Therefore, the overlap test will not recurse into the next finer level when the pixel opacity is *high enough* (i.e. $> T_\Omega$).

A pixel P in a coarse occlusion map in the pyramid corresponds to a block of pixels in the original, level-0 map; the opacity of P is the average opacity of all pixels in the block. If a small percentage of these level-0 pixels are low in opacity with the others being high, P's opacity can still be fairly high due to averaging. Now, if P's opacity is high enough ($> T_\Omega$), the overlap test will return true from P to the previous level in the hierarchical, *without* descending further into the finer levels. In effect, the low-opacity pixels in fine-level maps, i.e. the holes in the cumulative projection, are ignored.

When the occluders are rendered without any anti-aliasing, i.e. when the level-0 pixels are either black (opacity 0.0) or white (opacity 1.0), the opacity threshold determines how many 0-opacity pixels are allowed while the average opacity over a certain area remains high enough. Put another way, T_Ω actually bounds the size of the biggest negligible hole. This will become clearer when we discuss the derivation of the opacity thresholds from the size of negligible holes in the next section.

From the signal processing point of view, the process of low-pass filtering (averaging) with which the pyramid is built suppress the high-frequency noise (the small holes); the higher-level the map, the more the suppression. At a certain level, the noise is reduced enough to be ignored by thresholding with T_Ω. Intuitively, as we go from the original map to coarser ones, the holes dissolve into the surrounding high-opacity pixels; therefore, at a high level we can barely see them. This is illustrated in Figure 5.1 by a hierarchy of occlusion maps that corresponds to one view of a tree (shown at the top). The level number of the maps are marked on the bottom, and the rectangles highlight the region with holes among the leaves.

It should be noted that in some cases approximate culling may result in easily noticeable artifacts. For example, consider a bright object visible only through small holes (e.g. the sun shining through holes among the leaves of a tree) and thus culled

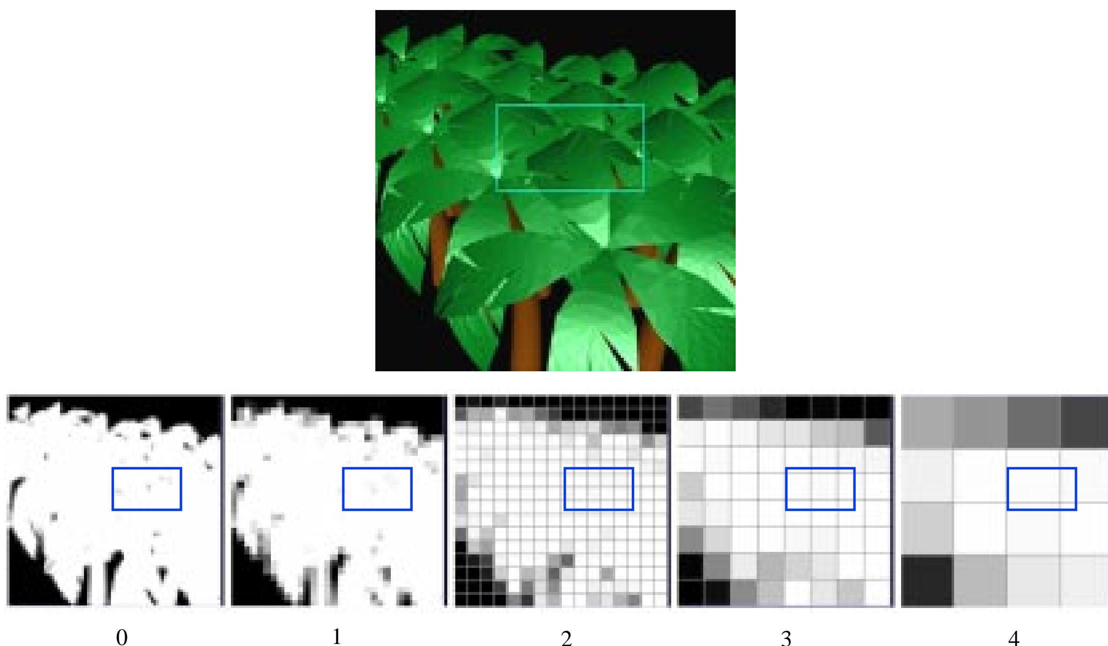


Figure 5.1: Approximate culling

away. As the viewer zooms closer to a hole, the hole becomes larger and no longer ignored, at which point the bright object suddenly pops in. In order to avoid this problem, we can compute the color contrast between the test object and the occluders, and relate the opacity threshold to the contrast—the opacity threshold should be high when the contrast is strong, effectively preventing a hole from being ignored when the tested object is relatively bright.

Objects sticking a little outside the "cumulative silhouette" (i.e. the boundary of the cumulative projection of the occluders) can also be removed due to approximate culling. A coarse-level pixel lying mostly inside, and a little outside the cumulative projection of the occluders can have an average opacity greater than T_{Ω}. The result is that black pixels along the silhouette can be ignored, which in effect extends the cumulative projection of the occluders. When an object that is approximately culled-away in the preceding frame pops in around the cumulative silhouette due to motion of the viewer, the visual artifact is more noticeable than when a object pops in through a hole. This popping effect is similar to the artifacts caused by switching level-of-details of the objects, and can similarly alleviated by making the newly visible object fade in instead of popping in.

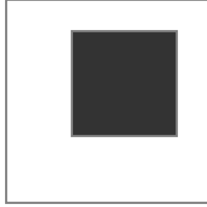
5.3.3 Computing the Thresholds

Different levels of the occlusion map pyramid must have different opacity thresholds in order to consistently bound the size of the holes we consider as negligible. Transparency thresholds must also vary between the levels to bound the amount of occlusion we ignore. In this section, we discuss the computation of the thresholds for the pyramid. We focus on the opacity thresholds; the transparency thresholds can be computed in a very similar fashion.

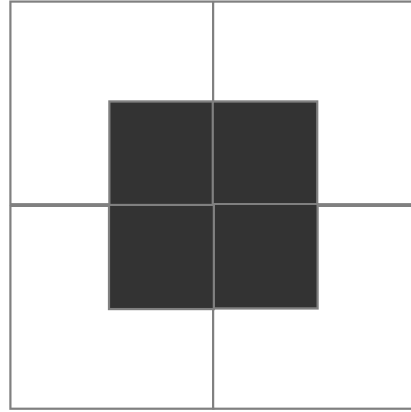
5.3.3.1 Opacity Thresholds

First, let's examine how the opacity threshold and the size of negligible holes are related. We assume the opacity of a screen pixel is either 1 or 0; this assumption is not inherently necessary but it simplifies our discussion. Also, assume the pyramid is built by averaging 2×2 blocks of pixels. Remember that each pixel in an occlusion map corresponds to a block of *screen* pixels, its opacity being their average. Suppose that at level-k in the pyramid, a map pixel represents a $m_k \times m_k$ block of screen pixels, and the opacity threshold at level-k is T_{Ω_k}. Clearly, at most $\mathcal{H}_k = (1 - T_{\Omega_k}) {m_k}^2$ screen pixels in the block can have opacity 0 in order for the map pixel to be considered fully opaque. So the question is how big a hole these 0-opacity pixels can possibly make. In other words, what distribution of these 0-opacity screen pixels can create a hole through which the viewer can see most clearly? Given a fixed number of black pixels that comprise a hole, we can glue them together in various ways to make the hole; however, it is intuitively true that the viewer can see most through the hole whose aspect ratio is close to 1. So, the worst-case hole that can correspond to a map pixel with opacity greater than T_{Ω_k} is approximately a square with $\mathcal{H}_k$ 0-opacity screen pixels. The biggest hole on the screen, then, is created when a 2×2 block of map pixels, each with opacity T_{Ω_k}, have their 0-opacity screen pixels concentrated around the common corner of the screen pixel blocks they correspond to. This distribution is illustrated in Figure 5.2. Consequently, the biggest negligible hole implied by threshold T_{Ω_k} has $4\mathcal{H}_k$ 0-opacity pixels.

We use the "$\mathcal{L} - k-$hole" constraint to specify the size of negligible holes. The constraint is defined as follows: for a given level, k, in the occlusion map pyramid, a set of 0-opacity pixels (i.e. a hole) on the screen can be ignored if and only if *any* $m_k \times m_k$ "window" on the screen contain no more than $\mathcal{L}$, $\mathcal{L} < m_k \times m_k$, 0-opacity pixels from the set. It follows from the above discussion on worst-case distributions



With an opacity threshold of 0.75, this is the biggest square hole an occlusion map pixel (whose opacity is above the opacity threshold) can correspond to.



The biggest possible negligible square hole on the screen under an opacity threshold of 0.75.

Figure 5.2: Biggest possible square holes given an opacity threshold

that the number of 0-opacity pixels a level-k map pixel can correspond to is at most $\mathcal{H}_k = \mathcal{L}/4$. Thus, the opacity threshold should be

$$\begin{aligned} T_{\Omega k} &= 1 - \mathcal{H}_k / {m_k}^2 \\ &= 1 - \mathcal{L}/4{m_k}^2 \end{aligned}$$

Also, the constraint implies that no negligible holes contain more than L 0-opacity screen pixels.

The $\mathcal{L} - k-$hole constraint determines not only $T_{\Omega k}$, but the opacity thresholds for all the levels in the pyramid. We turn now to derive the opacity threshold for the $k+1$-th level, $T_{\Omega_{k+1}}$, from T_{Ω_k}. Clearly, we have $m_{k+1} = 2m_k$. It appears that since the level-$k+1$ map pixels correspond to large regions on the screen, each of them can have more than $\mathcal{H}_k$ 0-opacity pixels while still satisfying the $\mathcal{L} - k-$hole constraint. However, this is not true with an arbitrary distribution of 0-opacity pixels. If pixels in the level-$k+1$ map are allowed to contain $\mathcal{H}_{k+1} > \mathcal{H}_k$ 0-opacity screen pixels, then there will be holes with as many 0-opacity pixels as $4\mathcal{H}_{k+1} > 4\mathcal{H}_k = \mathcal{L}$, due to the same kind of concentration of 0-opacity pixels as shown in Figure 5.2. So, in a $m_k \times m_k$ area containing this hole, the $\mathcal{L} - k-$hole constraint is clearly violated. Therefore, to guarantee the satisfaction of the constraint, we must have $\mathcal{H}_{k+1} = \mathcal{H}_k$; and then $T_{\Omega_{k+1}}$ is computed as:

$$
\begin{aligned}
T_{\Omega_{k+1}} &= 1 - \mathcal{H}_{k+1}/{m_{k+1}}^2 \\
&= 1 - \mathcal{H}_k/(2m_k)^2 \\
&= 1 - (1 - T_{\Omega_k})/4
\end{aligned}
$$

In practice, we usually specify the $\mathcal{L} - 0-$hole constraint. So by repeatedly applying the above formula, we can compute the opacity thresholds for all the pyramid levels.

The above derivation is based on the worst-case distribution of 0-opacity screen pixels which does not actually occur very often. In practice, we may want to assume more favorable distributions, at the risk of ignoring bigger holes than those allowed by the $\mathcal{L} - k-$hole constraint in rare cases. We introduce a *distribution factor*, $\mathcal{D}$, to model how favorable we assume the distribution to be: $\mathcal{H}_{k+1} = \mathcal{D}\mathcal{H}_k, 1 \leq \mathcal{D} \leq 4$. At level k, where we define the $\mathcal{L} - k-$hole constraint, D specifies the degree to which we deviate from the worst case: $H_k = \frac{\mathcal{D}}{4}\mathcal{L}$. Thus, the level-$k$ opacity threshold becomes:

$$
\begin{aligned}
T_{\Omega k} &= 1 - \mathcal{H}_k/{m_k}^2 \\
&= 1 - D\mathcal{L}/4{m_k}^2
\end{aligned}
$$

Between the neighboring levels, k and $k+1$, $\mathcal{D}$ indicates how many more 0-opacity screen pixels a level-$k+1$ map pixel can correspond to than level-k. It follows immediately:

$$
\begin{aligned}
T_{\Omega_{k+1}} &= 1 - \mathcal{H}_{k+1}/{m_{k+1}}^2 \\
&= 1 - \mathcal{D}\mathcal{H}_k/(2m_k)^2 \\
&= 1 - \mathcal{D}(1 - T_{\Omega_k})/4
\end{aligned}
$$

The worst case distribution, as discussed above, corresponds to $\mathcal{D} = 1$, i.e. $\mathcal{H}_{k+1} = \mathcal{H}_k$. Since a level-$k+1$ pixel represent 4 times as much screen area as a level-k pixel, $\mathcal{D} \leq 4$. We have $\mathcal{D} = 4$, and thus $T_{\Omega_{k+1}} = T_{\Omega_k}$, when the 0-opacity pixels exhibit an uniform random distribution on the screen, in which case a level-$k+1$ pixel can indeed correspond to 4 times as many 0-opacity pixels as a level-k pixel, while still conforming to the $\mathcal{L} - k-$hole constraint. In our implementation we usually set $\mathcal{D} = 2$.

To sum up, given $\mathcal{L}$ and k in the $\mathcal{L} - k-$hole constraint, and the distribution

factor $\mathcal{D}$, we can compute the opacity thresholds for pyramid levels higher than and including k. With $k = 0$, we can compute the thresholds for all the levels in the pyramid.

As a concrete example, suppose the screen image is 1024×1024 and the level-0 map is 128×128. A pixel in the level-0 map, then, corresponds to a 8×8 block of pixels in the screen image ($m_0 = 8$). Assume we consider a 3×3 block of 0-opacity screen pixels in a 8×8 block to be an negligible hole (i.e. $\mathcal{L} = 3^2$), then the opacity threshold for the map is $T_{\Omega_0} = 1 - (3^2)/(8^2) = 0.86$. One level up the pyramid, each map pixel corresponding to 16×16 screen pixels; assuming $\mathcal{D} = 2$, we have $T_{\Omega_1} = 1 - \mathcal{D}(1 - T_{\Omega_0})/4 = 0.93$. And so on.

5.3.3.2 Transparency Thresholds

Computing the transparency threshold is very similar to computing the opacity thresholds. The latter is driven by bounding the size of negligible holes (clusters of 0-opacity pixels), while the former by bounding the size of clusters of 1-opacity pixels. Proper opacity thresholds avoid ignoring oversized holes; proper transparency thresholds avoid losing too much occlusion. Without going into the details again, we give the formula for computing (with the distribution factor $\mathcal{D}$) the transparency threshold of level-$k+1$ from that of level-k,

$$T_{t_{k+1}} = \mathcal{D}T_{t_k}/4$$

5.3.4 Predictive Rejection

In presenting the basic algorithm for hierarchical overlap tests, we have observed that if the opacity is below 1.0 for a pixel completely covered by the bounding rectangle, some of its descendant pixels at the 0-th level must also have less-than-1.0 opacity. Thus, the overlap test can return false immediately since it must fail in the end. This rule can now be generalized to incorporate the opacity thresholds.

