HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science

Timo Aila

SurRender Umbra: A Visibility Determination Framework for Dynamic Environments

Master's Thesis
October 10, 2000

Supervisor: Tapio Takala
Professor of Interactive Digital Media

Instructor: Ville Miettinen
CTO, Hybrid Holding, Ltd.

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Timo Aila		
Name of the thesis:	SurRender Umbra: A Visibility Determination Framework for Dynamic Environments		
Date:	October 10, 2000	**Pages:**	14+140
Department:	Department of Computer Science	**Professorship:**	Tik-111
Supervisor:	Tapio Takala		
Instructor:	Ville Miettinen		

This thesis evaluates the practical usability of proposed visibility algorithms in addition to related data organization models and their traversal. The primary focus of this thesis is on conservative visibility determination and occlusion culling.

New concepts introduced include conservative occlusion-preserving silhouette extraction and visible point tracking. A new incremental variation of Hierarchical Occlusion Maps is presented. Solutions to several portal traversal problems are given. Occluder selection is analyzed in detail and a new adaptive algorithm is presented. A preliminary taxonomy of conservative visibility determination is introduced.

As a result, a platform-independent visibility determination framework is developed for dynamic scenes.

Keywords: computer graphics, optimization, visibility determination, occlusion culling, occluder selection, real-time

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä:	Timo Aila		
Työn nimi:	SurRender Umbra: Järjestelmä dynaamisten ympäristöjen näkyvyysratkontaan		
Päivämäärä:	10. Lokakuuta 2000	**Sivumäärä:**	14+140
Osasto:	Tietotekniikan osasto	**Professuuri:**	Tik-111
Työn valvoja:	Tapio Takala		
Työn ohjaaja:	Ville Miettinen		

Tämä diplomityö arvioi aiemmin esitettyjen näkyvyysalgoritmien ja niihin liittyvien tietorakenteiden käyttökelpoisuutta käytännön tilanteissa. Diplomityö keskittyy erityisesti algoritmeihin, jotka poistavat virtuaaliympäristöstä malleja jotka eivät näkyisi lopullisessa kuvassa vaikka ne piirrettäisiin. Kyseisiä tekniikoita hyödyntämällä voidaan visualisoida aiempaa huomattavasti suurempia virtuaaliympäristöjä.

Tässä diplomityössä esitetään useita uusia algoritmeja, joiden avulla näkyvyysratkontaa voidaan nopeuttaa. Erityisesti keskitytään ratkaisumalleihin, joiden avulla vapaudutaan laitteistosidonnaisuudesta.

Muita esineitä peittävien kappaleiden valinta on eräs näkyvyysratkonnan tärkeimmistä ongelmista ja siihen esitetään täysin uusi oppiva algoritmi. Siluettien käyttö piirtoprimitiiveinä on työn tärkeimpiä oivalluksia ja siihen liittyen esitetään useita uusia algoritmeja.

Lopputuloksena esitetään laitteistoriippumaton arkkitehtuuri näkyvyysratkontaan. Vastaavanlaisia tuotteita ei ole aikaisemmin julkaistu.

Avainsanat: tietokonegrafiikka, optimointi, näkyvyysalgoritmit, reaali-aikaisuus

Acknowledgments

Ville Miettinen for precious advisory work, co-designing the new algorithms and implementing a large part of the SurRender Umbra architecture.

Janne Hellstén for helping with several portal issues and implementing the Umbra Visualizer run-time debugging environment with Petri Kero.

Konsta Hansson for drawing the final versions of the figures.

The whole personnel at Hybrid Holding, Ltd for supporting the SurRender Umbra project.

Special thanks to Janne Kontkanen, who helped designing the first version of the Umbra architecture, and to John Airey, Theodore Jump, Matthias Wloka and Hansong Zhang for reviewing the text, as well as to Jaakko Lehtinen and Jussi Räsänen, who helped with many computational geometry problems and other practical issues. Saku Lehtinen for helpful ideas on portals.

Additional thanks to people at AMD, IBM, Hewlett Packard, Intel, Microsoft and NVidia, who provided information and hardware we could not have otherwise obtained.

The research and development of SurRender Umbra was partially financed by Tekes, the National Technology Agency of Finland.

Helsinki, October 2000

Timo Aila

Copyright Notice

Contents

List of Figures

Chapter 1

Introduction

1.1 Exact Visibility

Visibility determination has been an integral part of computer graphics since the 1960s. In order to correctly display three-dimensional graphics on a flat display, two fundamental requirements arise. First, proportions and relative sizes of objects must behave as we observe them in the real world. This is solved by using a perspective projection. Secondly, the closest object at any given screen pixel should be drawn.[1] An analytical solution that is not tied to any fixed resolution is called *continuous exact visibility* whereas *point sampled exact visibility* resolves the visibility for a fixed-resolution screen. Insufficient resolution causes *aliasing*. For our purposes continuous and point sampled algorithms solve the same problem, and despite their fundamental difference, we consider both categories as *exact visibility algorithms*. Numerous algorithms for solving the problem have been proposed, and finally in 1992 Charles Grant presented a complete taxonomy of exact visibility algorithms. At present, practically all graphics hardware solve the exact visibility determination by using the Z-buffer algorithm [15].

1.2 Conservative Visibility

Having the exact visibility problem solved, the output image is guaranteed to be correct as long as all contributing geometry is processed. If we somehow know that a car is hidden by a building, there is clearly no point in processing the car. Conservative visibility algorithms try to find objects that do not contribute to the output image. Furthermore, there is a potential possibility for optimizing complex physics, AI and collision calculations, if there is no-one to check the results. Philosophers in ancient Greece were puzzled with the question of whether a falling tree makes a sound, if there is no-one listening. In virtual environments, it is both safe and wise to say that it does not have to. Indeed, it is beneficial to declare that nothing in the world needs to happen, or to exist, if it is not observed.

During the 1990s the research emphasis moved strongly towards conservative visibility algorithms and occlusion culling. Compared to exact visibility algorithms, they are significantly more complicated, and no single algorithm is expected to solve all types of applications satisfactorily.

[1] If antialiasing is being performed, multiple objects can contribute to a single pixel.

1.3 Scope of This Thesis

Our goal is to create a platform-independent, general purpose visibility determination library, SurRender Umbra [2]. The library must not require any pre-processing or static data structures. However, it should benefit from *a priori* visibility information, if such knowledge is available. The architecture must scale from marginal to huge depth complexity, from handful to millions of objects and from thousands to hundreds of millions of polygons.

We evaluate existing algorithms according to the requirements listed in the previous paragraph and review new contributions in detail. First, we consider exact visibility determination algorithms because the conservative algorithms inherit algorithmic properties from the exact ones. We briefly list some of the non-conservative research areas, since both conservative visibility determination and non-conservative methods are required for visualization of arbitrarily complex scenes. [3]

We do not go into any depth with the nature of visibility; an interested reader should refer to [30]. We do not consider area visibility and related issues of radiosity, soft shadows and penumbrae. Discussion of algorithms used to handle dynamic spatial databases is limited to references to the ones used in Umbra. This thesis concentrates mainly on visibility issues related to *direct image formation*; higher-order visibility, such as reflections, is considered only in the context of portal rendering.

1.4 Visibility Pipeline

The process of visibility determination can be divided into the following sub-categories.

1. *View Frustum Culling* removes objects that are outside a virtual camera's view volume. In other words it resolves whether an object can contribute to the final image when not obscured by other objects.

2. *Occlusion Culling* removes objects or parts of objects that are obstructed from view by other objects. This test involves more than one object, and is therefore a global problem and significantly harder than view frustum culling. The primary focus of this thesis is on occlusion culling.

3. *Back-face culling* removes parts of the object that are facing away from the camera and therefore do not contribute to the final image.

4. *Exact Visibility Determination* resolves a single or multiple contributing primitives for each output image pixel.

5. *Contribution culling* removes objects that would have only a minor impact on the final image. Whereas the other four sub-categories leave the output image intact, contribution culling introduces artifacts. The term *aggressive culling* is sometimes used to refer to such non-conservative methods.

[2] **www.surrender3d.com**
[3] As the visible portions of the scene can be arbitrarily complex.

1.5 Target Applications of the Library

The library architecture presented in this thesis is designed to improve performance in several types of target applications. Unlike most of the research, practical applications do not generally have scenes with millions of polygons and a high depth complexity. Dominant portion of interactive 3D visualization happens in video games and in other VR applications. The library design is targeted to work well also with the modest scene complexities of video games. It is worth noticing that occlusion culling algorithms generally improve their performance as the scene complexity increases, and not many of them are of practical value with current video game complexities. The following target application types are considered:

- Architectural walkthroughs and indoor environments
- Urban environments
- Flight simulators
- Terrain visualization
- Fully dynamic scene visualization
- Applications with a pre-rendered background.

1.6 Plan of Development

Chapter 1 is this introduction, including the problem definition, the scope of this thesis, types of target applications, and the plan of development. Evaluation of previous work on both exact and conservative visibility determination is presented in chapter 2. New contributions are reviewed in detail in chapter 3. Chapter 4 describes the overall library architecture with discussion on design solutions made. Chapter 5 summarizes the results and conclusions. Finally, possible extensions to the library architecture and potentially useful pre-processing tools are listed in chapter 6.