Suppose we have a level-k map pixel, P_k, with opacity, Ω_{p_k}, that is completely covered by the bounding rectangle. If Ω_{p_k} is less than the opacity threshold of level-m, $T_{\Omega m}$ ($m < k$), then we know that the opacity of some level-0 pixels P_k corresponds to must be less than $T_{\Omega m}$. Since if not, we should have $\Omega_{p_k} > T_{\Omega m}$ because Ω_{p_k} is the average of the opacities of the corresponding level-m pixels. The overlap test has to fail somewhere at level-m, when we descend into that level. Anticipating this, we

may terminate the overlap test immediately and return false.

Similarly, if Ω_{p_k} is less than the transparency threshold, T_{tm}, of a finer level, m, $m < k$, then the opacity of some level-m pixels that P_k corresponds to must be less than T_{tm}; since if not, we should have $\Omega_{p_k} > T_{tm}$ due to averaging. The overlap test must return false at one of these level-m pixels whose opacities are below the level-m transparency threshold. Predicting this, the overlap test returns false right away.

5.4 Summary

The use of the opacity and transparency thresholds in the overlap test actually signifies the notion of *levels of visibility*, which represents the continuum between being visible and being occluded. With thresholding, we can choose to regard "almost occluded" objects as non-visible (aggressive approximate culling), or "probably visible" objects as really visible (conservative rejection). Conservative rejection based on the transparency threshold is another application of the widely-used conservative culling strategy, while aggressive approximate culling is a new concept and a unique feature of our algorithm.

The overall process of overlap tests is summarized in pseudocode form in figure 5.3. In the code, we assume that the occlusion map hierarchy is stored in array `HOM[]`. Each map level has the it own `TransparencyThreshold` and `OpacityThreshold`. Pixel opacities at level `k` are retrieved by `calling HOM[k].getOpacity(x, y)`, with `(x, y)` being the coordinates of the pixel. Each pixel in the map is a structure with fields `x`, `y`, and `CompletelyInRect`. The former two are the coordinates, and `CompletelyInRect` is a flag indicating whether the pixel is entirely contained in the bounding rectangle.

The `CheckPixel` function checks the opacity of a pixel, descending into sub-pixels as necessary. The `OverlapTest` function performs the whole overlap test by calling `CheckPixel`. It returns `TRUE` if the bounding rectangle falls within completely opaque areas and `FALSE` otherwise.

```
OverlapTest(HOM, Level, BoundingRect)
{
    for each pixel, P, in HOM[HOM.HighestLevel]
            that overlaps BoundingRect
    {
        if (CheckPixel(HOM, HOM.HighestLevel,
                       P, BoundingRect) = FALSE)
            return FALSE;
    }
    return TRUE
}

CheckPixel(HOM, Level, Pixel, BoundingRect)
{
    Op = HOM[Level].getOpacity(Pixel.x, Pixel.y);
    Omin = HOM[0].OpacityThreshold;

    if (Op > HOM[Level].OpacityThreshold)
        return TRUE;
    else if (Level = 0)
        return FALSE;
    else if (Op < HOM[Level].TransparencyThreshold)
        return FALSE;
    else if (Op < Omin AND Pixel.CompletelyInRect = TRUE)
        return FALSE;
    else
    {
        Result = TRUE;
        for each sub-pixel, Sp, that overlaps BoundingRect
        {
            Result = Result AND
               CheckPixel(HOM, Level-1, Sp, BoundingRect);
            if Result = FALSE
                return FALSE;
        }
    }
    return TRUE;
}
```

Figure 5.3: Pseudo-code for overlap tests

Chapter 6

Resolving Depth

The complete enclosure of an object's screen projection by the opaque area in an occlusion map is a necessary but not sufficient condition for the object's being occluded. The object must also pass the depth test before we finally decide that it is not visible. In section 3.1 we have stated that the depth test is to decide "whether the potential occludee is behind the occluders". "Behind the occluders" is actually not a very precise description of the criterion by which an object passes the depth test. In this chapter, we will first define our depth tests in a more precise way by highlighting its difference from depth tests in conventional visibility algorithms. We will then present several different representations of the occluders' depth information which form the basis of the depth tests.

6.1 Depth Tests

The important difference between our depth tests and those in previous visibility algorithms (e.g. the Z-buffer algorithm and the hierarchical Z-buffer algorithm) is that our depth tests do not by themselves determine an object's visibility. Given an occluder, P,[1] and a potential occludee, Q, our depth test does not establish whether P hides Q, but rather whether Q hides *any* part of P that is visible if Q were removed. Clearly, that Q does not occlude any visible part of P is a necessary condition for P to occlude Q. This condition, only when combined with the other necessary condition that Q's projection lies entirely in P's, leads to the conclusion that P occludes Q. In

[1] Here we simplify the discussion by talking about one occluder. It can be the combination of multiple occluders

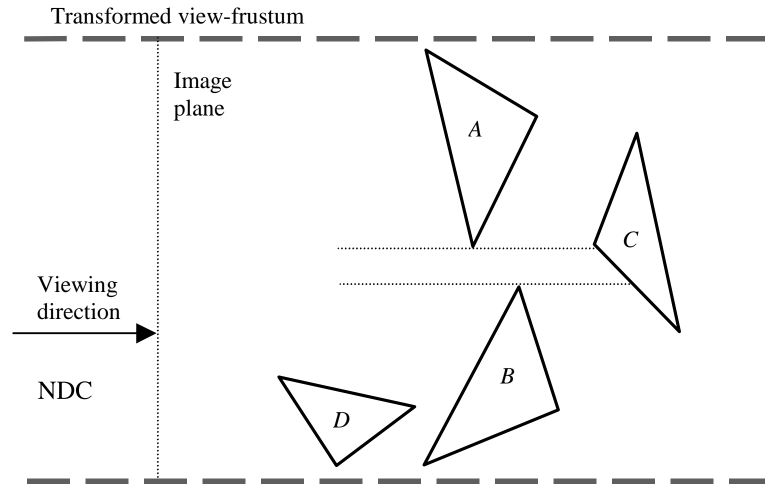


Figure 6.1: Definition of depth tests

Figure 6.1,[2] depth tests in the conventional Z-buffer algorithm would conclude that objects A and B fail to occlude object C. In our case, however, C will pass the depth test and is not considered visible as far as the depth test is concerned. This is because C does not occlude any part of A and B—more precisely, C does not occluder any visible part of A and B.

Another interpretation of our depth test is that it determines whether the tested object is far enough away so that it is behind the occluders and able to take advantage of the cumulative occlusion. In Figure 6.1, object C is at enough distance from the viewer to make use of the occlusion provided by A and B, and thus passes the depth test. Whether the cumulative occlusion of A and B is enough to completely hide C (i.e. if C "leaks" through the cumulative occlusion of A and B), however, is another question that must be answered separately by the overlap test. If the overlap test determines that the hole between A and B can be ignored, then C will be culled away; if not, C will be be regarded as visible. [3]

As a final example, consider the following extreme case. If we do not choose *any objects* as occluders, all the objects in the view-frustum will pass our depth tests because trivially none of them can be closer to the viewer than any occluder. The

[2]In this chapter, the figures are drawn in the normalized device coordinate (NDC) space instead of the world space, because we store depth information in NDC space. The NDC space is the world space after viewing and projection transformations.

[3]Although this implies that the depth test should precede the overlap test, they can actually be performed in any order.

overlap test, on the other hand, will correctly declare all the objects visible since all the occlusion map pixels have zero opacity.

To facilitate further discussion, we introduce the notion of *relevant occluders*. An occluder is relevant to a potential occludee if the (exact or estimated) screen projections of the two intersect. In other words, a relevant occluder is one that can possibly contribute occlusion to the potential occludee. So, an objects is considered occluded if and only if it is occluded by its relevant occluders. An example is shown in Figure 6.1, where object A and B are C's relevant occluders and D is not, supposing that we compute the exact projections of the objects. If we approximate the projections (e.g. using bounding rectangles as in the proceeding chapter and later in this chapter), D may also be C's relevant occluder.

The function of our depth tests, as described above, implies the kind of depth information required to support the depth tests. Basically, our depth representations should define, based on the depth of the occluders, a boundary beyond which an object cannot possibly be closer to the eye than any of its relevant occluders. The closer the boundary is to the viewer, the more objects will be behind it (and therefore passing the depth test). It is evident that the optimal (nearest-possible) boundary is made up of the visible surfaces of all the occluders. When the exact boundary is too expensive to obtain, we seek to compute a conservative approximation that is no nearer to the viewer at any point on the screen than is the exact boundary.

The important trade-off in depth representations is between the time required to build the representation and the "quality" of the boundary. In the rest of this chapter we will investigate this trade-off by discussing three different way of representing depth: with a single plane, with the depth estimation buffer, and with the no-background z-buffer. These representations are progressively more expensive to construct but more precise (less conservative) in approximating the optimal boundary.

6.2 A Single Plane

The simplest depth representation is a plane perpendicular to the viewing direction and through the farthest point of all the occluders (Figure 6.2). That is, the equation of the plane in NDC space is $Z = d$, d being the farthest depth of all the occluders. The depth test, then, is to determine whether an object is entirely on the far side of the plane, which is in turn determined by whether the nearest point on the object is on the far side of the plane. With the plane as a very conservative estimation of the

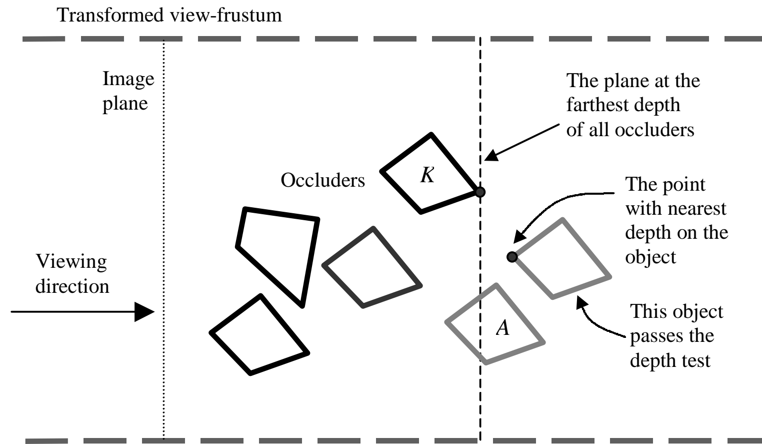


Figure 6.2: The single plane depth representation

depth of the occluders, objects that pass the depth test have a greater depth than *all* the occluders. Objects (partially) on the near side of the plane are regarded as visible as they could possibly be in front of some occluders.

This methods actually assumes that any occluder is relevant to any tested object. This makes it unnecessary to estimate the projection of the occluders and the test objects since we no longer have to find the relevant occluders. However, a single occluder with a large depth can put the plane at a great distance, making it impossible for most objects to pass the depth test. In Figure 6.2, occluder K is actually not relevant at all to object A; but due to the very conservative depth representation with the single plane, K becomes relevant to A. A is consequently not culled away because its nearest point is closer to the viewer than the farthest point of K.

Establishing the plane involves only finding the farthest of the farthest points of the individual occluders. Still, it can be computationally expensive to compute the farthest point on an object, e.g. when the object has many vertices. In practice, we always compute the farthest vertex of the bounding box as a conservative estimation. This involves transforming the vertices of the box into the NDC space. A fast method for transforming axis-aligned bounding boxes is given in Appendix A.

6.3 Depth Estimation Buffer

The depth estimation buffer is a generalization of the single-plane method discussed in the proceeding section. It represents a uniform subdivision of the screen, with each

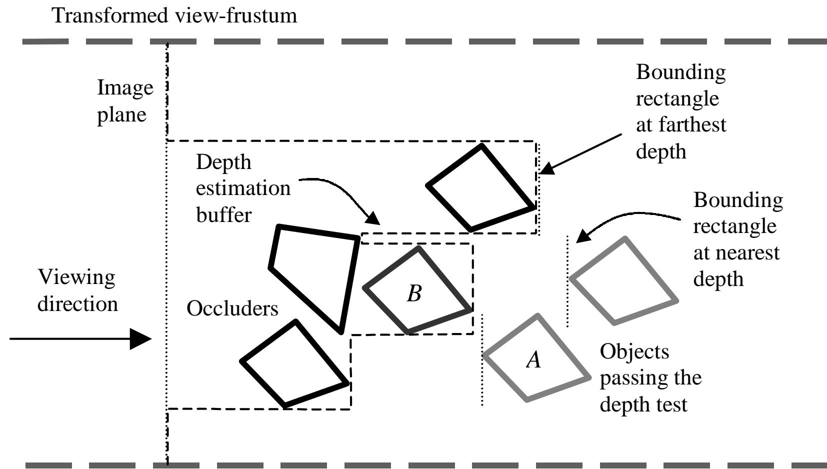


Figure 6.3: The depth estimation buffer.

pixel in the buffer corresponding to a partition. The pixel value stores the farthest depth of the subset of occluders that project into the partition. Put another way, the depth estimation buffer is a 2-D array of small planes, rather than a single plane for the entire set of occluders. The single-plane representation can be conveniently thought of as a single-pixel depth estimation buffer. With screen subdivision, the influence of an occluder's farthest depth is localized to only the partitions onto which it projects. Consequently, the relevant occluders of a tested object are typically only a small fraction of the entire occluder set.

6.3.1 Updating the Depth Estimation Buffer

To compute the depth estimation buffer from a given set of occluders, we have to find their projections into the depth estimation buffer, i.e. to determine which partitions on the screen they project onto. For the same reasons as discussed in section 5.1, we again use a screen-space bounding rectangle to overestimate the screen projection of an occluder. The bounding rectangle is placed at the depth of the *farthest* vertex of the occluder's bounding box, so that the whole object is guaranteed to be on the near side of the rectangle. For brevity, we call such a rectangle a *far* bounding rectangle.

For each frame, the depth estimation buffer is initialized to the depth value representing the *nearest*-possible NDC-space distance to the image plane (typically 0.0). Given an occluder object, we update the buffer as follows. First, we compute its bounding rectangle and set it to the depth of the farthest vertex of the occluder's

bounding box. Then, for each pixel in the depth estimation buffer overlapped or intersected by the rectangle, we compare the pixel depth with the rectangle's depth: if the latter is greater, the former is overwritten. The depth representation of a given set of occluders is built by repeating the above process for all the occluders.

The depth estimation buffer conservatively approximates the farthest depth extents within which an object can possibly be closer to the viewer than an occluder. That is, it represents a *far* boundary; any object behind this boundary is guaranteed to be behind all the surfaces of its relevant occluders. Intuitively, the construction of the buffer from given occluders is for each occluder to push the boundary away from the viewer (i.e. if the occluder depth is greater, it replaces the depths in the buffer). The "pushing" effect of an occluder is conservatively exaggerated both in 2-D (where screen projection is overestimated by the bounding rectangle) and in depth (when the rectangle is place at the farthest depth of the bounding box). The bounding rectangles put occluders into more partitions (buffer pixels) than they actually project to, which results in a greater number of relevant occluders for the tested objects than they actually have.

It is important to note that we are able to use the bounding rectangle and remain conservative exactly because we are estimating the *far* boundary of the occluders. A bounding rectangle at the *nearest* depth of an occluder—a *near* bounding rectangle—is not helpful in constructing the depth estimation buffer since it evidently does not guarantee that an object on its far side is behind the occluder. In section 6.4 we will see that the *near* boundary of the occluders has to be generated by scan conversion of the occluders with depth buffering.

Finally, as a more concrete annotation to Figure 3.2, the approximations in updating the depth estimation buffer (using the bounding rectangle and placing it at the far corner of the bounding box) have only local effect in making the whole occlusion culling process more conservative. That is, the only occluded objects that fail to be culled away because of these approximations are those closer to the eye than the boundary defined by the depth estimation buffer but farther than the optimal, nearest boundary, had we performed actual scan-conversion.

6.3.2 Depth Tests with the Depth Estimation Buffer

The depth estimation buffer defines the far boundary of the occluders. For each object, we would like to decide if it is entirely on the far side of the boundary. This is done by projecting the object to the buffer and comparing its depth to the buffer

pixels. Objects selected as occluders do not have to be tested, as they must be on the near side of the boundary.

Once again, a bounding rectangle is used to overestimate screen projection—this time for the tested object, and the rectangle is set to the depth of the *nearest* corner of the object's bounding box (and is thus called a *near* rectangle). The whole object is entirely behind the rectangle. The depth values of the buffer pixels touched by the rectangle are compared against the depth of the rectangle. If the rectangle's depth is greater than all of these buffer pixels, then it is behind the far boundary of the occluders. This implies that the corresponding object is also behind the far boundary and thus passes the depth test.

6.3.3 Discussions

The depth estimation buffer should have low resolution in order to reduce the cost of update and query operations. Actually, the resolution does not have to be very high, because accuracy is already limited by the extensive use of bounding rectangles. In practice we have found that using a resolution of 64×64 is enough, and higher resolutions do not provide further benefits.

In section 3.1 we pointed out that one benefit of our decomposition of the visibility problem into overlap tests and depth tests is that we can then use a more approximate, lower-cost representation for depth than for 2-D projection. [4] The depth estimation buffer is exactly such a depth representation, which can be significantly lower in resolution than the occlusion maps (which records the occluders' cumulative screen projection). Also, the occlusion maps must be built by actually rasterizing the occluders, whereas the depth estimation buffer can be constructed much more cheaply using bounding rectangles.

A major strength of the depth estimation buffer lies in its portability. It works efficiently in pure software and makes no assumption about the hardware graphics architectures. Since different hardware graphics systems manages depth in vastly different ways, the depth estimation buffer is key to the portability of our occlusion culling algorithm.

[4] Although we extensively use bounding rectangles to approximate screen projection in updates and queries to the depth estimation buffer, this approximation affects only the result of the depth tests. It is thus really a depth approximation instead of an approximation in projection, judging from its role in the overall process of occlusion culling.

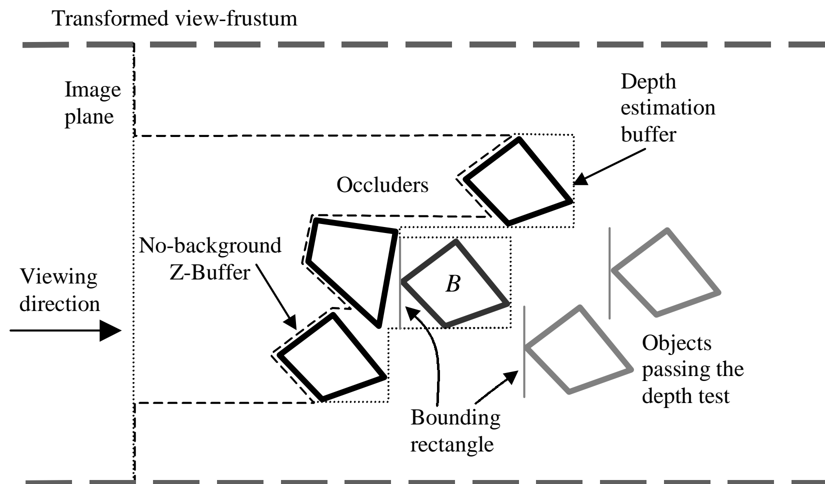


Figure 6.4: The no-background z-buffer.

6.4 No-background Z-Buffer

An obvious drawback of the depth estimation buffer is that no object that has been chosen as an occluder can later pass the depth test; all occluder objects have to be regarded as visible and finally rendered. In other words, we cannot get rid of redundant occluders using the depth estimation buffer. Obviously, this is because the buffer represents a *far* boundary of the occluders, and any occluder involved in building the boundary must be on its near side.

However, if the underlying graphics hardware supports z-buffer-based visibility determination, and the z-buffer is easily accessible at the application level, we may want to make use of the depth values in the z-buffer produced by rendering the occluders.[5] The non-background depth values in the z-buffer stores the *nearest* depths of the occluder surfaces, which form a sampled representation of the optimal boundary (i.e. the visible occluder surfaces) behind which an object cannot possibly occlude any part of its relevant occluders. However, the z-buffer cannot be used directly because of the background depth values, i.e. those initialized to the maximum depth (often 1.0) and never updated during occluder rendering because no occluder projects to them. If we directly use such a z-buffer in our depth tests, an object will fail to pass the depth test if its bounding rectangle touches any of the background pixels.

To transform a conventional z-buffer into what conforms to our definition of depth

[5]Note that in order to use the z-buffer we have to turn on z-buffering when we render the occluders, which is not necessary if we only render for the occlusion maps.

tests, we replace the depth of all the background pixels with the smallest possible depth (usually 0.0). The modified z-buffer is called a *no-background z-buffer*. The depth tests described in section 6.3.2 remain the same except that the depth estimation buffer is replaced by the no-background z-buffer. Using the no-background z-buffer means that pixels onto which no occluder projects are ignored in depth tests. The no-background z-buffer is compared to the depth estimation buffer in Figure 6.4, with the near and far boundary depicted in dashed and dotted lines, respectively.

Clearly, the major strength of the no-background z-buffer is that it represents (to the precision of rasterization) the optimal, nearest boundary for our depth tests, rather than the far boundary represented by the depth estimation buffer. Therefore, the tested objects can make full use of the available occlusion. In Figure 6.4, occluder B will pass the depth test with a no-background z-buffer and thus has a chance of being culled away.

Another advantage of the no-background z-buffer is that it is relatively insensitive to which objects are chosen as occluders. If a redundant occluder, i.e. one hidden by other occluders, is selected, the buffer will not be affected at all.[6] In contrast, the depth estimation buffer suffers from the fact that a redundant occluder can potentially push the far boundary away, especially if it is distant from the viewer. In the extreme case when all objects in the scene are selected as occluders, the no-background z-buffer still captures the near boundary made up of nearest occluder surfaces, but the far boundary defined by the depth estimation buffer will reside beyond all objects so that no occlusion culling is possible.

The no-background z-buffer is also not sensitive to the locality of the bounding boxes of the occluders. Recall that in updating the depth estimation buffer we place the bounding rectangle of the bounding box at the box's farthest corner. When the box has a large aspect ratio or fails to bound the object tightly, the bounding rectangle can become too conservative an estimation. The no-background z-buffer does not have this problem.

As a disadvantage, the no-background z-buffer has to have the same resolution as the the level-0 occlusion map, because they are generated in the same rendering process. This resolution usually has to be significantly higher than the depth estimation buffer, since the occlusion map has to fine enough to avoid too much error in overlap tests. Consequently, the depth tests are more expensive with the no-background

[6] More occluder primitives result in slower rendering when the z-buffer (and the occlusion map) is first built, so we still want to avoid selecting redundant occluders as much as we can.

z-buffer. For example, if we use a 256×256 no-background z-buffer instead of a 64×64 depth estimation buffers, the number of depth comparisons in the depth tests increases by 16 times. When considering this technique, we want to estimate that this significant increase in overhead does not more than cancel out the higher culling rate we gain from using the no-background z-buffer.

Chapter 7

Occluder Selection

As discussed in Chapter 2, occlusion culling must begin with occluder selection. The *optimal* occluder set (any object of which contributes to occlusion) is exactly the visible portion of the model. Thus, finding this optimal set is the visibility determination problem itself, and occluder selection is typically an approximation. More specifically, occluder selection is inherently an approximation, estimating which objects are likely to be visible without actually solving the visibility determination problem itself.

In this chapter, we present static (preprocessing) and dynamic (run-time) occlusion selection methods. Static occlusion selection does not actually estimate which objects are probable occluders for any particular view. Rather, it employs some heuristics to identify objects that are unlikely to be good occluders, so that the dynamic occluder selection algorithm does not consider them as candidate occluders. It also simplifies objects (i.e. approximating them with fewer primitives) while preserving their occlusion. Dynamic occluder selection is performed for every frame, choosing a subset of the objects as occluders based on the current viewing parameters, as well as on visibility information from previous frames.

7.1 Static Occluder Selection

As preprocessing, we employ several simple criteria to identify and flag objects that are unlikely to be good occluders. It should be noted, however, that any object can be a good occluder given a favorable view setup, so there is rarely an object that is a bad occluder for *all* views. Static occluder selection uses heuristics that are based on how the model is likely to be viewed. For example, in a walkthrough of a factory we can reasonably assume that the viewer is unlikely to zoom very close to a bolt in the walkway.

We use the following criteria to evaluate an object's potential to be a good (or bad) occluder:

- **Size:** Very small objects (bolts in the hallway of a factory model, street lamps in a city model) are unlikely to be good occluders.
- **Spatial Locality** When the depth estimation buffer is used, the occluders should have well-localized bounding boxes, i.e. boxes that are small compared to the size of the whole scene. Also, it helps if the boxes have small aspect ratios. Occluders with oversized and ill-shaped bounding boxes can make the depth estimation buffer, and thus occlusion culling, too conservative (Chapter 6).
- **Rendering Complexity:** Objects with a high rendering complexity (e.g. high polygon count) are not preferred, as rendering them to build the occlusion map takes considerable time. However, many of these objects can be simplified, and their simplified versions, which have low polygon count but preserve the occlusion of the original objects, can be good occluders.
- **Redundancy:** An object attached to a larger object, e.g. a clock on the wall, may not occlude much that the larger object does not. It is thus considered redundant when the larger object is already selected as an occluder.

Other environment-specific heuristics can be used. For example, in an architectural environment, the walls are by far the most important occluders. Even if we regard only the walls as candidate occluders, we will still retain most of the occlusion that exists in the environment.

Although obvious to human eyes, object configurations to which these criteria apply are not always easy to detect automatically with an algorithm. Size, spatial locality and rendering complexity are easy to evaluate automatically, but we have yet to find a satisfactory algorithm to automatically identify redundant occluders.

7.2 Occlusion Preserving Simplification (OPS)

Since the selected occluders are rendered at each frame to build the occlusion map pyramid, the overhead of occlusion culling is directly related to the number of primitives comprising the selected occluders. Given a complex object in the original model,

we would like to derive from it a simplified version (with fewer polygons) that retains most of the occlusion the original object can provide. This is what we call occlusion preserving simplification.

Current geometry simplification algorithms [Tur92, SZL92, RB93, CVM+96, Hop96] operate under the constraint that the simplified object should preserve the shape of the original object to certain fidelity. The error bound of the simplification can be measured by various distance metrics, e.g. the maximum distance between the the simplified surfaces and the original.[1]

Given a discrete set of error bounds, *view-independent* algorithms create a hierarchy of discrete *levels-of-detail*, or *LOD*'s, that approximate the original object to the error bounds.

More recently, view-dependent, dynamic simplification algorithms [XESV97, LE97, Hop97] have been proposed to exploit the fact that the silhouette of an object is more important to its appearance than the parts inside the silhouette. These algorithms operate in a similar error-driven manner as view-independent algorithms, except that the error bounds vary across an object: a lower error bound is used for surfaces near the silhouette and larger errors are allowed inside. By comparison, view-independent algorithms do not have the notion of a silhouette and thus must use a single error bound at a time. Another advantage of view-dependent simplification is that objects can be simplified to continuously varying error bounds as required by the views, while with view-independent simplification we are limited to choosing the pre-computed, discrete levels of detail.

Traditional simplification algorithms, whether view-independent or not, preserve the shape (geometric appearance) of the original object to certain error bounds. Shape preservation is a stronger constraint than occlusion preservation, since if the shape is preserved so must the occlusion. So, these algorithms may as well be used for OPS.

For the purpose of occlusion preservation only, however, we may be able to achieve more simplification because we do not care about the appearance of the objects at all. As an example, Figure 7.1 shows a building on the left, which is approximated by the two of its diagonal rectangles on the right. The approximation does not look similar to the building at all, but when the viewer is moving on ground (i.e. the X-Z plane), it does provide similar occlusion to the building. Since the viewer's position

[1] Some algorithms do not use such fidelity measures directly but specify the maximum number of polygons in the simplified object; the polygon count, however, still implicitly indicates the maximum error for the simplification.

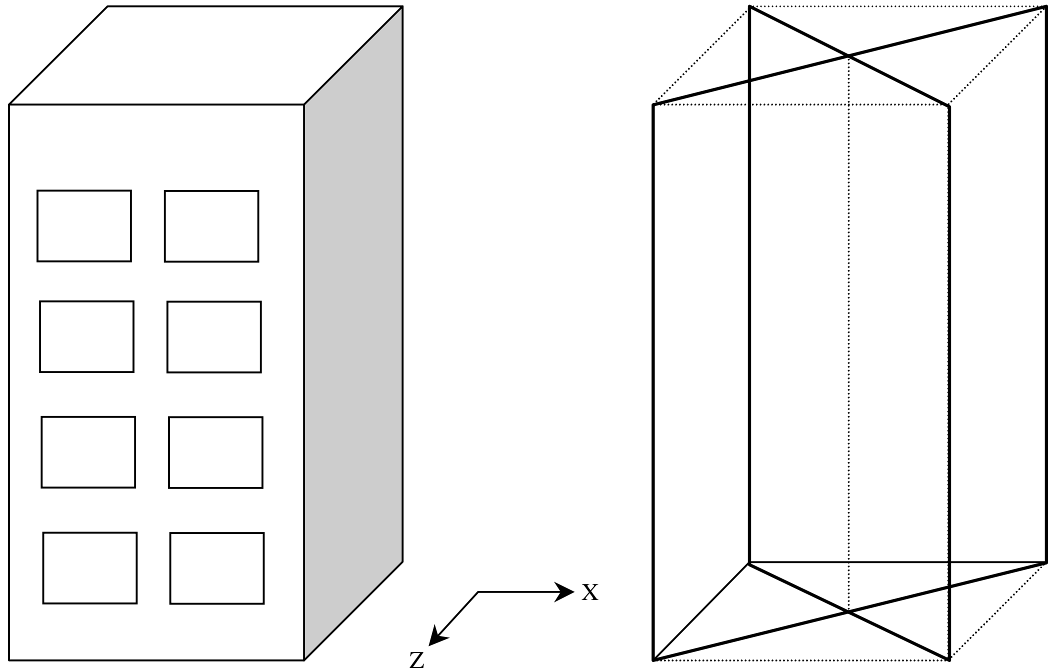


Figure 7.1: View-independent Occlusion-Preserving Simplification

is not limited to a point in order for 7.1(b) to approximate 7.1(a), 7.1(b) is actually an example of *view-independent* occlusion-preserving simplification.