1.7 New Algorithms and Concepts

- Temporally coherent occlusion-preserving silhouette extraction (section 3.8.1).
- Occlusion Write Postpone Queue (section 3.7).
- Validation Buffer (section 3.10.7).
- Construction-Time Contribution Culling (section 3.10.5).
- An incremental variation of Hierarchical Occlusion Maps (section 3.9).
- Visible Point Tracking (section 3.4).
- Hierarchical Depth Estimation Buffer (section 3.11).
- Adaptive Occluder Selection with Feedback (section 3.3.2).
- Portal PVS (section 3.14).

- Preliminary Taxonomy of Conservative Visibility Algorithms (3.2).
- Spatial database nodes as dynamic Virtual Occluders (section 3.13).

Chapter 2

Previous Work

In this chapter we first state our primary design principles for algorithms. Then we evaluate exact visibility determination algorithms by following the taxonomy of basic visibility algorithms [48] shown in figure 2.3.2. Exact visibility algorithms are reviewed, since they can be seen as the basic building blocks of conservative visibility algorithms, and because conservative algorithms generally require an exact visibility algorithm as a back-end to visualize the scene correctly.

In 2.4 an overview and analysis of conservative visibility algorithms and their practical usability is presented.

Both conservative visibility determination and non-conservative methods are required for visualization of arbitrarily complex scenes. Non-conservative methods are briefly discussed in 2.5. They are seen as an important future extension to our framework, and should therefore be considered when making design decisions.

Data organization models for spatial databases are briefly discussed in 2.6.

Antialiasing is outside the primary scope of this thesis, but since visibility determination and antialiasing are tightly coupled, we refer to aliasing issues in a few places in the text.

2.1 Fundamental Concepts

Bounding Volumes

Instead of performing computations directly with the actual geometry, it is often beneficial to surround an object with something simpler and to perform the computation first using the simplified bounding volume. If the bounding volume is found to be hidden, all geometry inside it, however complicated, is hidden as well [114]. Usually this surrounding object would be a bounding sphere, an axis-aligned bounding box (AABB), an oriented bounding box (OBB), a k-dop [1] or the convex hull of a mesh.

[1] Construction of a k-dop begins by selecting K directions that cover the entire direction-space as evenly as possible. 6-dop is a box. Vertices are projected onto the direction vectors and the maximal projection on each direction is chosen. These K maximal distances represent the k-dop.

Occluders and Occludees

An *Occluder* is an object that obstructs other objects from the viewpoint of the camera. Any visible object is a potential occluder. *Occlusion power* is a characteristic estimating how much geometry an object manages to obstruct from the viewpoint of the camera. An *occludee* is an object obstructed by one or more objects. An *occluder set* is a list of all objects that were chosen as occluders. The optimality of an occluder set can be evaluated using a *cost-benefit function*, which compares the time lost by using the set of objects as occluders and the processing time won by avoiding processing the occluded geometry. Typically the losses and gains have to be approximated. Finding the *optimal occluder set* is view-dependent and an NP-hard problem [2], therefore an approximation must be used.

Virtual Occluders

Law and Tan [70] made a superb observation that any hidden [3] object or volume in the scene can act as an occluder. Occluding geometry can be replaced with considerably simpler virtual occluders.

A classic example is a dense forest where visibility is only 15 meters. If a virtual occluder could be placed at the distance of 15 meters from the camera, all the trees actually blocking the visibility could be ignored as occluders. Savings can be tremendous. Virtual occluders mainly reduce occluder setup time, and in some cases can provide pre-computed occluder fusion.

Schaufler *et al.* [94] discretize the entire scene and flood fill the occupied regions of space to create a discrete and simple presentation of combined occluders. Filling the interiors of non-manifold, non-closed surfaces is potentially an ambiguous problem, and certainly a challenging task to perform dynamically. This approach can be seen as a way of compactly expressing Potentially Visible Sets (see chapter 2.1).

The implementation of Koltun *et al.* [67] is limited to 2.5D and the authors present an algorithm for calculating the augmented umbrae from an area. Creating such umbrae in 3D is difficult, but would be the key for constructing temporally coherent virtual occluders. Such occluders would in fact solve many important issues. We believe that efficient construction of virtual occluders has become one of the most important research topics of the visibility research community.

Bernardini *et al.* [11] use octree faces as virtual occluders. They create the virtual occluder octree at a pre-processing phase. We adopt the principles of this approach and extend it to dynamic scenes in section 3.13.

Hierarchical bounding volumes are commonly used as virtual occludees. Therefore the introduction of virtual occluders provided a missing piece to a puzzle.

Occluder Selection

Selecting a good set of objects as occluders is a fundamental requirement for an efficient occlusion culling algorithm. Regardless of their importance, occluder selection algorithms have received only modest attention. The usual classification criteria are based on:

1. *Size*: Small objects are bad occluders.

[2] With N objects there are O($N!$) different occluder sets. Due to *occluder fusion*, the occlusion power of an object depends on possibly all other objects. Thus finding the optimal set is a combinatorial problem.

[3] Hidden object is an object that is occluded by other objects.

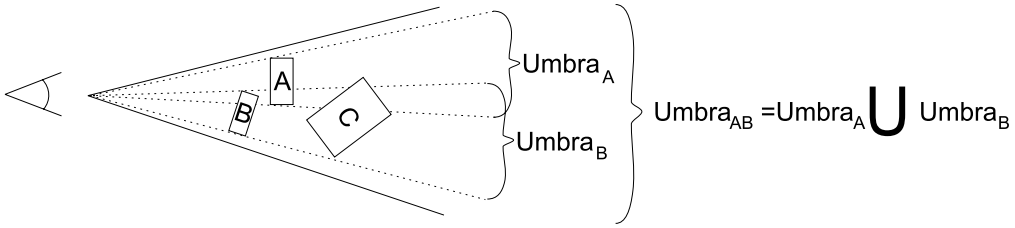

Figure 2.1 The principle of occluder fusion. Objects A and B can together occlude object C, even though neither of them can occlude C on their own.

2. *Complexity*: Complex objects are expensive to use as occluders.
3. *Redundancy*: Certain objects, such as a clock on a wall, do not contribute significantly to the overall occlusion. These objects are sometimes called *detail objects*.

The question of a good occluder set is *completely* context-dependent and none of the above criteria can provide actual information. A tennis ball can occlude the Statue of Liberty. When there is no feedback from the occlusion system, no context-dependent information can be obtained. We can go even further and to claim that the occlusion culling algorithm must be *incremental* to provide such information. See section 3.3 for a cost-benefit analysis of occlusion culling.

Occluder Fusion

Merging multiple objects into a single occluder always has greater or equal occlusion power than the sum of the separate occlusion powers. In many cases the difference is significant. In case of figure 2.1 neither occluder *A* nor *B* alone can obstruct object *C*, but together they can. To illustrate the idea, consider a forest. A single tree does not occlude much, but a hundred trees may be able to occlude large portions of the screen.

Shadows of point light sources and the hidden regions of a camera's view frustum are schematically identical. Thus the concept of shadow volumes, introduced by Crow [29], has an analogue in visibility determination, *occlusion volumes*. In fact, all algorithms applicable to shadow computation of point light sources can be used for visibility determination and vice versa.

Another aspect of occluder fusion are consecutive semi-transparent surfaces. Even if it is possible to see through each of the surfaces individually, stacking enough of these surfaces creates a virtually opaque surface[4].

Potentially Visible Sets

The potentially visible set (PVS)[5][3] is a conservative superset of the actually visible set (AVS) of primitives or objects. We call the initial PVS consisting of the entire scene database PVS_{scene}. View frustum culling bounds the PVS_{scene} to view-dependent $PVS_{frustum}$, where usually $\|PVS_{frustum}\| \ll \|PVS_{scene}\|$. If the scene has even modest depth complexity, $\|PVS_{frustum}\| \gg \|AVS\|$ holds. The purpose of occlusion culling is to reduce $PVS_{frustum}$

[4]Limited by the color resolution of the output device.
[5]A PVS contains *all* visible objects and some hidden ones.

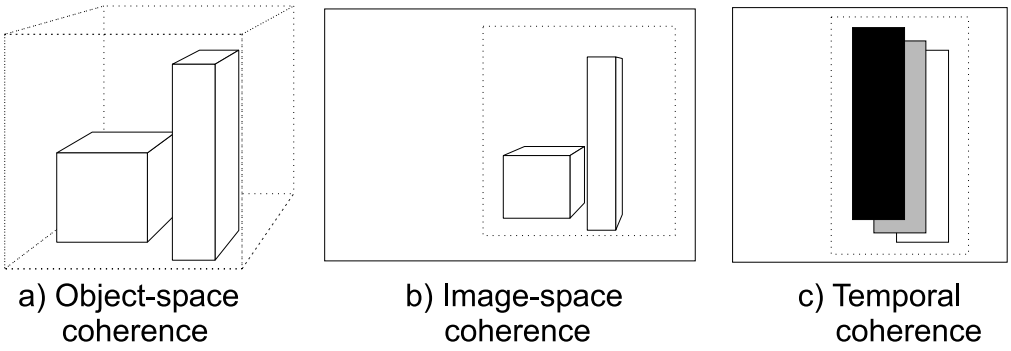

Figure 2.2 Object-space, image-space and temporal coherence

to $PVS_{occlusion}$, that better matches AVS. Exact visibility algorithms are needed to resolve AVS from any given PVS. Conservative visibility algorithms minimize the input $||PVS||$ of the exact algorithm.[6]

2.2 Design Principles

Coherence

As recognized by Sutherland, Sproull and Schumacker some twenty-five years ago in their classic survey of visible-surface algorithms [104], exploiting coherence is the key for designing efficient visibility algorithms. A more recent study by Gröller and Purgathofer [53] lists more than a thirty different types of coherence present in computer graphics. Of these, our primary emphasis is on object-space, image-space and temporal coherence (figure 2.2).

Object-space coherence is based on some known relationships between objects or parts of an object. Adjacent regions of space are likely to be occupied by the same objects. Therefore a single computation may suffice to say something definitive about a group of objects or all primitives of an object. If the bounding volume of an object intersects only the left plane of the view frustum, its primitives need only be tested against the left plane. In a similar fashion, if a bounding volume is determined to be hidden, the actual object inside the volume does not have to be processed.

Image-space coherence is the view-dependent analogue of object-space coherence. Object-space coherence transforms into image-space coherence on the view plane under well-behaved projections. The primary difference is that a projection also *creates* coherence, since distant objects may form coherently behaving groups on the view plane. All operations taking place in 2D are considered to be exploiting image-space rather than object-space coherence.

Temporal coherence exists when there is a smooth change in the motion of either the objects or the camera. Exploiting temporal coherence can open possibilities for calculating results that are valid during a longer period of time, not just a single moment. We are using the word 'can' here, because temporal coherence is very hard to exploit properly and even though many algorithms claim to be temporally coherent, they usually benefit only marginally from the previous results. Literature is extremely ambiguous with the precise differences between temporal and *frame coherence*. Therefore

[6] Some of the inequalities do not hold if higher-order visibility, i.e. reflections, is considered.

we classify temporally coherent properties to the following categories:

1. An algorithm can decide that a calculated result is valid until a specific condition is broken. The condition can be a time period, a point inside a given volume or another preferably simple condition. In interactive worlds time alone is hardly ever a sufficient condition.

2. An algorithm can use results from the previous frames to speed up computations of the current frame. Due to cache architectures of modern computers, basically every algorithm benefits from the work done in the previous frames.

3. Algorithms that calculate results always from scratch.

Output-Sensitivity and Hierarchical Processing

In 1976 Clark [22] set an objective to design a visibility algorithm "in which the computation time increased linearly with the visible complexity of the scene".[7] A detailed model of New York could contain billions of objects, and yet only a small fraction would be visible from any given location. Selecting the visible objects without testing the hidden ones is clearly possible only, if the set of visible objects has been stored during a pre-process. In absence of pre-computed visibility information, the best one can do is to organize the objects into bounding volume hierarchies and use object-space coherence to apply the results downwards in the hierarchy. If Manhattan is hidden, there is no need to test the visibility of Carnegie Hall and definitely not a chair inside it. As first noted by Warnock [112], hierarchical processing can be thought of as a logarithmic search for primitives of interest. Maintaining output-sensitivity is a strict constraint. It should be noted that any operation accessing more objects than a linear function of visible objects violates the principle of output-sensitivity. Hierarchical processing has been found very efficient in acceleration of both object-space and image-space operations.

Lazy Evaluation

Nothing has to be processed before the result is needed. This principle has the following benefits:

1. Simplifies the logic outside the component. When not all control paths require all results, no performance penalty occurs.

2. Improves performance by skipping work.

3. Improves cache-coherence by executing consecutive operations on the same data.

Portability

The algorithms should not rely on specific hardware or rarely supported features.

[7] Clark limits the output-sensitivity to *linear* processing of visible complexity of the scene. As a result, more complex operations such as quicksort do therefore violate the original definition of output-sensitivity.

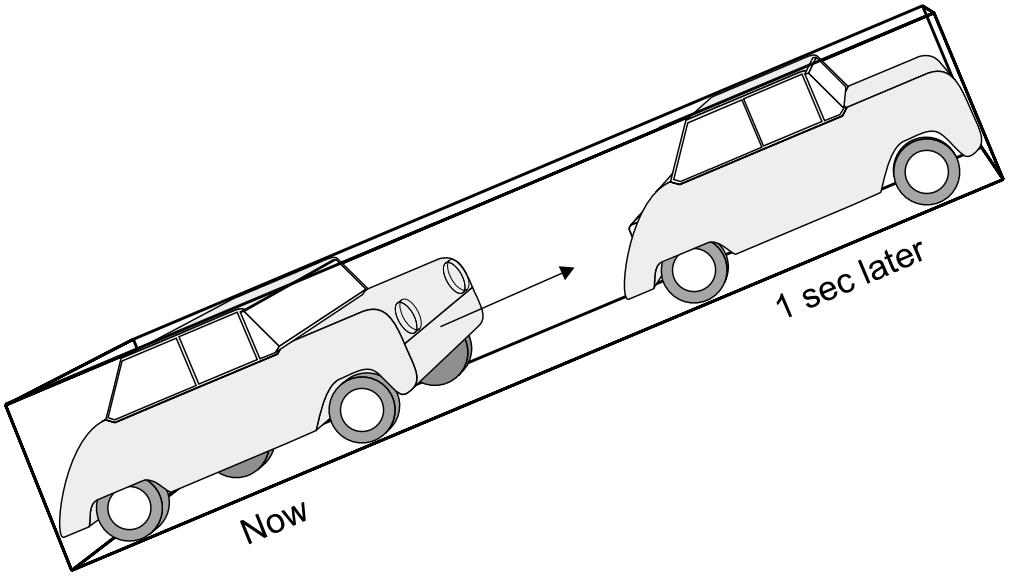

Figure 2.3 Temporal bounding volume of a car over the duration of one second

Generality

The algorithms should be applicable to arbitrary models. Limiting the applicability to convex models, models with large polygons, or architectural models are considered unacceptable.

Robustness

The algorithms must not be prone to floating-point errors. Computational geometry algorithms are often prone to degeneracies, discontinuities and other problems.

Applicability to Dynamic Environments

If we constrain the world to be fully static, everything becomes much simpler. First of all, the need for physics, dynamics and collision detection[8] are all gone. Visibility determination and spatial data structures also become significantly easier. Unfortunately, such a constraint is only suitable for walkthroughs, and no decent virtual environment can be restricted to be static. However, the fact that most objects are indeed static is an important observation. A house does not move, but when the window gets broken, the house is a dynamic object for a short moment. Bearing the above in mind, we conclude that classifying something as static must be done dynamically, and the fact that many objects are static should be exploited. No two static objects can ever interact. Maintaining a dynamic hierarchical spatial index was thought impractical by several researchers during the past two decades, and finally shown feasible by Sudarsky in 1996 [102]. Indeed, the introduction of *temporal bounding volumes* (TBV) (figure 2.3) significantly reduces the amount of required database updates, and provides temporal coherence to bounding volume tests of dynamic objects. We consider algorithms relying solely on pre-processed static information unsuitable for practical use.

[8] Except for the camera itself.

2.3 Exact Visibility Determination

In this section some of the basic visibility algorithms are reviewed and their most significant features are underlined. Especially suitability for parallel hardware implementation is considered.

2.3.1 Important Algorithmic Properties

Hardware Parallelization

It is crucial that an algorithm is suitable for *hardware implementation*, which in turn implies the requirement of *parallelization*. Parallelization can be divided into image-space parallelization (subdivision) and per-object parallelization.

Combinatorial Blowout

Many algorithms consider variable amounts of data for determining the visibility of a single element. Such an approach can lead to the problem commonly referred to as *combinatorial blowout*. Algorithms suffering of combinatorial blowout are usually not suitable for hardware parallelization due to the complex logic needed.

Cache-Coherence

An algorithm should preferably be *cache-coherent*. Since current computer hardware only has a small amount of fast cache memory, an object should optimally enter the cache only once per frame. This problem is in fact getting worse all the time, as silicon speeds grow faster than bus speeds. To illustrate this problem, consider a scene of 100 objects each consisting of 1000 triangles. Such a scene can be interactively rendered with current graphics hardware. Sorting these 100.000 triangles is theoretically an O(n) operation[9] that will fall out of the cache memory on almost all consumer-level computers. As a result, global sorting would indeed be a dominant cost in rendering.

Incremental

An algorithm is *incremental*[10] if new primitives can be added to the current solution. In absence of this property, all primitives of the entire scene must be available at the same time. This is hardly ever practical. If the primitives are cached, and finally processed in a single batch, the principle of cache-coherence is violated and geometry $\leftrightarrow$ rasterization parallelism is lost.

2.3.2 Grant's Taxonomy

The first systematic study in exact visibility determination algorithms was published in 1973 and 1974 by Sutherland, Sproull and Schumacker[11] [104]. Since their study many new algorithms have

[9] With bucket sort, if cyclic overlaps are not handled properly. O(n^2) worst case if cyclic overlaps are handled.
[10] Some authors use the term *progressive* instead.
[11] SSS taxonomy.