Although the above example is visually obvious to human eyes, such simplification would be difficult to compute automatically. Similar results, however, are easier to achieve in a view-dependent fashion. Specifically, OPS can be conveniently viewed as a variation of view-dependent simplification in which we allow very large errors for the object portions inside the silhouette. For even more drastic simplification, we may choose to keep the silhouette and ignore the interior surfaces and vertices altogether. OPS will then become a triangulation of the simplified silhouette. An example of such simplification is depicted in Figure 7.2. For the spherical occluder and the particular view shown on the left, the result of OPS is a disk (or a squashed sphere).

It is obvious that view-dependent OPS results in simplifications with lower polygon counts than statically-generated view-independent levels-of-detail. However, the overhead involved in computing/tracking the silhouette and performing the simplification or triangulation is significant, and can become prohibitive for a large model. This overhead often more than cancels out the benefit of a lowered polygon count for the occluders. Because of this, in practice we simply use statically-generated, discrete

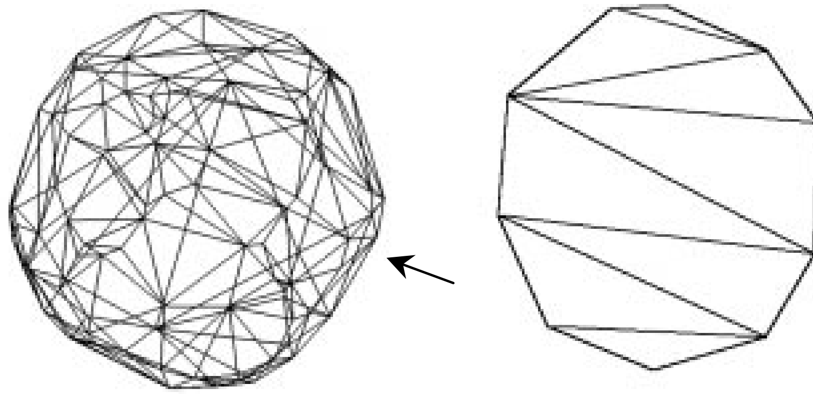


Figure 7.2: View-dependent Occlusion-Preserving Simplification

levels-of-detail for OPS purposes. Static LODs have very little run-time overhead. Furthermore, the LOD's are the same as those used in the final rendering of the scene, so we do not consume any extra memory due to OPS, either.

We use the static LODs generated by the simplification algorithm proposed by Erikson et. al. ([EM98]). The major reason for our choice is that this algorithm can simplify arbitrary models and thus qualifies as part of our general-purpose approach to occlusion culling. Better still, the algorithm has a *area-preserving* feature that simplifies small objects more drastically than objects made of bigger polygons and even removes them altogether. With LODs generated by this simplification algorithm, the choice of level-of-detail for any object is controlled by the *LOD-scale*. The *LOD-scale* indicates the allowable screen-space deviation of the simplified surfaces from the original, in terms of the number of pixels. The larger the *LOD-scale*, the greater the allowable error, and the coarser the level of detail. We have two separate *LOD-scales* for occluder rendering and final rendering, respectively. To reduce the cost of occluder rendering, we use a coarser level-of-detail when rendering an object that has been selected as an occluder into the occlusion map, than when rendering it into the final image (if it is not culled away). Rendering occluders at coarse levels-of-detail may introduce visibility artifacts, i.e. an object not occluded by the original objects may be occluded by the simplified objects, when the simplified objects project to screen areas the originals do not. This is a typical trade-off between accuracy and speed. In fact, we have found that with the simplification algorithm we employ we can set the occluder *LOD-scale* to be four times greater than the normal *LOD-scale*, without noticing any serious visual artifacts.

It is worthy of mentioning that a more restrictive form of OPS is the conservative occlusion preserving simplification (COPS), which requires that the screen-space projection of the simplified object reside *entirely inside* that of the original, from any view point. Intuitively, this means the simplified object should be enclosed entirely by the original object. No existing simplification algorithm can be applied directly to satisfy this constraint. We have modified the simplification-envelope algorithm [CVM+96] to do COPS, by simplifying an inner offset-surface between the original surface and another inner offset-surface with twice as much offset. This algorithm, however, suffers from the same problem as the original simplification-envelope algorithm: it works only on manifold surfaces and thus cannot deal with arbitrary polygonal models. This lack of generality make it impossible for the algorithm to be part of our general-purpose solution to the occlusion culling problem.

7.3 Run-time Occluder Selection

At run-time, each frame begins by selecting a set of objects as occluders. As shown in the framework in section 3.2, the candidate occluders are objects at least partially inside the view-frustum, excluding those marked as bad occluders by our preprocessing. In this section, we describe methods by which the occluders are selected from the candidates. Once an object is selected as an occluder, we render the object at its proper level of detail (according to the occluder *LOD-scale*) to merge it into our occlusion representations.

It is important to bear in mind that in order for occlusion culling to accelerate interactive graphics, its overhead must be well controlled. This is why we have opted for simple, fast methods over complex, theoretically superior ones.

7.3.1 Z Plane

The single plane method for representing depth (section 6.2) is also an occluder selection method. We can put a plane a certain distance from the viewer and choose all objects (totally or partially) on the near side of the plane as occluders. This method is very inexpensive, but works only for scenes where geometry is evenly distributed, unless we vary the distance per frame. The more serious problem, though, is that it does not bound the amount of geometry in the selected occluders, a quantity we must bound strictly to control the occlusion-culling overhead.

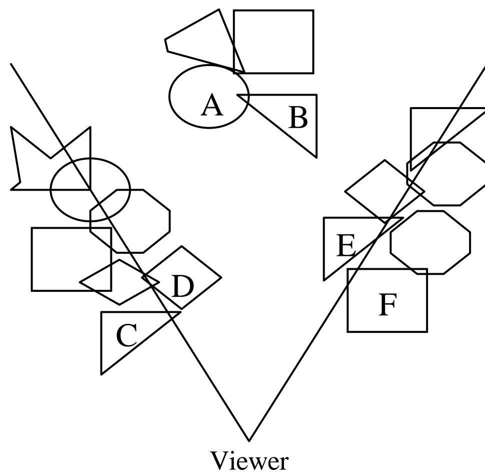


Figure 7.3: Distance-based run-time occluder selection

7.3.2 Distance-Based Selection

Our major occluder selection algorithm is based on a distance criterion and a limit (L) on the number of occluder polygons. Given L, our task is to find a set of good occluders whose total polygon count is less than L.

The objects not culled away by view-frustum culling become occluder candidates, which are considered one by one. The distance between the viewer and the center of an object's bounding volume is used as an estimate of the distance from the viewer to the object. We sort these distances[2] and select the nearest objects as occluders until their combined polygon count exceeds L.

This method works well for most situations, except when a good occluder is relatively far away. One such bad case is shown in Figure 7.3. The distance criterion will select C, D, E, F, etc. as occluders, but L will probably be exceeded *before* A and B are selected. In other words, the distance-based algorithm would choose the nearest objects as occluders, many of which are redundant, ignoring good occluders that are relatively far away. As a result, objects behind A and B are not culled away and must be rendered.

Also, the distance-based method assumes that the distance from the viewer to the center of a bounding box is representative of the distance from the viewer to the polygons in the box. This, in turn, assumes that the bounding box is small compared

[2]Note that full sorting is not necessary. A priority queue suffices since we need only to be able to extract the distances one by one from near to far. We use the word "sort" for simplicity of description.

to the size of the whole scene. When this is not the case, the algorithm may fail to choose near, big polygons as occluders because they could belong to a bounding box whose center is relatively far away.

7.4 Temporal Coherence

Distance-based occlusion selection can be improved by employing *temporal coherence* (also called frame-to-frame coherence) in visibility. Temporal coherence means that non-visible objects from the previous frame tend to remain non-visible for the current frame. It follows that objects culled away (by view-frustum or occlusion culling) for the preceding frame are unlikely to be good occluders for the current frame. So, we may trivially exclude the objects culled away in the previous frame from the set of candidate occluders. The use of temporal coherence reduces the size of the candidate set, which can be particularly beneficial when the number of objects is so large that distance-sorting takes considerable time. Also, by ruling out unlikely candidates we increase the probability of selecting other candidates as occluders, given the constraint on the maximum number of occluder polygons.

The direct application of the above idea means that we select occluders only from the objects that are determined to be visible in the proceeding frame. In situations where visibility changes considerably between frames, however, this approach may seriously reduce the amount of occlusion culling by ignoring important occluders. Two such cases are shown in Figure 7.4. Figure 7.4(a) illustrates a rotation of the view-frustum that causes a large number of objects to enter the frustum. Objects A, B, C, and D, which are outside the view-frustum in frame N, enter the view-frustum in frame $N+1$ due to the rotation. Since these are not visible in frame N and thus not considered occluder candidates in frame $N+1$, it is impossible for A to be chosen as an occluder for frame $N+1$. The result is that object B, C and D must be considered visible due to the loss of A's occlusion. Although it is true that A, which is visible in frame $N+1$, can be selected as an occluder for frame $N+2$ and subsequently cull away the three other objects, there is an undesirable sudden slow-down for frame $N+1$ due to the objects that newly enter the view-frustum. The same problem may also appear if the viewer moves backward. In general, such problems arise because small motion of the view-frustum may result in significant change in the result of view-frustum culling—in other words, the problem is that frame-to-frame coherence is low for view-frustum culling.

To circumvent the above problem, we always treat objects that have just entered in the view-frustum as occluders. To identify these objects, we have to perform view-frustum culling per frame. Objects that are excluded from the set of candidate occluders are those in the view-frustum for both frame N and frame $N+1$, and occluded in frame N. That is, we do not use the temporal coherence for view-frustum culling, but still take advantage of the frame-to-frame coherence in occlusion. More precisely, let the set of objects *inside* the view-frustum be F, and the set of *non-occluded* objects be V ($V \subset F$), then the set of candidate occluders C for frame $N+1$ is:

$$
\begin{aligned}
C_{N+1} &= [F_{N+1} - (F_{N+1} \cap F_N)] \cup (F_{N+1} \cap F_N \cap V_N) \\
&= F_{N+1} - (F_{N+1} \cap F_N) \cup (F_{N+1} \cap V_N) \qquad (7.1)
\end{aligned}
$$

where the subscripts indicate frame number. As an example, in Figure 7.4(a), objects A—D will be occluder candidates for frame $N+1$; E and F will not—because they are occluded in frame N. The overhead of view-frustum culling is relatively low, so doing it per frame is not a problem, especially when the system is pipelined.

The most favorable situation for the application of the formula 7.1 is when the viewer stays in the same place, looking around. In this case, there is perfect temporal coherence in occlusion, i.e. an occluded object in the preceding frame will definitely remain occluded.

Formula 7.1 makes use of temporal coherence in occlusion (i.e. objects occluded in frame N will still be occluded in frame $N+1$), and will thus fail when occlusion changes considerably between two consecutive frames. Such a case is illustrated in Figure 7.4(b). The translation of the view-frustum between frame N and $N+1$ exposes object A, B, C, and D. But since they are occluded in frame N, none of them will be occluder candidates in frame $N+1$. To "correct" our formula for this case, we would have to give up using temporal coherence for occlusion. This means we would not use any temporal coherence at all (since we have formerly given up temporal coherence for view-frustum culling). Usually we do not want to discard temporal coherence altogether because of special cases, so the formula 7.1 is what we employ in practice.

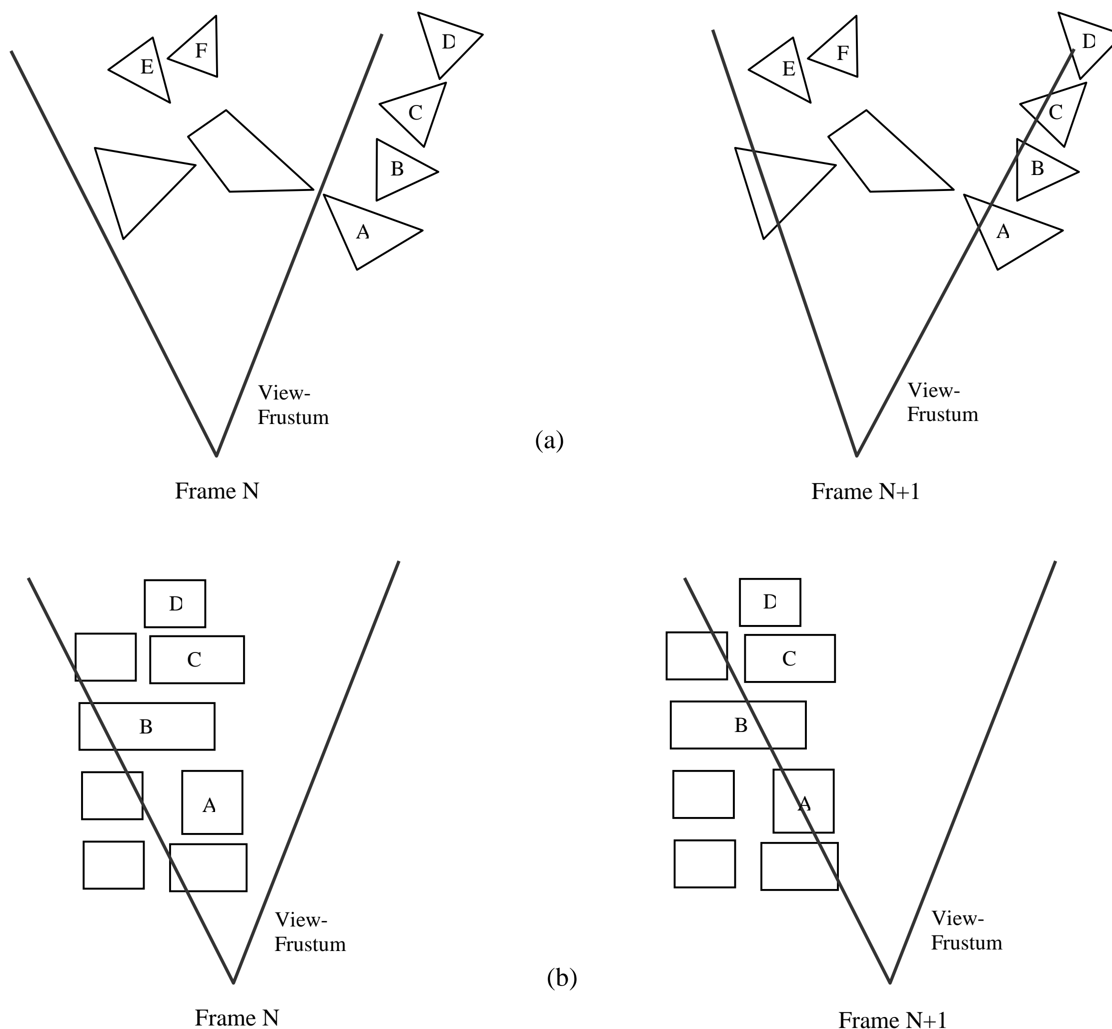


Figure 7.4: Temporal coherence hazards in view-frustum culling and occlusion culling

7.5 Visibility Pre-processing

Dynamic occluder selection can be assisted by visibility preprocessing, which finds visible objects for discrete points in space. The model space can be uniformly subdivided into a 3-D grid. Visibility is sampled at each grid point by surrounding the point with a cube and projecting the objects in the scene onto the cube surfaces, using depth-buffering to resolve visibility. This is similar to the hemi-cube algorithm ([CG85]), except that we use a full cube. Using an item buffer algorithm similar to the hemi-cube algorithm used in radiosity, we can find all the objects visible from any grid point. These objects can be further sorted by their importance as occluders, e.g. their area of projection (indicated by number of pixels they project to on the cube

faces).

For run-time occlusion selection, the eye position is snapped onto the nearest grid point. The list of objects visible from this point (obtained in preprocessing) becomes the set of candidate occluders. Objects from the list are considered in sequential order; if an object is inside the view-frustum, it is used as an occluder. This process is repeated until the maximum occluder polygon count is fulfilled. In so doing, the overhead of occluder selection is only that of performing view-frustum culling on objects visible from the grid point. If the eye is too far away from the closest grid point, the standard distance-based algorithm is used instead.

The grid has to be reasonably dense so that visibility does not change too much between the eye and the grid point onto which it snaps.[3] To reduce the cost of preprocessing, we can identify areas in the model where the viewer cannot reach, and skip samples in those position. This is particularly applicable to models where user motion is naturally limited. For example, in walking through a power plant the viewer will most often traverse pathways and stairs.

In practice, visibility preprocessing is typically applied when a sparse grid is sufficient to significantly improve occluder selection. If dense grids have to be used, visibility preprocessing becomes costly in both time and storage space. approach. In the extreme case, each point in space is sampled for visibility and the run-time visibility determination problem (except for view-frustum culling) is totally eliminated. The problem then is database management, i.e. how to efficiently store and query the huge database of visibility samples. It remains unclear how far this database approach can go.

7.6 Discussion

One-pass occlusion culling (which is used in our two-pass framework) requires more accurate occlusion selection than progressive culling or culling with more passes. This is because occluders have to selected from all of the objects in the view-frustum (when temporal coherence is not used). In progressive culling, due to the prompt updates to the occlusion representations, many objects are culled away and not considered as occluders (i.e., objects go through occlusion tests before they are selected as an occluder). So is the case with multi-pass culling with many passes, but to a lesser

[3] Note that "eye snapping" happens only for occluder-selection purposes, the eye remains in its position for both occluder rendering and the final rendering.

degree. In one-pass algorithms, however, occluder selection cannot use the results of occlusion culling to reduce the number of occluder candidates. On the other hand, as we discussed in Chapter 2, the benefit in occluder selection that progressive or multi-pass algorithms possess) can be canceled out by the overhead caused by multiple updates to the occlusion representations.

Temporal coherence is an interesting remedy to the occluder-selection problem for one-pass occlusion culling. It uses the results of occlusion culling in frame $N-1$ to assist occluder selection in frame N, assuming that that results are still applicable. In occluder selection, progressive and multi-pass algorithms can take advantage of partial results of occlusion culling for the same frame ("immediate feedback"), whereas an one-pass algorithm, when assuming frame-to-frame coherence, utilizes result of occlusion culling from the previous frame ("frame-to-frame feedback").

Occluder selection is also affected by whether updates to the occlusion representations are a by-product of normal rendering or pure overhead. When the updates are pure overhead, the number of occluder polygons must be more strictly controlled. In Chapter 4, we pointed out that occlusion maps are generated by rendering occluders at a much lower resolution than the screen image. The separate rendering makes the updates pure overhead. However, as we also pointed out, obtaining occlusion maps as by-products of normal rendering imposes the constraint that the map resolution be the same as that of the screen image. In practice, the overhead involved in processing such a large image proved to be much more than the cost of rendering the maps separately.

Chapter 8

Implementation and Results

The algorithms described in the preceding chapters have been implemented as part of an interactive viewer for the walkthrough and inspection of 3-D models. The system runs either in a single-thread mode or in a parallel mode, depending on the number of processors available. We have tested the system on several models of different sizes and characteristics. In this chapter we discuss the hardware and software issues in our implementation, and present the performance data we have obtained on our test models. In general, we are able to cull away a significant portion of the occluded geometry and substantially increase frame rates.

8.1 Pipelining

The sequence of operations required to generate a frame are often called a graphics pipeline. Given multiple processors, the pipeline can be partitioned into sections, or processing stages, so that different stages can be executed concurrently for different frames. The partitioning and concurrent execution of the stages are called *pipelining*. A pipeline can be partitioned in many ways, but usually a stage performs a special type of work (e.g. view-frustum culling) on its dedicated processor. Pipelining is an important technique for the acceleration of graphics applications.

Pipelining has been extensively used both at the software application level (e.g. by the Iris Performer [RH94]) and inside the graphics hardware (e.g. [AJ88]). For our two-pass occlusion culling framework (section 3.2), we are interested in software pipelining schemes for the parallelization of the application pipeline, and regard the hardware pipeline as a blackbox to which we send graphics primitives. Also, we assume that there is only one hardware pipe, which is the case for most commercial graphics platforms.

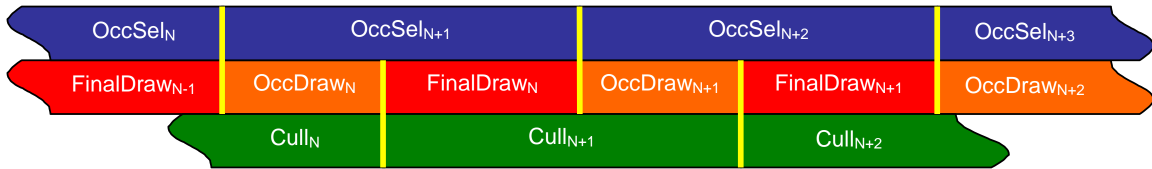


Figure 8.1: Parallelization using three-stage pipelining

Rohlf and Helman [RH94] described an *App-Cull-Draw* pipeline that includes three parallel stages. The *App* stage performs application-specific processing (numerical simulations, modifications to the scene, etc.), the *Cull* stage involves view-frustum culling, and the *Draw* stage issues graphics commands to the graphics hardware to finally render the polygons. There can also be an *ISect* stage if collision detection is activated. Our pipeline differs in several aspects. The two-pass version of our algorithm (see section 3.2) accesses the hardware pipeline twice per frame, once for rendering occluders and building the occlusion map pyramid, and the second time for the final rendering (i.e. drawing the visible geometry). Since there is only one hardware pipeline that needs to be accessed in sequential order, we still have one *Draw* stage. However, the *Draw* stage must now be split into two sequential sub-stages, *OccDraw* and *FinalDraw* to time-share between the two types of rendering. We have also added a new stage, *OccSel*, for occluder selection. The *Cull* stage now includes both view-frustum culling and occlusion culling. Figure 8.1 illustrates the stages in our pipeline and how they execute together at run-time. Subscripts indicate the frame number a stage is processing. The thick, vertical line segments indicate the synchronization point between the stages; that is, the stage on the left must finish before the one on the right begins execution. For example, *OccSel* must finish frame N in order for *OccDraw* to begin rendering occluders for the same frame. Similarly, visibility culling for frame N ($Cull_N$) must be finished before the final rendering for frame N ($FinalDraw_N$) can begin.

The pipeline is controlled by the *Draw* stage, which releases and waits for the other two stages using UNIX semaphores. The pseudo-code in Figure 8.2 outlines the process scheduling, performed by the *Draw* stage, for frame N. Operations skipped when the pipeline “cold-starts” (i.e. starting from frame 0) are indicated by NOP (no operation).

The benefit of pipelining is that the overhead of visibility culling is almost entirely hidden by performing culling concurrently with the final rendering. With pipelining, the only non-overlapped overhead for occlusion culling is the rendering of the oc-

```
For frame N:
    (1) Wait for OccSel N+1 (NOP for N=0)
    (2) Release OccSel N+2
    (3) Do OccDraw N+1 (NOP for N=0)
    (4) Wait for Cull N (NOP for N=0,1)
    (5) Release Cull N+1 (NOP for N=0)
    (6) Do FinalDraw N (NOP for N=0,1)
```

Figure 8.2: Process scheduling in the *Draw* stage

cluders and and the construction of the map hierarchy, which have to be performed in serial order with the final rendering. As we will see later in this chapter, this overhead is typically small compared to the frame time for displaying a large model. When the time consumed by occlusion culling is lower than the time taken by direct rendering of the polygons culled away, we achieve a speed-up. So, by reducing the occlusion-culling overhead, pipelining makes it much easier to achieve increased frame rates.

Inside the 3-stage pipeline, the overlap tests and depth tests can each be parallelized as well. Since these tests can be performed homogeneously and independently for each object, they can be easily distributed onto multiple CPUs, if available. However, this is necessary only when the *Cull* stage takes more time than the *Draw* stage, so that *Cull* becomes the bottleneck in the pipeline. Otherwise, the culling time will be hidden by the drawing time anyway, and speeding up the culling stage will not result in any increase in frame rates.

8.2 Test Environments

In this section we discuss several issues in our implementation, e.g. the construction of the bounding volume hierarchy, the choice of active layers in the occlusion map pyramid, etc. These discussions apply to all of our test cases. System parameters that vary with different models (e.g. the number of occluder polygons) are given later in the Results section.

8.2.1 Scene graphs and the spatial hierarchy

Our system is based on a screen graph structure that represents a bounding volume hierarchy. More often than not, input models are not already organized as bounding volume hierarchies, and we must subdivide the model to build such hierarchies and scene graphs. The input models may be a collection of ungrouped, individual polygons ("polygon soup"). The models may have already been divided into objects, but the division may be not spatial, but functional (e.g. the "pipe" object in a submarine model may contain all the pipes that exist all over the model). Such objects have to be broken up to form spatially localized objects.

In the subdivision of an input model, we treat the model as a point cloud comprised of the centers of the input polygons. The box containing the point cloud is recursively subdivided into $M \times N \times P$ sub-boxes until the terminating criteria are fulfilled. This results in a hierarchical spatial subdivision of the points. M, N, and P can be automatically computed according to the aspect ratios of the scene (to produce cube-shaped sub-boxes), or they can be specified by the user. The terminating criteria include the maximum number of points in the leaf node, and the maximum depth of the leaf node in the hierarchy. After the subdivision of points, we compute for each box a bounding volume that bounds the polygons whose centers are in the box. Thus, the subdivision of the points yields a bounding volume hierarchy. An assumption in building bounding volume hierarchies in this way is that the polygons are relatively small compared to the size of the scene, so their centers represent their positions reasonably well.[1]

8.2.2 Resolution for occluder rendering.

For a final display size of 1280×1024, we render the occluders at a resolution of 256×256. In other words, the original, highest-resolution occlusion map is 256×256. This resolution proved to be sufficient in practice. The coarsest map in the pyramid is usually 4×4.

[1] We could subdivide large polygons, but this is usually avoided in order to avoid increasing the already-overwhelming number of polygons in a large model.

8.2.3 Construction of the occlusion map pyramid.

We have found that on SGI InfiniteReality graphics systems (more detailed specifications will be given later), the hardware/software break-even point for the construction of the occlusion map pyramid (section 4.4) is at the level of the 128×128 map. That is, if we filter the 256×256 map to 128×128 using hardware texture mapping, and do the rest of the filtering in software, we can generate the map pyramid in the shortest time. The construction, from 256×256 to the 4×4 minimal resolution, takes approximately 4 milliseconds.

8.2.4 Active levels in the map pyramid.

We do not use all of the levels in the pyramid for overlap tests, because the benefit of further recursion diminishes as the overlap test descends into finer maps. That is, if the overlap test still cannot decide on complete overlap at a high-resolution map, descending further into finer maps is unlikely to help. So, we stop the overlap test at the level of the 64×64 occlusion map, treating it as if it were level-0—the overlap test returns false if the bounding rectangle still covers or intersects low-opacity pixels at the 64×64 level. The choice of this particular level is based on our experience with various test cases, which shows that very few overlap tests succeed at levels of higher resolution than 64×64. The five levels (from 4×4 to 64×64) are called the *active levels* in the pyramid, and the 64×64 map is the level-0 active map. By using fewer levels, we reduce the overhead of overlap tests. Further, much less memory is needed to store only the active layers than to store the whole pyramid, which improves cache performance (our test machines have 32K L1 cache). The active layers take only 5456 bytes of memory, whereas the 256×256 map itself takes 64K.

8.3 Performance Measures

Our goal in testing our system is to demonstrate the effectiveness of our occlusion culling algorithm in real-world graphics systems. Like our model viewer in which occlusion culling is embedded, these systems make use of primitive-reduction techniques such as view-frustum culling and level-of-detail simplification. Occlusion culling further reduces the number of primitives *in addition to* these techniques. The benefit of occlusion culling should be measured with all the other techniques present.

The use of levels-of-detail of the objects implies that the geometric complexity of

the scene changes as the viewer moves. To measure the true benefit of the visibility culling, the number of polygons culled away should be computed based on the current levels-of-detail of the objects, not the polygon counts of the original objects (i.e. the highest level-of-detail). In other words, the true impact of visibility culling should be measured *after* the level-of-detail techniques have been applied.

More specifically, the number of polygons culled-away by view-frustum culling and occlusion culling are calculated based on the polygon counts of the culled-away objects in their current levels of detail (selected based on the LOD-scale). For example, suppose an object, A, has 25,000 polygons in its highest level-of-detail. Suppose also that at a frame, the LOD-scale indicates that it suffices to use a coarse version of A, which contains only 2,000 polygons. Then, if A is culled away, the reduction of primitives due to culling is only 2,000 polygons, instead of 25,000.

Also, the amount of geometry culled away by occlusion culling is calculated *after* view-frustum culling. If an object outside the view-frustum is occluded by other objects, its absence from the final rendering is not due to occlusion culling. As is shown in section 3.2, occlusion culling is performed only for objects inside the view-frustum.

8.3.1 Graphs

For each of the models used in our experiments, statistics are taken on a pre-recorded path (i.e. the viewer's position and viewing direction for each frame), both with and without occlusion culling. We display our statistics on two graphs: the frame-rate graph and the culling graph.

The frame-rate graph: Frame rates are the ultimate measure for the effectiveness of our occlusion culling algorithm. The frame-rate graph, such as Figure 8.5, shows increased frame rates due to occlusion culling, compared to frame rates obtaining by using the other techniques (level-of-detail and view-frustum culling) without occlusion culling.

The culling graph: The culling graph, such as Figure 8.6, shows the number of remaining polygons as the primitive reduction techniques are successively applied. For each frame, level-of-detail simplification performs the first round of primitive reduction. The total number of polygons in the scene is the sum of the polygon counts of all the objects at their proper levels-of-detail.[2] Then, view-frustum culling

[2]In system implementation, it appears as if LOD is preceded by the other techniques, since

is applied, followed by occlusion culling on the objects inside the frustum. The first three curves in a culling graph correspond to the number of remaining polygons after these successive steps of primitive reduction.