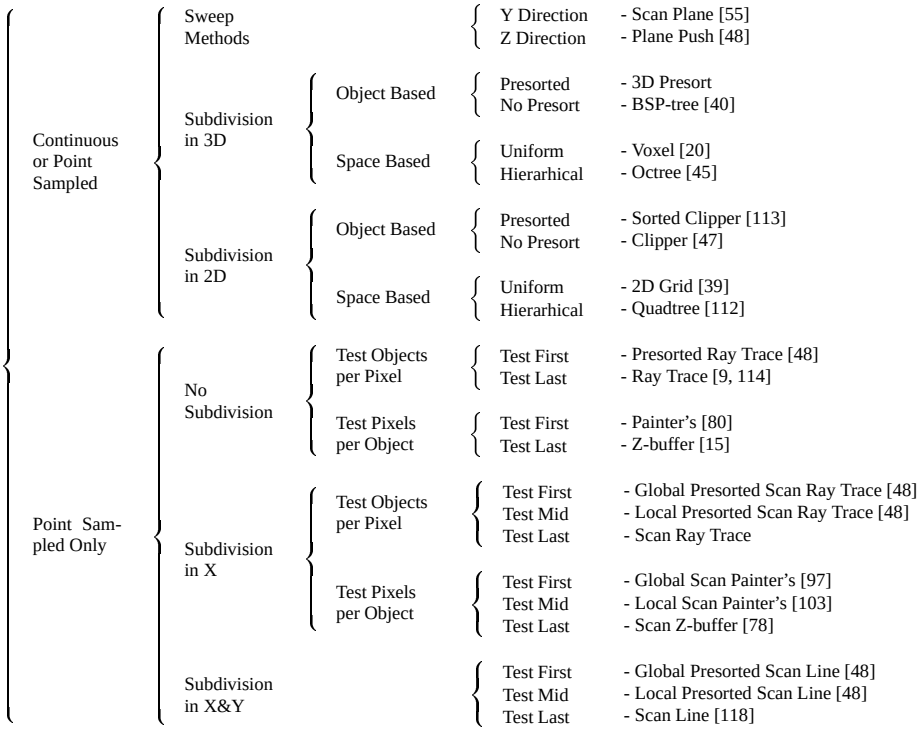

Figure 2.4 Complete taxonomy of exact visibility algorithms according to Grant[48]

been published and Grant presented his revised taxonomy in 1992[48]. This taxonomy did not consider conservative visibility algorithms, which have been the major area of research in the 1990s. Nor did it contain the fundamental new hierarchical algorithm HZ-buffer (section 2.4.2), which was introduced a few years later. Unfortunately it does not seem possible to add HZ-buffer into Grant's taxonomy.

2.3.3 Painter's Algorithm

An artist creating an oil painting first paints the background colors, then the distant objects and finally the foreground objects are painted on top of everything. Painter's algorithm[80] is motivated by this behavior. First the objects are sorted in order of decreasing depth, then drawn starting from the back.

In the first pass, primitives are sorted according to their largest depth values. Primitive P with the largest depth value is compared with other primitives for depth overlap. If no overlap occurs, the primitive is drawn. In the case of a depth overlap, additional tests are required to solve whether any of the primitives should be reordered. Reordering is unnecessary if any of the following tests are true.

1. The 2D screen-space bounding rectangles of the two primitives do not overlap.

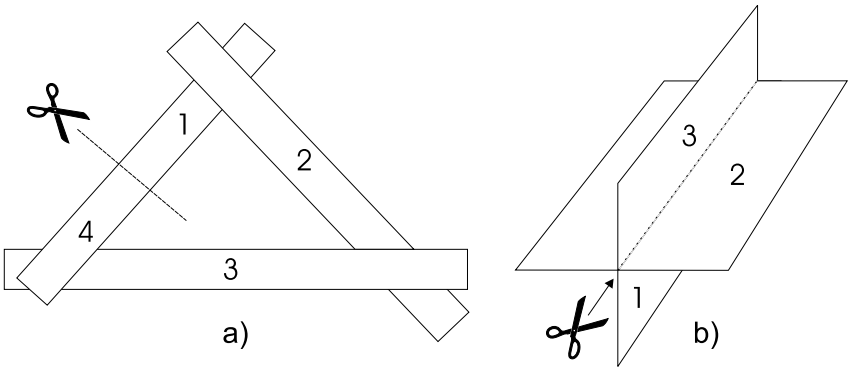

Figure 2.5 Classic problem cases of the Painter's algorithm. The numbers indicate the correct drawing order.

2. Primitive P is completely behind the overlapping primitive.
3. The screen projections of the two primitives do not overlap.

Reordering can get stuck in an infinite loop when two or more primitives alternately obscure each other (figure 2.5)[12]. To avoid such situations, the primitives are flagged as they are reordered to a farther depth position. When attempting to reorder a flagged primitive, the primitive is split into two parts and the procedure continues.

Advantages

1. Primitives with transparencies are handled correctly.

Drawbacks

1. Combinatorial blowout due to sorting process.
2. All primitives must be considered at the same time, therefore the algorithm is not incremental.
3. Hardware parallelization is difficult and would be coarse-grained.
4. Every pixel in every primitive is processed, resulting in a lot of unnecessary work.

2.3.4 Z-buffer

In addition to the frame buffer, another equal-sized buffer holding depth values for each pixel is kept. First the depth buffer is initialized to the farthest possible value. For every pixel in every primitive, we test whether the pixel is located closer to the camera than the current depth buffer value. If the test fails, the current primitive pixel is not visible. Otherwise the depth buffer value is updated, and the pixel is drawn into the frame buffer (figure 2.6).

[12] Numbers on the primitives indicate the correct drawing order.

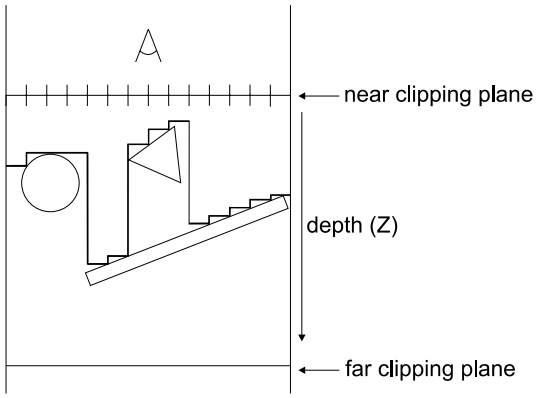

Figure 2.6 Z-buffer principle

Advantages

1. A single primitive can be considered at a time. Therefore Z-buffer is incremental and does not suffer from combinatorial blowout.
2. Different types of geometric primitives can be used together.
3. Parallelization is trivial via subdivision.
4. Rasterization is image-space coherent. The depth values can be computed incrementally.
5. Logic required for depth coordinate interpolation is the same as for all other surface parameters, and is therefore readily implemented into pixel hardware pipelines.
6. Can be generated from real-life imagery [1] [13]
7. The depth buffer can be stored along with the output image to allow further modifications.

Drawbacks

1. Every pixel in every primitive is processed, resulting in a lot of unnecessary work done in densely occluded scenes. [14]
2. Additional memory requirements per pixel [15], although this drawback is constantly diminishing as memory becomes cheaper.
3. Transparent primitives are handled incorrectly.

Usually opaque primitives are processed first, and depth buffer updates are disabled before processing back-to-front sorted transparent primitives. To handle transparencies correctly, the sorting stage should be implemented using the Painter's algorithm. In practice this is hardly ever done, and some combinations of transparent primitives are indeed drawn incorrectly. These effects do not show

[13] Minolta VIVID 700 3D digitizer grabs a 200x200 depth buffer in addition to a 400x400 color buffer in 0.6 seconds.
[14] *Deferred shading* can be used to limit the unnecessary work to just the depth tests.
[15] Memory requirements can be bounded by dividing the screen to tiles but this removes some of the advantages.

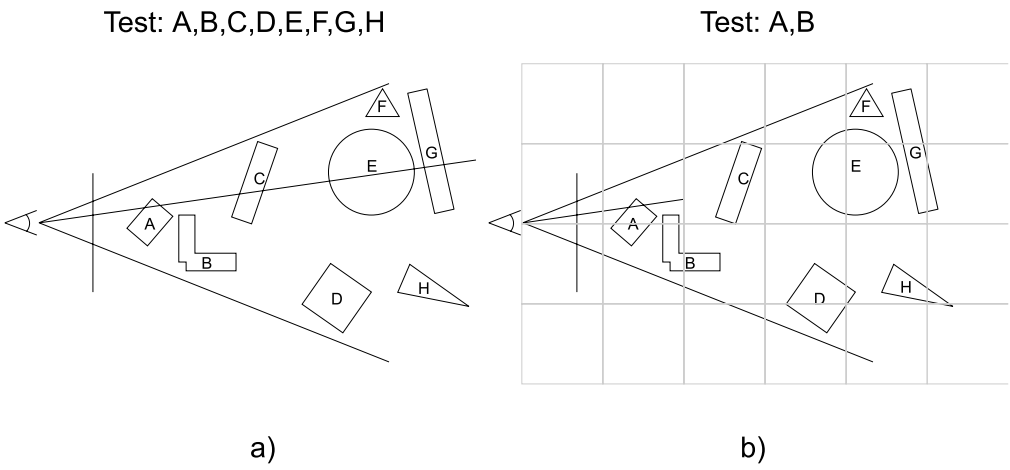

Figure 2.7 Raycast principle. A brute force approach (a) tests all objects for intersection with a ray. A more sophisticated approach (b) only seeks the first intersection and utilizes spatial subdivision to reduce the number of tests.

up very often, and can be almost fully avoided during the modeling phase by avoiding the known problem cases.

Mammen [74] was the first to propose a solution to make the Z-buffer handle transparencies correctly. Though theoretically a functional approach, his multi-pass algorithm is rarely used in practice.