A fourth curve in the culling graph shows the number of visible polygons if we perform *exact*, not conservative, occlusion culling. This provides a reference point for judging how closely conservative occlusion culling approaches exact culling. This number is obtained by encoding object identifiers as colors and rendering the objects into the frame buffer with z-buffering. After rendering, a visible object has its identifier in at least one pixel. Note that all the polygons in an object are counted as visible when any of them is visible. Alternatively, we could use polygon identifiers, instead of object identifiers, so that we compute the number of visible polygons after exact, *per-polygon* occlusion culling. However, since we perform per-object culling, it is only fair that we compare to per-object exact culling.

8.4 Experimental Results

We now show experimental results on three models. The first model is a city, serving as an example of a small-scale environment. The second is a medium-sized CAD model of a portion of a submarine. The third is a large-scale model of a power plant.

Performance data are obtained on two Silicon Graphics workstations. The first two models are displayed on an Onyx II with InfiniteReality graphics and four 195MHz R10000 processors. The second machine, on which tests on the power plant model were conducted, is an Onyx I also with the InfiniteReality graphics subsystem. It has four 250MHz R4400 processors.

8.4.1 The City Model

The city model is shown in Figure 8.3. It has 312,524 polygons and comprises 12 copies of a model of London, which was originally constructed by Vasilis Bourdakos at the Centre for Advanced Studies in Architecture of Bath University. City models are favorable for occlusion culling in that they usually have high depth complexity for

the proper LOD is often determined immediately before an object is rendered. However, LOD is conceptually applied before any other culling techniques, because the objects the other techniques process must be at their current level of detail. As we pointed out in Section 8.3, the number of polygons culled away should be computed based on the polygon counts of the culled-away objects *after* LOD is applied.

viewers walking in the streets. For this model we have applied the most basic form of our occlusion culling algorithm, using a single CPU and employing the depth estimation buffer for depth tests. The distance-based algorithm is employed for occluder selection, without taking advantage of temporal coherence; the maximum occluder polygon count is 5,000. Objects in the model do not have variable levels-of-detail, since the buildings are often simple boxes that are already minimally tessellated. The model is loaded by our system as a set of individual polygons and subdivided to build a bounding volume hierarchy. The bounding boxes of the objects (i.e. leaf nodes in our scene graph) are well-localized, because the polygons are small compared to the size of the whole city. This facilitates both occluder selection and depth estimation. The visible geometry is rendered with lighting on (one light). We use display lists to accelerate rendering but do not use triangle strips.[3] Figure 8.4 is a frame on the path along which performance data is recorded; the occlusion map pyramid is shown on the right. The opacity threshold is set to 1.0, meaning that aggressive approximate culling is not used.

In summary, the system parameters for displaying the city model are:

- Single CPU
- Depth estimation buffer
- Distance-based occluder selection without temporal coherence
- 5,000 occluder polygons
- Opacity threshold 1.0
- Lighting; display lists; no triangle strips

The frame-rate graph and the culling graph for this model are shown in Figure 8.5 and 8.6, respectively. Whereas the number of polygons in the view-frustum varies greatly, the number of visible polygons remains relatively constant. As the frame-rate graph shows, we gain up to six times speed-up due to occlusion culling. Our algorithms perform well in their simplest forms for environments with high depth complexity and well-localized bounding boxes. When the viewer is close to the border of the city looking outside, there is not much geometry in the view-frustum (i.e. not

[3] Our current triangle-strip generator produces incorrect normals, so the resulting strips cannot be used when lighting is on.

much geometry to be culled by occlusion) at all, as indicated by the low points on the "After VFC" line in the culling graph. In this case, occlusion culling produces a slower frame-rate than direct rendering due to its constant overhead (e.g. constructing the map pyramid).

8.4.2 The Submarine Auxiliary Machine Room (AMR)

The notional CAD model of the auxiliary machine room in a submarine was shown in Figure 1.1, color-coded to illustrate occlusion culling. The model was provide by Electric Boat. Figure 8.7 is a ordinary view of the model, and Figure 8.8 shows a frame on our test path, with the active levels in the occlusion map pyramid displayed on the right. The original model contains 632,252 triangles. Triangles that belong to the same part (e.g. a rib, the engine case, etc) are grouped into an object. Four levels of detail are generated for each original object using Erikson's simplification system [EM98]. Each simplified level has half of the polygon count of its neighboring more-detailed level. The original objects and their simplified versions are split as necessary in building the bounding volume hierarchy. For final rendering, we use a screen-space error bound (the LOD-scale) of one-pixel deviation, whereas for occluder rendering we set it to 5 pixels. The bounding boxes in this model are not as well localized as in the city model, due to the existence of big polygons comparable in size to the whole model. Thus, a no-background Z-buffer is used for depth tests. For occluder selection, we use the distance-based method with temporal coherence.[4] The maximum number of occluder polygons is set to 25,000. The three-stage pipeline is used to display this model, using three out of the four available CPUs.

Aggressive approximate culling is applied to the AMR model. A pixel in the level-0 active map (64×64) corresponds to a 20×16 block of pixels in the final image (1280×1024). Defining the $\mathcal{L}-0-$hole constraint, we choose $\mathcal{L} = 12 \times 8 = 96$. Also, we set the distribution factor $\mathcal{D} = 2$. The opacity threshold for the level-0 active map is thus $T_{\Omega_0} = 1 - \mathcal{L}\mathcal{D}/(4 * 20 * 16) = 0.85$. The thresholds for the coarser levels are 0.925 (for the 32×32 map), 0.96 (16×16), 0.98 (8×8), and finally 0.99 for the 4×4 map.

The system parameters used in displaying the AMR model are summarized as follows:

[4] Actually, we do not benefit much from using temporal coherence here. Temporal coherence reduces the overhead of occluder selection. But when occluder selection is done in a separate pipeline stage, its overhead tends to be hidden by the rendering time anyway.

- 3 CPU (the three-stage pipeline)
- No-background Z-buffer
- Distance-based occluder selection
- 25,000 occluder polygons
- Opacity thresholds from 0.85 (for the 64×64 map) to 0.99 (for the 4×4 map).
- LOD-scale = 1 pixel
- Lighting; display lists; no triangle strips

The frame-rate graph and the culling graph are shown in Figure 8.9 and 8.10, respectively. In general, we get approximately two-fold speed-up in frame rates.

Figure 8.11 illustrates the benefit of aggressive approximate culling as a result of lowering opacity thresholds. In the figure, only the opacity threshold (OT) for the finest active level (64×64) is marked in the legend. We have seen objects popping in and out through holes among the foreground objects, but they were only noticeable when we look for the artifacts on purpose.

8.4.3 The Power Plant Model

The power plant model was provided by ABB Engineering. It comprises 13 million triangles, half of which are from the various piping structures. Figure 8.12 shows the model from the outside; Figure 8.13 shows a frame (together with the corresponding occlusion maps) from our walkthrough of the interior along the path we used to gather the performance data. The model is originally subdivided into large parts according to their function. Each part has five levels of detail, with the number of polygons decreased by a half in each next-coarser level. The parts (and all the levels of detail) are further subdivided when first loaded into our system, resulting in objects each with no more than 8,000 polygons. The model takes approximately 1.5 gigabytes when loaded into the memory with all the levels of details. For this reason, tests had to be performed on our second machine, which is slower but has 2GB of memory. Considering the slower graphics performance of the machine, the triangles are rendered without real-time lighting (but with pre-computed lighting), in triangle strips. Also, there is not enough memory for creating display lists, so they are not used.

Due to the huge number of polygons involved, the frame rates are sensitive to the quality of occluder selection. Missing a good occluder could mean having to render hundreds of thousands more polygons, resulting in a major negative impact on the frame rates. Although distance-based occlusion selection (assisted by temporal coherence) still works well at most view points, the occasional omission of important occluders causes severe fluctuation in frame rates. For this reason, we employed visibility preprocessing to facilitate run-time occluder selection. A coarse, 20x20x1 grid was put above the ninth floor of the power plant (where our test path is) and visibility is sampled at the grid points. Figure 8.14 shows the walkways, the grid (in white) and the path (in red). On the right, the box in the coarse rendering of the power plant shows the relative position of the floor we are on. The preprocessing took approximately 15 minutes.

The LOD-scale (screen error bound) is set to 1 pixel for final rendering and 5 pixels for occluder rendering. The maximum occluder polygon count is set to 20,000. The system runs in parallel mode on three CPUs and uses the no-background Z-buffer for depth tests.

In summary, the major system parameters are listed as follows:

- 3 CPUs (the three-stage pipeline)
- No-background Z-buffer
- Coarse visibility preprocessing
- 18,000 occluder polygons
- Opacity thresholds from 0.6 (for the 64×64 map) to 0.99 (for the 4×4 map).
- LOD-scale = 1 pixel
- No lighting, no display lists; use triangle strips

Figures 8.15 and 8.16 show the frame rates and the amount of culling along the path shown in Figure 8.14. View-frustum culling discards most of the polygons in the model, which is common anywhere inside the power plant. We gain two to six times speed-up in frame rates due to occlusion culling. The sudden decrease in frame rate (and equivalently, the increase in the number of polygons rendered) in the middle of the path is due to the sudden exposure of complex piping structures, which require a higher occluder polygon count.