A-Buffer[16] [14, 37] handles transparencies correctly, and additionally provides substantial improvements to the image quality. The algorithm is a simplified version of Catmull's algorithm[16], which clips polygons to pixel boundaries. A-Buffer is currently used by Pixar in their rendering system called REYES[17][25]. The basic idea is to store more than one contributing sample per output image pixel, and to finally combine all samples according to their coverage percentage, transparency and other surface attributes. The price to pay is a varying per-pixel memory usage. Variants with constant per-pixel memory usage have been proposed [64]. All algorithms in this category have a very high memory consumption, and are usually implemented by processing the output image in fixed-size tiles, which in turn violates the principle of incrementality.

An *item buffer* stores an ID for each output pixel. The ID is usually a 32-bit identifier indicating which object or primitive is visible at the output pixel. Potential uses of item buffers include the generation of PVSs, object/primitive picking and speeding up certain radiosity computations [24].

2.3.5 Ray Casting and Ray Tracing

Ray casting [9] casts a ray from the camera through the center of each output image pixel (figure 2.7). The output color is calculated from the primitive with the closest intersection. The brute-force version (a) would test every primitive against every ray, but we adopt here the somewhat more realistic approach (b) and expect an approximate front-to-back traversal, which is achieved usually with an octree [45], or some other object-space hierarchy[17]. More accurate results can be obtained by shooting multiple rays per pixel and filtering the resulting shading values. Whereas ray

[16] antialiased averaged area accumulation buffer.
[17] Renders Everything You Ever Saw

casting stops when the first intersection is found, ray tracing [114] continues recursively by shooting reflected and refracted rays from the intersection points. The contributions of the recursive rays are finally accumulated to get a single color per pixel.

Advantages

1. Exploits object-space coherence by hierarchically culling hidden geometry. Works efficiently in densely occluded scenes.
2. Transparent primitives are handled correctly.
3. Highly parallelizable.
4. Handles complex primitive types such as spheres, metaballs and parametric surfaces.

Drawbacks

1. Since the depth of the closest intersection is not stored, new primitives cannot be added into an existing solution.[18] Therefore ray casting is not incremental.
2. Intersection calculations are expensive.

There has been exhaustive research on data structures for speeding up ray tracing and trying to merge advantages of Z-buffer and ray casting.

ZZ-buffer by Salesin and Stolfi [88] approaches this by subdividing the output image into image-space tiles. A first pass assigns primitives into tiles, and a second pass ray casts the primitives assigned to the tiles. While being able to exploit image-space coherence, this approach loses the most significant advantage of ray casting: exploiting the available object-space coherence.

An alternative approach would be to use a tile-sized Z-buffer or A-Buffer. Whenever the ray intersects a primitive, the depth values of the primitive are applied to the Z-buffer values of every pixel it occupies[19]. The ray casting intersection depths are processed normally, but a single primitive has to be considered only once per tile since its depth values are already cached into the local Z-buffer. This results in exploitation of both image-space and object-space coherences.

2.3.6 Subdivision

Area subdivision by Warnock [112] is a divide-and-conquer algorithm, which recursively subdivides the output image until each rectangle is fully covered by a single primitive or no primitives at all. Unless terminated earlier, ultimately 1x1 pixel rectangles are tested.[20]

Starting with a rectangle of the total output image area, we recursively perform tests to determine whether we should subdivide the area into quadrants. Each primitive has one of the following relations with the current rectangle (figure 2.8).

[18] This can be achieved by combining ray casting with Z-buffer and is currently used for example in the previewer of 3D Studio Max™.

[19] Inside the tile.

[20] Assuming no antialiasing is performed.

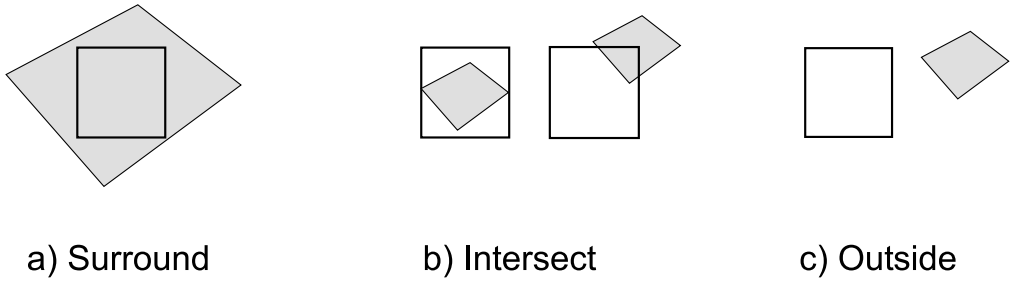

Figure 2.8 Warnock subdivision classification cases

1. The primitive completely encloses the rectangle (surrounding).
2. The primitive is partially or completely inside the rectangle (intersecting).
3. The primitive is completely outside the rectangle (outside).

The rectangular area does not have to be subdivided further if any of the following are true:

1. All primitives are outside the rectangle.
2. Only one primitive surrounds or intersects the rectangle.
3. A single surrounding primitive obscures all other primitives.

Advantages

1. Exploits image-space coherence by hierarchically culling hidden geometry.

Drawbacks

1. Scenes with a lot of small primitives suffer from substantial overhead due to subdivision. This corresponds to *bucket rendering* with very small buckets. The overheads involved in bucket rendering have been studied in [28, 21].

While Warnock's area subdivision algorithm has never been widely used in its original form, its principles have been utilized in several newer algorithms. Similar subdivision schemes have been used by Greene [50, 49] and Naylor [79].

Warnock's subdivision is always axis-aligned, and therefore may not be very efficient on diagonal primitive edges. To overcome this, Weiler and Atherton [113] presented a scheme where primitives are first sorted into front-to-back order, then the subdivision proceeds along the edges of primitives. This results in very awkward clipping problems and the algorithm has only theoretical and historical value.

2.3.7 Scan-Line Algorithms

Scan-line algorithms [118, 97, 103, 78, 48] are not fundamentally different from previous algorithms, but simplify the problem by solving visibility in 1D instead of 2D. It could be stated that these are mainly data structures used to simplify the problem domain. Usage of scan-line algorithms usually reduces memory usage, but requires an additional sorting pass. Many of the scan-line algorithms can also be implemented using tiles.[21]

2.3.8 Beam Tracing

Beam tracing by Heckbert and Hanharan [56] is a hybrid technique of ray tracing and Weiler and Atherton's sorted clipper (see section 2.3.6).

Whereas ray casting casts a ray through the center of each output image pixel, beam tracing only casts a single viewport-sized beam. The beam is recursively clipped against the scene geometry and consequently several smaller beams result. While able to exploit both image-space and object-space coherence, beam tracing inherits the awkward clipping problems present in Weiler and Atherton's algorithm. Unfortunately, refraction is not a linear function and can therefore only be approximated by reflecting beams.

We shall refer to beam tracing later when discussing portals (section 2.4.8).

2.3.9 Frustum Casting

Frustum casting by Teller [106] is a hybrid technique of Warnock subdivision and ray casting.

The output image is subdivided in a fashion similar to Warnock subdivision, but instead of projecting geometry into image-space, the subdivided view is ray casted into the scene. From an analytical point of view the difference is significant. While Warnock subdivision needs to concern all primitives, frustum casting inherits the hierarchical culling property of ray casting and is therefore able to exploit object-space coherence in addition to image-space coherence inherited from subdivision. In scenes with a lot of small primitives, frustum casting converges to ray casting, and is therefore no longer able to exploit image-space coherence.

2.3.10 Immediate Mode Ray Casting

Immediate mode ray casting by Alex and Teller [4] is a hybrid technique of Z-buffer and ray casting.

Assume we have a surface of the form $F(x, y, z) = 0$ and a bounding mesh surrounding it. We first test the bounding mesh against the Z-buffer and tag all visible pixels. We then cast rays towards all visible pixels and calculate intersections with the actual surface. This method could be generalized to work with any parametric surfaces and modifications to existing hardware are quite small.

[21] Image-space coherence is often better exploited by processing tiles than scan-lines.

2.3.11 Conclusion

SSS taxonomy concluded that Z-buffer and ray tracing were impractical brute-force algorithms. The memory requirements of Z-buffer were considered outrageous.

Currently Z-buffer is the *de facto* standard for exact visibility determination due to its simplicity and easy hardware implementation.

Hybrid Algorithms

Hybrid algorithms inherit algorithmic properties from their base algorithms. There is some similarity to object-oriented programming and inheritance. However, the process of inheriting algorithmic properties is not as straightforward as class inheritance. The following three cases are possible when a hybrid algorithm H is created from algorithms A and B.

1. A property of A (or B) is present in H.
2. A property present in A is lost due to a fundamental change in behavior.
3. A and B contain conflicting properties and the effect may be either canceled out, blended or inherited from either A or B.

The resulting set of properties can only be deduced by understanding the base algorithms. We shall use this model when presenting and analyzing new visibility algorithms.

Future Directions

The lack of proper antialiasing is still a big problem in real-time rendering. Antialiasing and visibility algorithms are tightly connected [52, 48]. Additionally motion blur, depth of field and proper surface shaders are needed to create images with a convincing appearance. These additional features are possible by using the A-Buffer, which is currently being used in Pixar's Photorealistic RenderMan[110, 8], a software widely used by the movie industry. Hardware implementation of A-Buffer is still years away due to its inherent memory consumption.

The accumulation buffer [82] is the current solution for improved image quality. The accumulation buffer is an extra buffer where the frame buffer can be accumulated. The only operations required for antialiasing, motion blur and depth of field are addition and scaling with a constant.