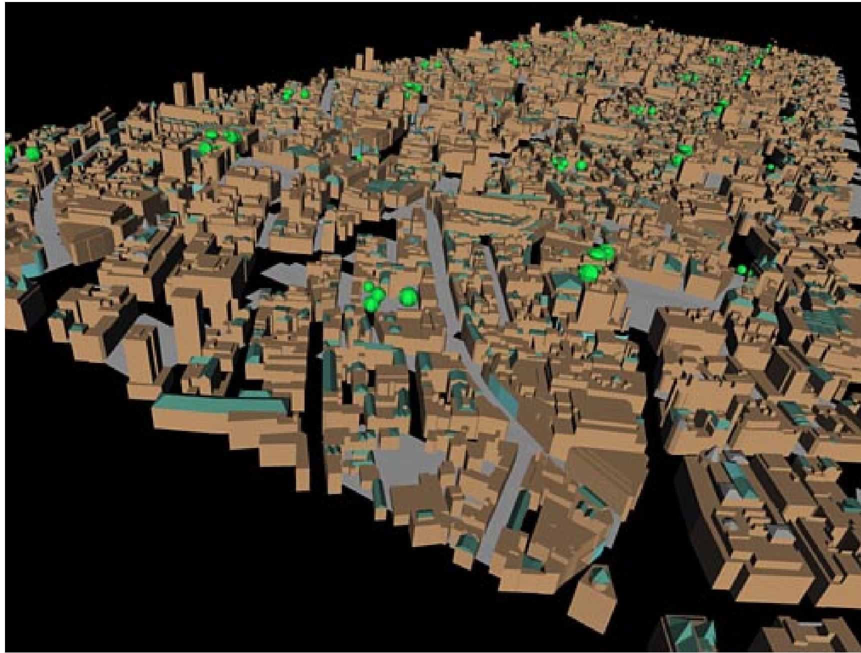


Figure 8.3: The city model

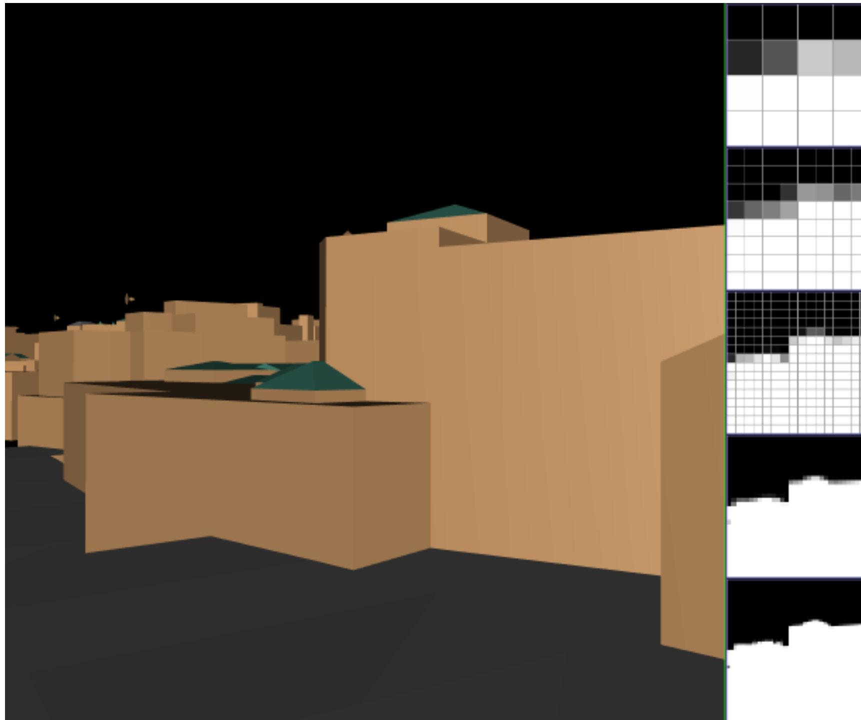


Figure 8.4: A frame from the walkthrough of the city

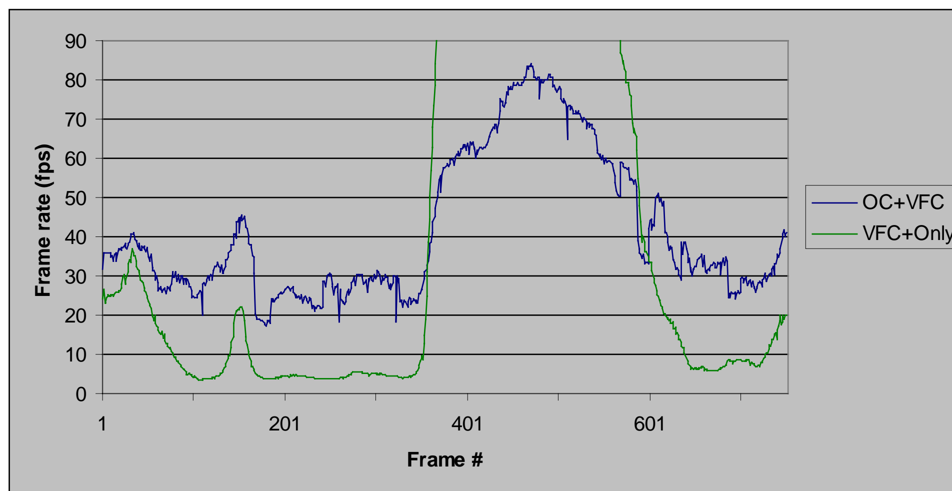


Figure 8.5: Frame rate for the city model

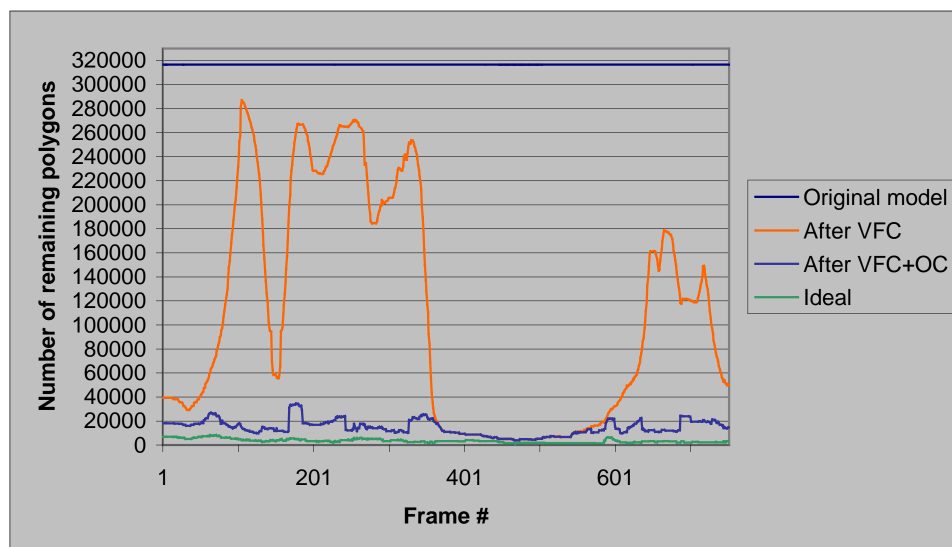


Figure 8.6: Culling in the city model

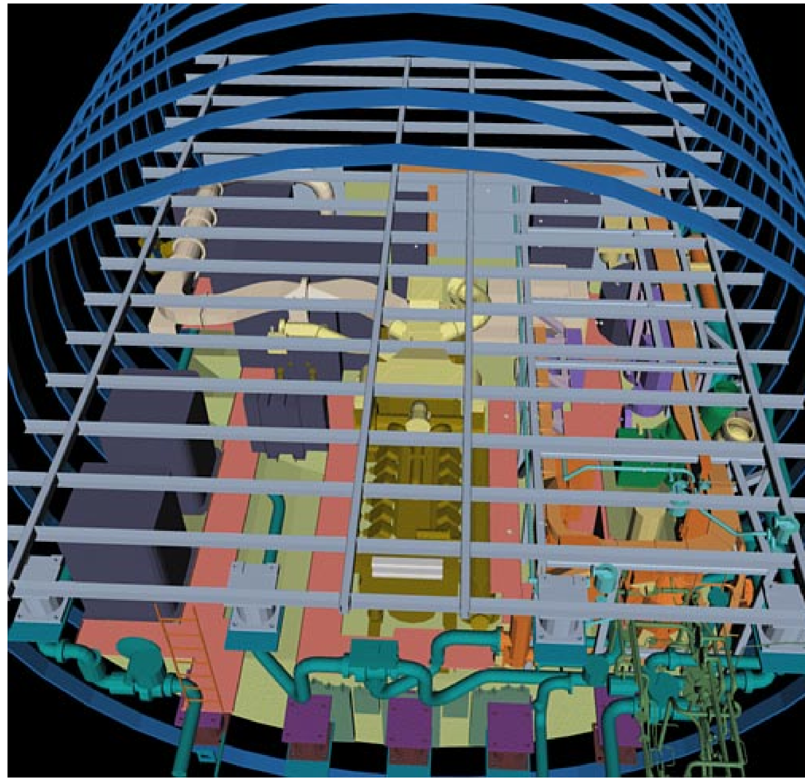


Figure 8.7: The auxiliary machine room in a submarine

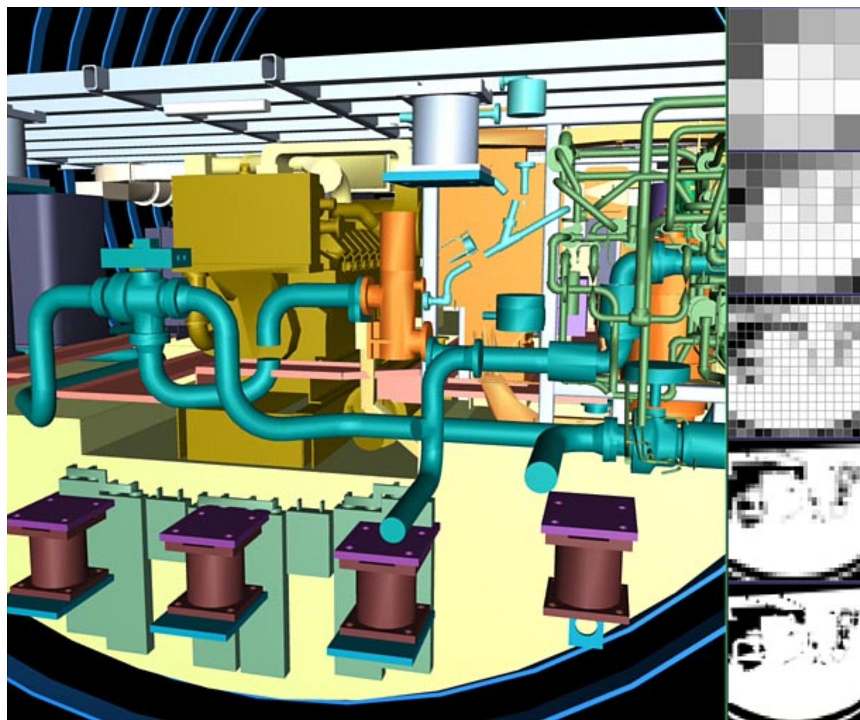


Figure 8.8: A frame on the test path for the AMR model

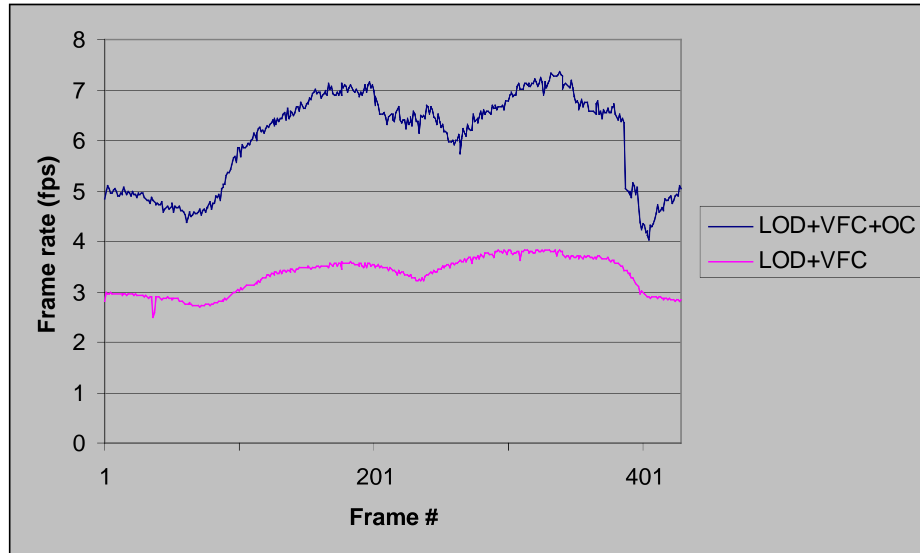


Figure 8.9: Frame rates for the AMR model

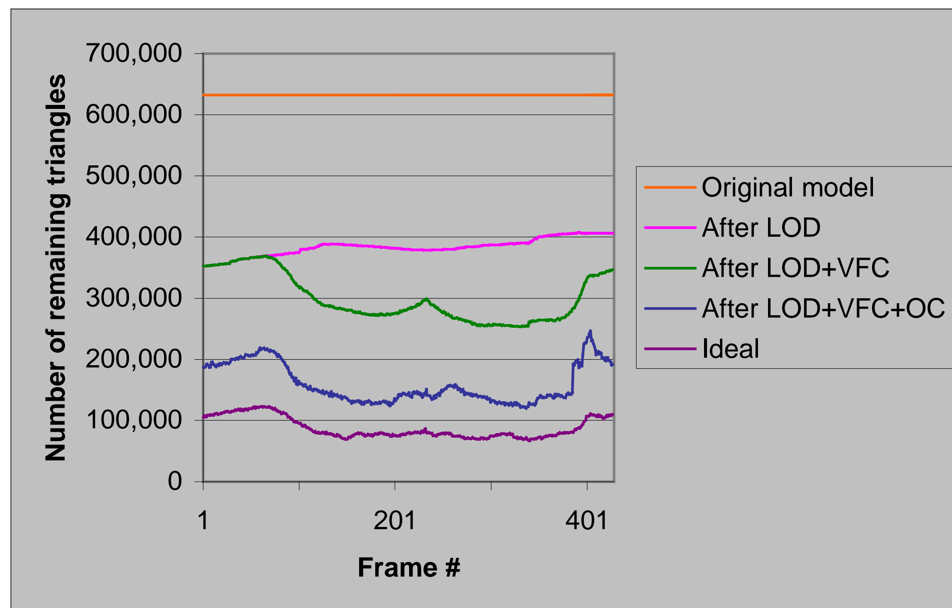


Figure 8.10: Culling in the AMR model

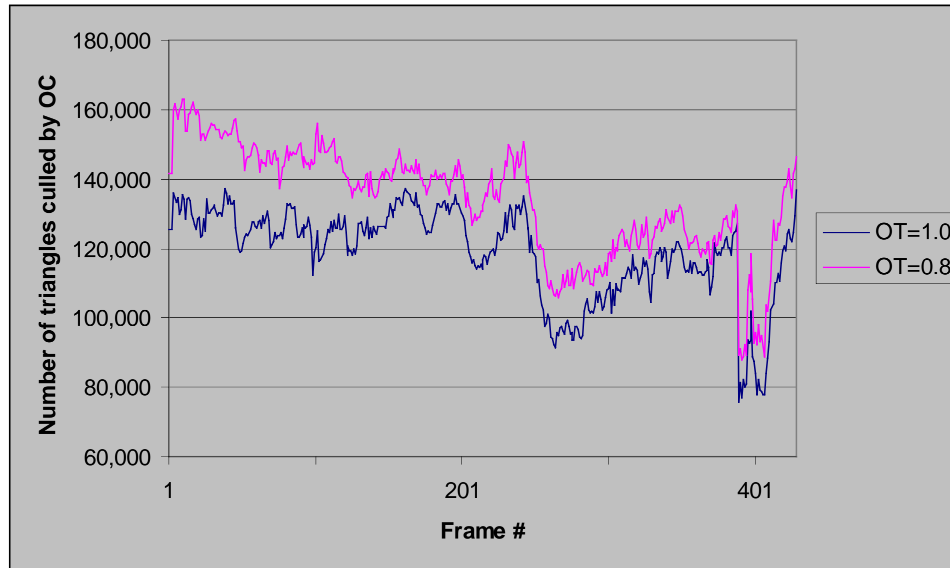


Figure 8.11: Aggressive approximate culling on the AMR model

Figure 8.12: The power plant model

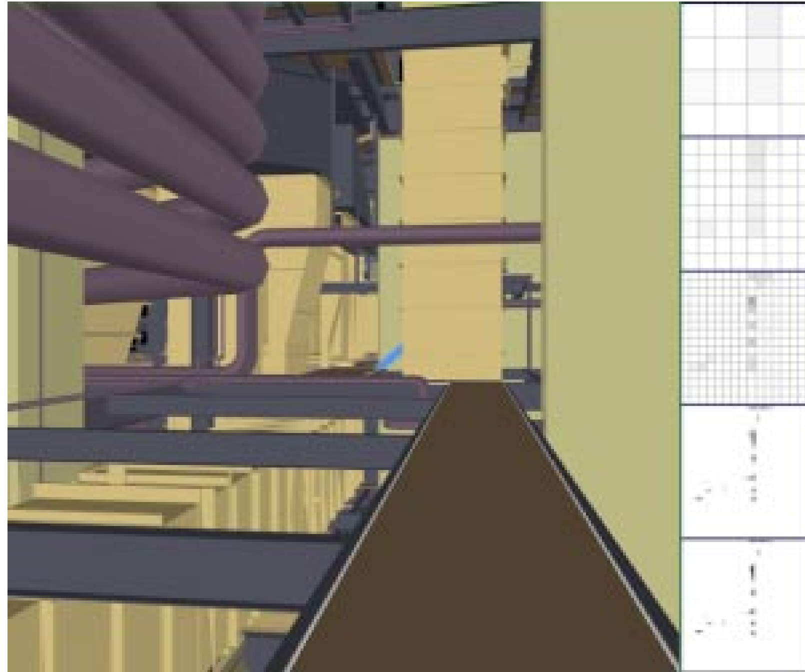


Figure 8.13: A frame on the test path for the power plant model

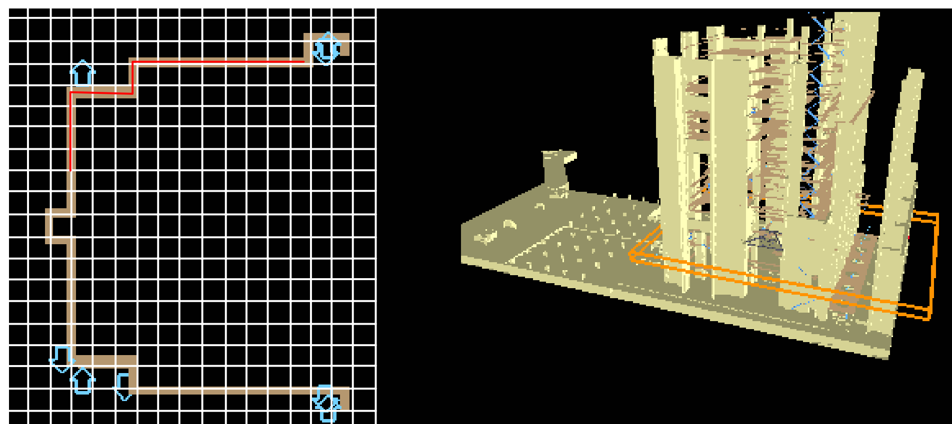


Figure 8.14: The walkway of the 9th floor in the power plant model

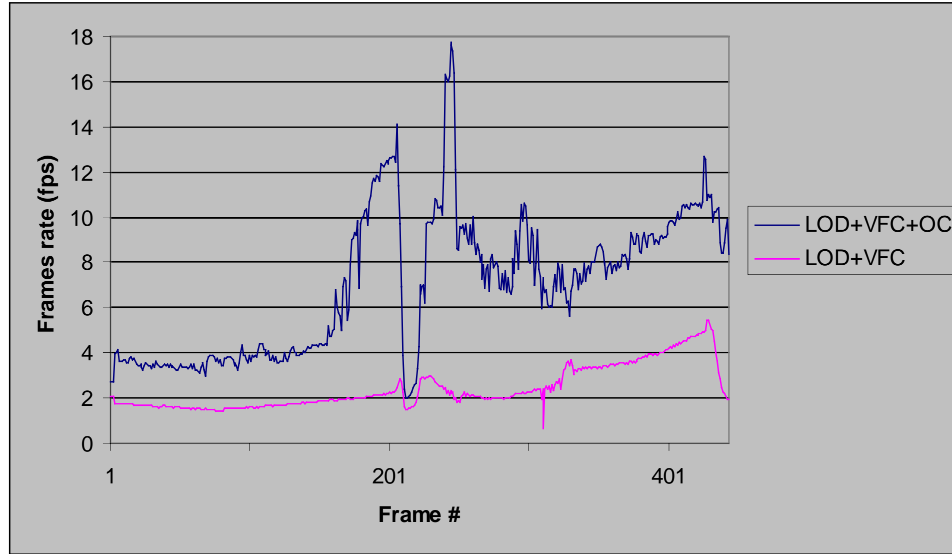


Figure 8.15: Frame rates for the power plant model

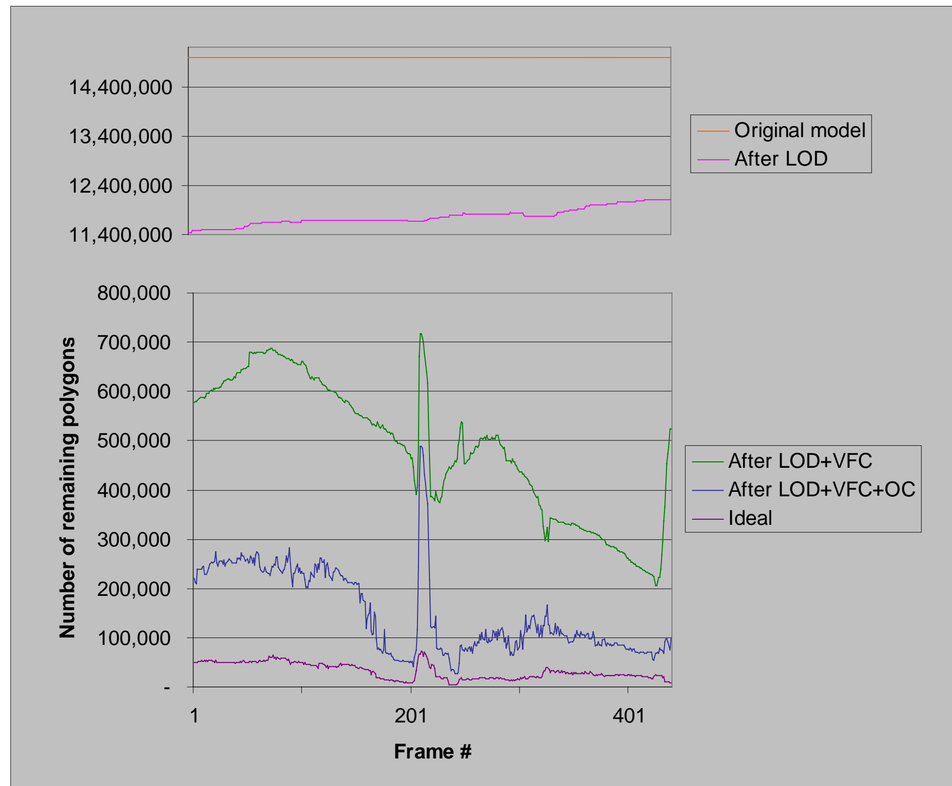


Figure 8.16: Culling in the power plant model

Chapter 9

Conclusion and Future Work

In this dissertation we have presented a new approach to occlusion culling. The goal of our work is to provide *effective* occlusion culling algorithms that can accelerate the interactive display of real-world large scale models. We have observed that such algorithms must simultaneously satisfy three criteria: it must be general with respect to the types of scenes handled; it must increase the frame rates; and it must be easy to implement. We have presented algorithms that have been proven to meet these goals.

Our approach is based on the observation that the representation of cumulative occlusion largely decides the capabilities of an occlusion-culling algorithm. We begin by decomposing the occlusion-culling problem into two sub-problems—that in order for an object (the potential occludee) to be occluded by the occluders, its screen-space projection must be inside the cumulative projection of the occluders, *and* it must not occlude any part of the occluders. These two necessary conditions for occlusion are verified by the overlap tests and the depth tests, respectively. The cumulative projection and the depth of the occluders are represented separately to support these tests.

The problem decomposition allows us to be more approximate in estimating depth than screen projection, since the amount of occlusion we can obtain is more sensitive to the latter. It also makes our algorithms more portable across different graphics platforms with varying hardware strategies in resolving depth. The most notable benefit of the decomposition, however, is that the screen projection can be represented by a 2-D image (i.e. an occlusion map) and 2-D image analysis techniques can be utilized to analyze occlusion.

We have employed hierarchical occlusion maps—an image pyramid derived from the cumulative screen projection of the occluders—for the analysis of occlusion. With

the pyramid, occlusion is represented at multiple resolutions, and overlap tests are performed hierarchically through the pyramid. Due to this multi-resolution representation, we have developed such concepts as *levels of visibility*, and presented unique features such as *aggressive approximate culling* (i.e. culling away barely-visible objects).

For depth representation, we have presented the *depth estimation buffer* and the no-background Z-buffer. The former conservatively estimates the *far* boundary of the occluders, while the latter is derived from a conventional Z-buffer and captures the *near* boundary.

We have described the framework for a two-pass implementation of our algorithm, which is tailored to take advantage of currently-available graphics workstations (e.g. the Silicon Graphics workstations). The framework has been parallelized using software pipelining. The details in the choices of system parameters have also been discussed. Our performance tests on three different models, which are small, medium and huge in size, respectively, have shown encouraging results.

Our future work will focus on the implementation of our core algorithm on different hardware architectures. Depending on the underlying platform, an implementation may look rather different from the two-pass framework presented here. Nonetheless, the gist of our approach, i.e. our occlusion representations and our algorithms for the overlap tests and depth tests, will still play a fundamental role. For example, the new Intel Pentium processors supports MMX instructions that facilitate software transformations and scan-conversion of polygons, making it possible to render occluders and build occlusion maps on the main CPU. Many operations in our culling algorithm are easy to accelerate in hardware and, once supported by hardware, will greatly increase the efficiency of occlusion culling. More intimate integration of occlusion culling into graphics hardware will make it feasible to perform multi-pass or even progressive culling, which can be much less conservative than the two-pass variation.

The prospect of the wide-spread use of occlusion culling in graphics applications is indeed very exciting. We expect occlusion culling to be widely supported by graphics systems in the near future.

Appendix A

Incremental Transformation of Axis-Aligned Bounding Boxes

We define an axis-aligned bounding box (AABB) by a base point, (x_0, y_0, z_0), and the increments (dx, dy, dz), so that the eight corners of the bounding box are:

$$(x_0, y_0, z_0)$$
$$(x_0, y_0 + dy, z_0)$$
$$(x_0, y_0 + dy, z_0 + dz)$$
$$(x_0, y_0, z_0 + dz)$$
$$(x_0, y_0, z_0)$$
$$(x_0 + dx, y_0, z_0)$$
$$(x_0 + dx, y_0 + dy, z_0)$$
$$(x_0 + dx, y_0 + dy, z_0 + dz)$$

$(x_0 + dx, y_0, z_0 + dz)$

The transformations of the corners by a 4×4 matrix share many common subexpressions, which can be computed once and stored for later use.

Let the elements of the transformation matrix, M, be m_{ij}, $0 \leq i, j < 3$:

$$M = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix}$$

Then

$$M \begin{bmatrix} x_0 + dx \\ y_0 + dy \\ z_0 + dz \\ 1 \end{bmatrix} = M \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} + M \begin{bmatrix} dx \\ 0 \\ 0 \\ 0 \end{bmatrix} + M \begin{bmatrix} 0 \\ dy \\ 0 \\ 0 \end{bmatrix} + M \begin{bmatrix} 0 \\ 0 \\ dz \\ 0 \end{bmatrix}$$

By computing the full transformation of the base point, and one increment vector in each dimension ($M(dx, 0, 0, 0)^T$, etc.), the transformation of the remaining 7 corners is no more than adding proper increment vectors, and dividing by the w component. The increment vectors are:

$$M(dx, 0, 0, 0)^T = (m_{00}dx, m_{10}dx, m_{20}dx, m_{30}dx)^T$$

$$M(0, dy, 0, 0)^T = (m_{01}dy, m_{11}dy, m_{21}dy, m_{31}dy)^T$$

$$M(0, 0, dz, 0)^T = (m_{02}dy, m_{12}dy, m_{22}dy, m_{32}dy)^T$$

In computer graphics, the last column of M is often not full, a fact that further reduces (slightly) the amount of computation in the "full" transformation of the base point.

Bibliography

[AJ88] Kurt Akeley and Tom Jermoluk. High-performance polygon rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 239–246, August 1988.

[App67] Arthur Appel. The notion of quantitative invisibility and the machine rendering of solids. *Proc. ACM Natl. Mtg.*, page 387, 1967.

[App68] A. Appel. Some techniques for shading machine renderings of solids. In *IFIP*, volume 32, pages 37–45, 1968.