Proper motion blur and depth of field require at least 64 samples per pixel, which puts enormous pressure on geometry processing, pixel fill rate and visibility algorithms. With 64 samples per frame the frame rate is effectively 64 times than what is needed without the special effects. Temporally coherent algorithms can provide immense boosts to performance.

During the past years graphics hardware performance has doubled annually. Extrapolation of this tendency indicates that hardware capable of full quality A-Buffer rendering of *current scenes* could be available around 2005.

The first steps towards real-time surface shaders are right now being taken by Microsoft with their DirectX8.0 specification. Peercy *et al.* [83] have made an interesting attempt to use OpenGL as

a SIMD machine and have managed to create a low-level language that models RenderMan shading language using OpenGL as its primitive assembly language. It seems obvious that real-time graphics industry will follow the footsteps of the movie industry. Maybe in ten years we will *play* Dragonheart.

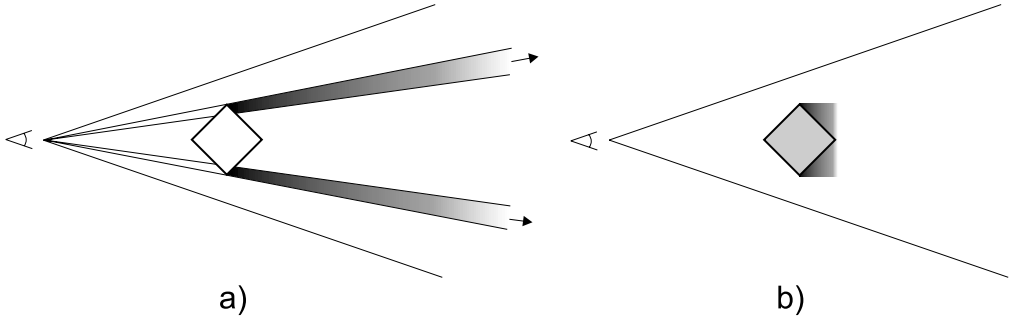

Figure 2.9 Sources of occlusion volume loss include image-space quantization (a) and depth quantization (b).

2.4 Conservative Visibility Determination

Exact visibility algorithms guarantee the correct output as long as all contributing geometry is processed. There might be a large amount of redundant geometry inside the view frustum. Various conservative algorithms have been proposed to avoid processing the hidden geometry. There is currently no standard way of estimating how conservative the algorithm is. Consequently we propose to use the following metric

$$conservativity = \frac{R}{V} \tag{2.1}$$

where R is the amount of visible objects reported by a conservative visibility algorithm and V is the amount of actually visible objects. V can be robustly calculated using an item buffer (see section 2.3.4). It can be easily seen from the equation that conservativity of 1.0 is the perfect result, and values under 1.0 indicate erroneous non-conservative results.

In this section we first consider conservative image-space algorithms in 2.4.1. Object-space algorithms are then reviewed in 2.4.5.

2.4.1 Image-Space Algorithms

Image-space algorithms are usually highly robust, since no computational geometry is involved. Degenerate primitives, non-manifold meshes, T-junctions, holes or missing primitives are all handled without special treatment. Parametric surfaces and everything that can be rendered and bounded with a volume can be used, at least in theory.

Lost Occlusion Volume

Occlusion culling is usually performed using a somewhat lower precision than the actual model geometry. Image-space algorithms may solve the visibility using a resolution smaller than the output resolution. This conservativity in image-space causes potentially significant loss in the *occlusion volume*[121]. An error in image-space extends to a sweep volume reaching the far plane of the view frustum. With a long view frustum an image-space error of a few pixels can correspond to

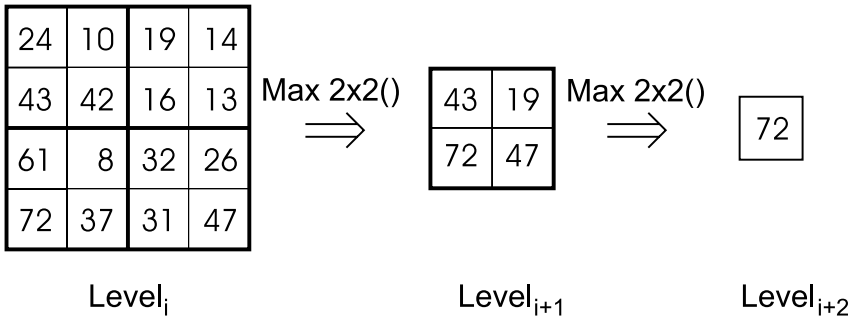

Figure 2.10 Hierarchical Z-buffer principle

a large volume. This is illustrated in figure 2.9 a). Similarly the depth values of an object may be approximated by using a single depth value instead of per-pixel depth values. This also causes loss in occlusion volume. However, being perpendicular to the viewing direction and bounded by the sides of view frustum, the resulting volume is usually smaller than with image-space error (see figure 2.9 b)). We call these losses *image-space occlusion volume loss* and *depth representation occlusion volume loss* respectively.

2.4.2 Hierarchical Z-buffer

Greene and Kass [51] propose extensions to existing Z-buffer hardware to avoid per pixel depth comparisons. Their first major idea is to organize the scene into an octree for approximate front-to-back traversal. For each octree node the question 'Would this node be visible if rendered?' is asked. If not, the entire node is skipped. If only hardware could answer this query, significant performance improvements would be possible. But current hardware does not and according to hardware vendors a fast feedback is against the one-way design principles of current hardware. This implies that the query would have to be implemented either in software or with a separate occlusion culling unit, which should in fact be possible right now. Georges [44] describes an implementation for Pixel-Planes 5 [42], a parallel graphics computer.

Another invention is the hierarchical Z-buffer itself, which is similar to the mipmap-pyramid [22] [115] used in texture mapping, except that whereas a mipmap pyramid is constructed using the average operator, the Z-pyramid uses the *maximum* operator. Figure 2.10 demonstrates the contents of pyramid levels. All writes are performed to the lowest, full resolution pyramid level. A primitive can be quickly determined hidden if its bounding rectangle with the primitive's minimum depth value fails the visibility test. In many cases this can be done with a single comparison on the appropriate Z-pyramid level.

Unfortunately changes to current hardware are required and the efficient implementation of a constantly updating Z-pyramid is problematic in hardware. If the Z-pyramid updates only happened infrequently, the problem would be reduced. Intel has proposed a related concept of *Adaptive Hierarchical Visibility* [119] where Z-pyramid updates are only triggered after a certain amount of primitives have been processed. Their implementation is based on a tile architecture and the pyramid updates get simplified considerably. This causes depth representation occlusion volume loss as primitives introduced between two updates cannot occlude each other.

[22] Mipmap-pyramid has the original image as the lowest pyramid level and each higher level is created from the lower one by averaging 2x2 blocks. Complete mipmap-pyramid of 256x256 texture consists of 9 levels (256x256, 128x128, ... , 1x1).

While the HZ-buffer is not available in consumer-level hardware, it still leaves the possibility of updating the Z-buffer with hardware, downloading it to the system memory and creating the required hierarchy in software. In general, a Z-buffer cannot be read back from the hardware, and in any case, that would be too slow with current bandwidths. Additionally Z-buffer formats are largely hardware-dependent and often unspecified. As a conclusion, we must say that hardware-software implementation is possible with certain hardware configurations, but not in a generic case.

Advantages

1. Provides occluder fusion.
2. Logarithmic search for visible primitives. Only visible or nearly visible primitives are processed.
3. Hierarchical Z-buffer is able to exploit both object-space and image-space coherence and also benefit from previous frames. This makes the HZ-buffer belong to temporal coherence class 2 (see section 2.2).
4. No restrictions on scene structure or primitive types.
5. Should hardware of required kind become available, there might not be need for further development of occlusion culling algorithms.[23]

Drawbacks

1. Neither the depth query nor the Z-pyramid are implemented in current hardware.
2. Octree is hardly the best spatial structure, since most scenes are far from box shape and a waste of resources follows. BSP or kd-tree subdivisions adapt much better into many common scene types.

2.4.3 Hierarchical Polygon Tiling with Coverage Masks

In order to support high-quality antialiasing Greene [50] presented a hierarchical polygon tiling algorithm using coverage masks.

If strict front-to-back traversal is guaranteed, there is no need for depth tests. Greene achieved this, undeniably limiting, goal by organizing the scene into an octree of BSP-trees. The hierarchical culling of octree nodes, similar to the Hierarchical Z-buffer, can be used and each BSP-tree can be efficiently traversed in strict front-to-back order by employing the algorithm by Naylor [79].

If the projection area of an octree node is fully covered, the node is guaranteed to be hidden and all geometry inside it can be culled. Otherwise the primitives are hierarchically tiled into a mask pyramid. Masks used in the hierarchy are special tri-valued masks called *triage masks* (see Fig. 2.11). Pixel value V indicates that all pixels downwards in the hierarchy, starting from this one, are vacant. Value O indicates they are all occupied, whereas P indicates that only part of them are occupied. Hierarchical tiling only writes to fully covered pixels with value V. When tiling encounters value O, it terminates. Value P causes recursive subdivision to the next lower pyramid level. This

[23] It is an interesting question whether hierarchical Z-buffer could become the *de facto* standard for conservative visibility determination as Z-buffer is for exact visibility determination.

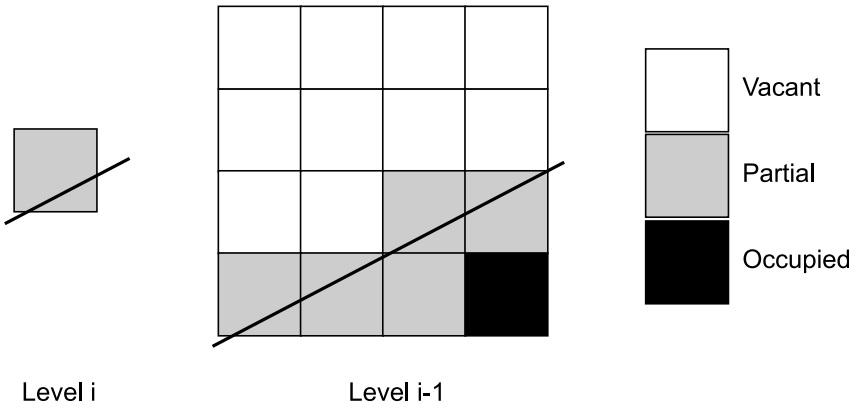

Figure 2.11 Triage masks

writing procedure is similar to Warnock subdivision and manages to avoid much of the tiling work with hierarchical rejection of parts of primitives.

The authors achieve antialiasing by setting the *second* level of the coverage pyramid to the output image resolution. The lowest level is therefore large enough to supply information for 4x4 super-sampling. Due to strict front-to-back traversal, it is possible to calculate the output color by simply accumulating the coverage-weighted samples. As a result, it is possible to get supersampled output quality in a single pass without the need for a 4x4 frame buffer.

Advantages

1. Elegantly bundles visibility determination and antialiasing to significantly reduce memory requirements of supersampling.

2. The algorithm is incremental. This property is inherited from the Z-buffer algorithm.

Drawbacks

1. Only works with fully static environments, since dynamic construction of an octree of BSP-trees is computationally very expensive.

2. Hierarchical tiling should be implemented into hardware to be competitive in scenes with anything less than millions of polygons.

3. BSP-trees of millions of polygons use a considerable amount of memory and in turn limit the usability with large models.

The first drawback is a totally unacceptable requirement and effectively rules out every interactive environment. The second and third are conflicting conditions and imply that while having millions of polygons, the BSP-trees must be kept small. This can only be achieved by making multiple instances of the same BSP-tree. Our conclusion is that while being a very elegant approach, 'Hierarchical Polygon Tiling with Coverage Masks' has mostly theoretical value.

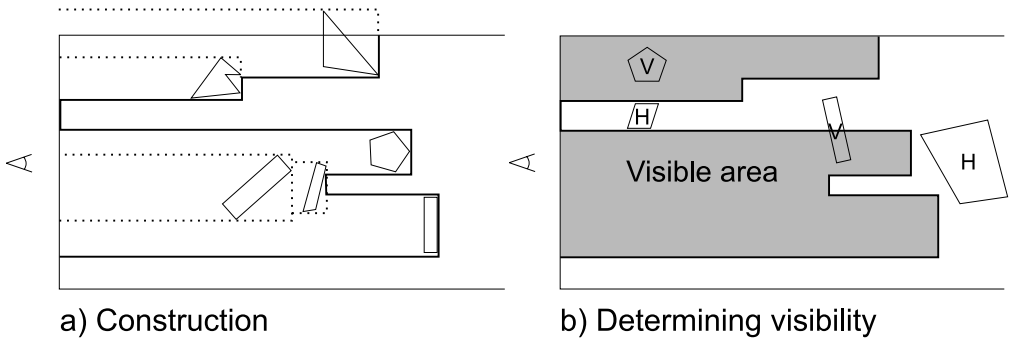

Figure 2.12 Depth Estimation Buffer principle

Figure 2.13 An example of Hierarchical Occlusion Maps

2.4.4 Hierarchical Occlusion Maps

Zhang *et al.* introduce Hierarchical Occlusion Maps (HOM) in [121, 124]. Their primary innovation is to split the visibility test into two sub-tests:

1. *Coverage test*: Is the primitive spatially covered by occluders?
2. *Depth test*: Is the primitive behind the occluders?

New data structures introduced are HOM (figure 2.13) for the coverage test and *depth estimation buffer* (DEB) (figure 2.12) for the depth test. After the visibility test has been split into the sub-tests, resolution of the more involved depth test can be decreased without introducing visible artifacts. Depth representation occlusion volume loss increases when the resolution of the depth test decreases.

Coverage

The Hierarchical Occlusion Map (HOM) is constructed by rendering the opacity values of all occluders into a monochromatic frame buffer. A mipmap pyramid of the frame buffer is built using mipmapping hardware if available. In absence of such hardware, the frame buffer is first downloaded to the system memory and the pyramid is constructed with software. Pixel values in the hierarchy indicate the average opacity percentage of the underlying pixels. Value 0.75 in the topmost layer

indicates that 75% of the highest resolution image pixels are set. Coverage tests of any given rectangle can now be carried out in a hierarchical fashion similar to Warnock subdivision (2.3.6) and *contribution culling* can be enabled by setting the opacity threshold to something less than 100%. It is worth pointing out that if contribution culling is either not required or is done at *construction time*, HOM can be presented with a single bit per pixel and memory requirements are reduced by 87.5%[24]. This is the approach taken by our algorithm (see 3.9 for details).

HOM is conservative if "for each pixel in the output image, its corresponding pixels in HOM have the same or less opacity". The exact condition for HOM being conservative is to store a floating point coverage percentage per pixel. At that point, HOM becomes conservative to the same extent as object-space algorithms [122]. Since authors use a limited base resolution (256x256), high quality antialiasing capabilities are required in the hardware.

Depth

DEB functionality has a strong resemblance with the Z-buffer. Whereas the Z-buffer is initialized to the farthest possible value, DEB is initialized to the *nearest* possible value. A DEB update occurs if the current value is *greater* than the value stored in DEB. It would be possible to implement DEB with current Z-buffer hardware by initializing the buffer value to near, and reversing the depth-test function. Each object is approximated with an axis-aligned rectangle and the farthest depth value of an object is chosen.

No-background Z-buffer[121] is an alternative depth representation scheme. It is constructed exactly like a traditional Z-buffer. During the depth tests *far* values must be interpreted as *near* values.

Occluder Selection

The occluder selection algorithm returns a list of objects to be used as occluders. These occluders are written both into HOM and DEB. All other objects inside the view frustum are tested first against HOM and then against DEB. If both of the tests succeed, the object is hidden.

The basic algorithm is not incremental, and receives no feedback from the occlusion system. In absence of context-dependent information, there is not much one can do. This is similar to a blind guy trying to hide behind objects; in order to succeed, user guidance is required. One can measure the distances to objects and to use the metrics listed in section 2.1 to aid in the selection. Our options are to select every object closer than X meters or to select N closest objects. Size and complexity can merely be used as thresholds, not as proper weights, since there is no cost-benefit function to balance. Only an incremental occluder selection algorithm can be guaranteed to break all lines of sight. Failing to do so can cause catastrophic situations when visualizing very large databases.

Advantages

1. Provides occluder fusion.
2. Depth estimation buffer is fully portable as it can be maintained with software.
3. The proposed implementation is able to handle alpha textures [15], which are often used to model impostors [73].

[24] compared to 8 bits per pixel.

4. Can handle all degeneracies and different primitive types.
5. Is possible to implement with current hardware.

Drawbacks

1. No incremental versions of the algorithm were discussed. [25]
2. Antialiasing is required to assure conservativity.
3. Occluder selection is insufficient.
4. DEB accuracy:
   (a) Using bounding rectangle area to perform DEB writes results in unnecessary worsening of DEB. In [121] Zhang states that these effects are only local and therefore insignificant. While it is true that the effects are local, our empirical experiences indicate that the effects should not be considered insignificant.
   (b) Using a single depth value for DEB writes results in an unacceptable amount of occlusion volume loss.
5. Contribution culling is prone to popping artifacts when an occluder has small holes and an object behind is either culled or not culled depending on discrete samples. Slow smooth camera motion causes discretized hole sizes to vary in pixels. In the worst case this results in totally random culling of slightly visible objects.
6. Reading back the frame buffer from the graphics hardware is very slow, except on high-end graphics computers such as Silicon Graphics Infinite Reality. [26]

Ho and Wang [57] propose replacing the mipmap pyramid with *Summed-Area Tables* (SAT). Even though SAT allows significantly faster, constant time coverage queries, the memory requirements increase from $\frac{4}{3} * 8bpp \rightarrow 32bpp$. Remembering that $1bpp$ representation is sufficient, we observe that SAT uses 32 times as much memory. Authors also propose replacing the depth estimation buffer with a sparse span buffer. This does not have any benefits over the depth estimation buffer. We conclude that the proposed changes do not solve any of the real problems of HOM and only complicate the otherwise elegant algorithm.

Incremental Variants

Klosowski and Silva [65] present an algorithm for fixed-budget rendering. Their *Prioritized-Layered Projection* (PLP) algorithm is basically a database traversal, but can also be used for finding occluders for HOM. Zhang [121] discusses *multi-pass occlusion culling* and concludes that some properties of the current hardware are not favorable for it. Bormann [13] proposes slicing the frustum into several layers. He creates a Hierarchical Occlusion Map for each slice and uses only a single depth value in each slice.

Due to favorable features, we have selected a variant of HOM as our image-space solver. Our contributions are presented in section 3.9.

[25] There is nothing inherent that prevents the algorithm from being incremental. However, when implemented using the current hardware, the update costs are prohibitively high.