[ARB90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 41–50, March 1990.

[BE89] M. Bunker and R. Economy. *Evolution of GE CIG Systems.* General Electric Company, Daytona Beach, FL, 1989.

[BK70] W. J. Bouknight and K. C. Kelly. An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources. In *Proc. AFIPS JSCC*, volume 36, pages 1–10, 1970.

[BM96] D. Blythe and T. McReynolds. Programming with OpenGL: Advanced rendering. *SIGGRAPH'96 Course Notes*, pages 27–28, August 1996.

[Bou70] W. Jack Bouknight. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Communications of the ACM*, September 1970.

[Bur88] Peter J. Burt. Smart sensing within a pyramid vision machine. In *Proceedings of IEEE*, volume 76, pages 1006–1015. IEEE, August 1988.

[Car84] Loren Carpenter. The A-buffer, an antialiased hidden surface method. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 103–108, July 1984.

[Cat74] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974.

[CH92] I-Cheng Chang and Chung-Lin Huang. Aspect graph generation for non-convex polyhedra from perspective projection view. *Pattern Recognition*, 25(10):1075–1096, 1992.

[Cla76] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[CT97] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1997.

[CVM+96] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick P. Brooks, Jr., and William Wright. Simplification envelopes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 119–128. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[DDP96] Frédo Durand, George Drettakis, and Claude Puech. The 3D visibility complex: A new approach to the problems of accurate visibility. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 245–256, New York City, NY, June 1996. Eurographics, Springer Wein. ISBN 3-211-82883-4.

[DDP97] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 89–100. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[Dor94] S. E. Dorward. A survey of object-space hidden surface removal. *Internat. J. Comput. Geom. Appl.*, 4:325–362, 1994.

[EM98] Carl Erikson and Dinesh Manocha. Simplification culling of static and dynamic scene graphs. Technical Report TR98-009, Department of Computer Science, UNC-Chapel Hill, 1998.

[FDFH90] J. D. Foley, A. Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1990.

[FKN80] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 124–133, July 1980.

[GCS91] Ziv Gigus, John Canny, and Raimund Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transaction on Pattern Matching and Machine Intelligence*, 13(6), 1991.

[Geo95] C. Georges. Obscuration culling on parallel graphics architectures. TR 95-017, Department of Computer Science, UNC-Chapel Hill, 1995.

[GK93] Ned Greene and M. Kass. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[GK94] Ned Greene and Michael Kass. Error-bounded antialiased rendering of complex environments. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 59–66. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[GM90] Ziv Gigus and Jitendra Malik. Computing the aspect graph for the line drawing of polyhedral objects. *IEEE Transaction on Pattern Matching and Machine Intelligence*, 12(2), 1990.

[Gre96] Ned Greene. Hierarchical polygon tiling with coverage masks. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 65–74. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[HMC+97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proc. of ACM Symposium on Computational Geometry*, pages 1–10, 1997.

[Hop96] Hugues Hoppe. Progressive meshes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[Jon71] C. B. Jones. A new approach to the 'hidden line' problem. *Computer Journal*, 14(3):232–237, August 1971.

[KMGL96] Subodh Kumar, Dinesh Manocha, William Garrett, and Ming Lin. Hierarchical back-face computation. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 235–244, New York City, NY, June 1996. Eurographics, Springer Wein. ISBN 3-211-82883-4.

[Lat94] R. Latham. Advanced image generator architectures. Course reference material, 1994.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 199–208. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[LG95] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[Lor94] Loral. GT200T Level II image generator product overview, 1994.

[McK87] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

[Mue95] C. Mueller. Architectures of image generators for flight simulators. TR 95-015, Department of Computer Science, UNC-Chapel Hill, 1995.

[Mul89] K. Mulmuley. An efficient algorithm for hidden surface removal. *Computer Graphics*, 23(3):379–388, 1989.

[Nay92] Bruce F. Naylor. Partitioning tree image representation and generation from 3D geometric models. In *Proceedings of Graphics Interface '92*, pages 201–212, May 1992.

[NNS72] Martin E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. In *Proc. ACM Nat. Mtg.* 1972.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[PD90] H. Plantinga and C. R. Dyer. Visibility, occlusion and the aspect graph. *Internal J. Comput. Vision*, 5(2):137–160, 1990.

[Poc92] M. Pocchiola. The visibility complex. In *Proc. 5th Franco-Japanese Days on Combinatorics and Optimizatio n*, page ??, 1992.

[PV95] M. Pocchiola and G. Vegter. Computing the visibility graph via pseudotriangulations. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 248–257, 1995.

[RB93] J. Rossignac and P. Borrel. Multi-resolution 3D approximation for rendering complex scenes. In *Second Conference on Geometric Modelling in Computer Graphics*, pages 453–465, June 1993. Genova, Italy.

[RH94] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[Rob63] Lawrence G. Roberts. Machine perception of three-dimensional solids. TR 315, Lincoln Lab, MIT, Lexington, MA, May 1963.

[(SO94] J. C. Chauvin (SOGITEC). An advanced z-buffer technology. In *IMAGE VII*, pages 76–85, 1994.

[SSS74] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *Journal of the ACM*, March 1974. summarized in "Naval Research Reviews", June 1975, pp. 21–23.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

[TH93] Seth Teller and Pat Hanrahan. Global visibility algorithms for illumination computations. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 239–246, 1993.

[TP75] S. L. Tanimoto and Theo Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[TS91] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 61–69, July 1991.

[Tur92] Greg Turk. Re-tiling polygonal surfaces. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 55–64, July 1992.

[WA77] K. Weiler and K. Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11(2):214–222, July 1977.

[War69] J. Warnock. A hidden-surface algorithm for computer generated halftone pictures. Technical Report TR 4–15, NTIS AD-733 671, University of Utah, Computer Science Department, 1969.

[Wat70] G. S. Watkins. A real time visible surface algorithm. Technical Report UTEC-CSc-70-101, NTIS AD-762 004, Computer Science Department, University of Utah, Salt Lake City, UT, June 1970.

[Wil83] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 1–11, July 1983.

[WREE67] C. Wylie, G. W. Romney, D. C. Evans, and A. C. Erdahl. Halftone perspective drawing by computer. In *FJCC*, pages 49–58, Washington, DC, 1967.

[XESV97] Julie C. Xia, Jihad El-Sana, and Amitabh Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), April–June 1997. ISSN 1077-2626.

[YR96] R. Yagel and W. Ray. Visibility computations for efficient walkthrough of complex environments. *Presence*, 5(1):1–16, 1996.