[26] The base resolution is generally smaller than the output resolution and therefore the bandwidth requirements are reduced.

2.4.5 Object-Space Algorithms

While image-space algorithms are characterized by robustness and applicability to a wide variety of problems, object-space algorithms usually have no such desirable features. What makes them important, is their theoretical possibility for utilizing temporal coherence, which is missing from every image-space algorithm.

2.4.6 Aspect Graph Variants

The scene can be divided into partitions of constant *aspect*. Any such partition, called Visibility Space Partition (VSP) [90], has the property of seeing a fixed set of visible primitives. Having such information provided, the visible primitives can be found by locating the current VSP. Moving from one VSP to another causes a *visual event*, which implies that the set of visible primitives has changed.

It was shown by Plantinga and Dyer [84] that a VSP with inconvex polyhedra and perspective projection yields the spatial complexity of O(n^9), which is a totally ridiculous requirement. Even worse, some VSP boundaries are quadric surfaces. They proposed a complex structure called *asp* to represent the aspect graph, but apparently did not implement it. Durand *et al.* introduced the *3D Visibility Complex*[31], which obviously was not implemented either. Later they slightly simplified the concept and introduced the *Visibility Skeleton*[32, 33]. They report memory consumption of 400Mb for a 1500 polygon scene and because the complexity is not even close to linear, it is obvious that a scene with millions of polygons would consume memory beyond comprehension. From these results it seems safe to conclude that purely VSP-based approaches can only be used in certain special cases.

The aspect graph of a box consists of 26 VSPs[27]. Since visibility events only occur when the VSP changes, the list of silhouette edges remains constant inside a single VSP [96]. Each VSP has a fixed set of features (faces, edges and vertices), one of which is the closest feature of an object. This information can be used to optimize box vs. plane intersection tests [58]. We first determine the direction of the normal of the plane in the object's space to obtain the current VSP. According to the current VSP we use the pre-computed closest and farthest vertex indices to test the intersection. As every object can be bounded with a box, this method is fully general. [28]

Convex Aspect Graph Variants

Coorg and Teller [26] simplify the aspect graph complexity by limiting the occluders to convex polygons. While this certainly does simplify the problem, it also makes the algorithm quite useless in realistic scenes, where no large polygons are available. The algorithm might also not be suitable for complex or highly dynamic scenes due to the relatively high amount of visual events.

The above algorithm is temporally coherent and might be useful with large and convex *virtual occluders*.

In [27] Coorg and Teller generalize the algorithm to polyhedral occluders. They introduce a new method for estimating the occlusion potential of a polygon:

[27] 6 faces, 12 edges and 8 vertices

[28] We use this method for optimizing our database voxel culling.

$$-\frac{A(\vec{N}\cdot\vec{V})}{d^2} \tag{2.2}$$

Where A is the polygon area, $\vec{N}$ is the polygon normal, $\vec{V}$ denotes the viewing direction and d the distance from the viewpoint to the center of the polygon. How this relates to polyhedral occluders remains unspecified in the paper. One alternative is to estimate the size of the axis-aligned projection of the object. Regardless of the dynamic nature of equation 2.2, the algorithm selects potentially good occluders during a pre-process. If the occluders are selected at a pre-processing stage, one could consider the actual occlusion powers of the candidates, instead of selecting them based on such simple criteria.

The authors also propose an interesting idea of selecting a dedicated occluder for each detail object. How the detail objects are chosen is left as an exercise to the reader and might not be easy to perform automatically. As pointed out by Saona-Vázquez *et al.* [90], the correct amount of detail objects is an important question.

2.4.7 Shadow Volumes

Hudson and Manocha [61] first select all convex objects in a scene. For each one of them the solid angle[29] is measured. If the solid angle exceeds a fixed threshold, the object is accepted as an occluder. At navigation-time they first build shadow frusta [29] from some of the best occluders, and hierarchically cull axis-aligned bounding boxes of the scene hierarchy using the frusta.

The algorithm in its original form works only with convex occluders and can therefore only be used in special cases. While extending it support inconvex occluders is possible, it is unlikely that it will achieve the desired performance level.

Culling with convex shadow frusta is identical to hierarchical view frustum culling and could therefore be easily integrated into many existing occlusion systems.

2.4.8 Portals

Airey [3, 2] proposed subdividing the scene into *cells* connected by *portals*. This subdivision is particularly well-suited for architectural scenes. Most intuitively this is explained with cells corresponding to rooms, and portals corresponding to doors and windows, through which other rooms can be seen.

Concept

Usually we are taking considerable effort to prove that we cannot see through some part of the scene. Cells are implicitly assumed fully blocking, and are therefore a very powerful way of expressing *a priori* visibility information. The shapes of the cells are not restricted, although most practical purposes require convex cells. The places where this blocking property is untrue are places to insert portals. Automatic portalization of generic scenes is a hard problem and no solution has been proposed. Teller [107] presented an algorithm for performing automatic portalization of axis-aligned

[29] same as in [27].

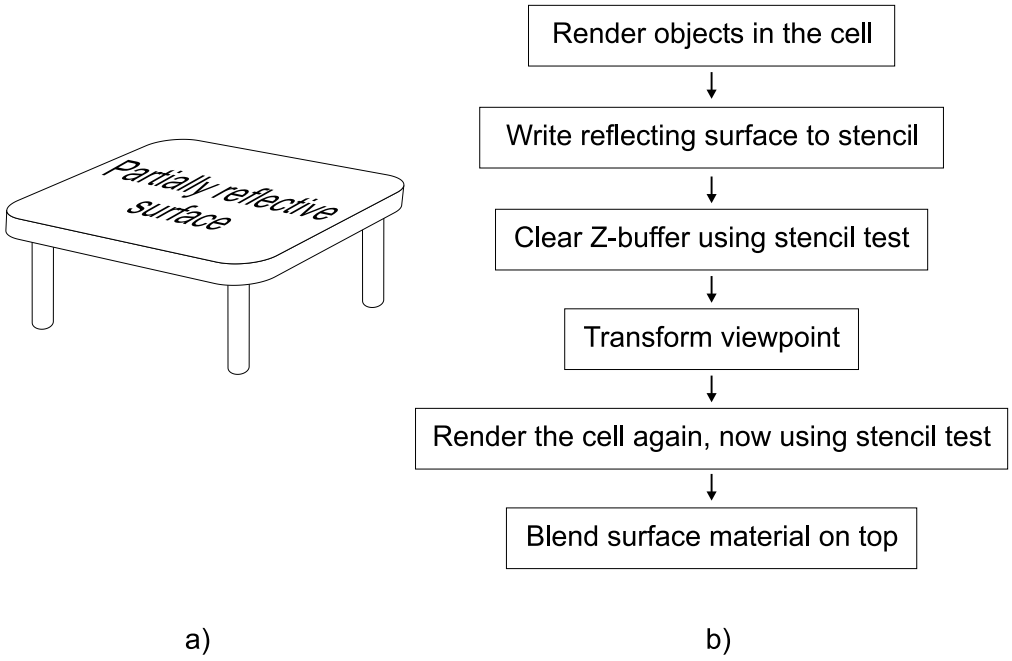

Figure 2.14 Portals can be used for creating partially reflective surfaces, such as the surface of the table. The process is outlined in b).

scenes. Even though this is often the case with architectural scenes, no such assumptions should be made.

Static Version

In portalized scenes the visibility problem is reduced to resolving cell-to-cell visibilities. Both Airey and Teller [108] solved this by creating PVSs in a pre-processing step. To create PVSs Airey proposed sampling from several locations and using shadow frusta, neither of which can guarantee a correct result. Teller gave an analytical solution by formulating the problem into a linear programming problem.

Dynamic Version

Luebke and Georges [72] observe that the current view frustum can be clipped against the portal when traversing through it. If we clip the frustum against the axis-aligned projection of the portal instead of the actual geometry, we always get a frustum with four screen-space axis-aligned side planes. Thus the portal traversal and cell-to-cell visibility determination can be merged to view frustum culling and performed dynamically. The need for PVSs disappears and it is possible to open, close, move and even to create new portals at run-time. Furthermore, the authors observed that adding a transformation matrix to portals made it possible to create reflecting surfaces (figure 2.14) and non-physical continuities in space.

We call portals that do not define a transformation matrix *physical portals* and the ones that do *virtual portals*.

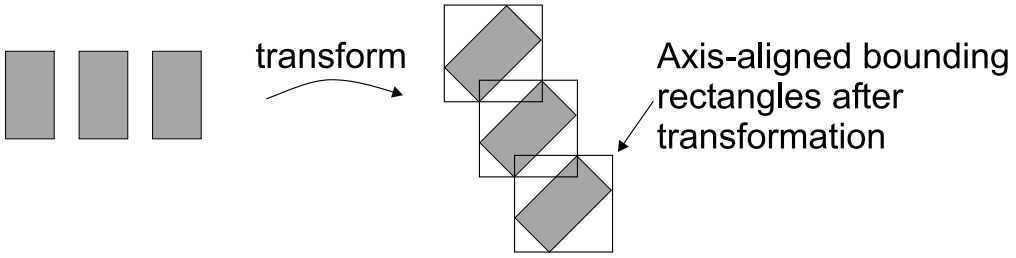

Figure 2.15 Portals are often approximated using axis-aligned bounding rectangles. The bounding rectangles may overlap after viewport transformation and therefore the rendered result is incorrect.

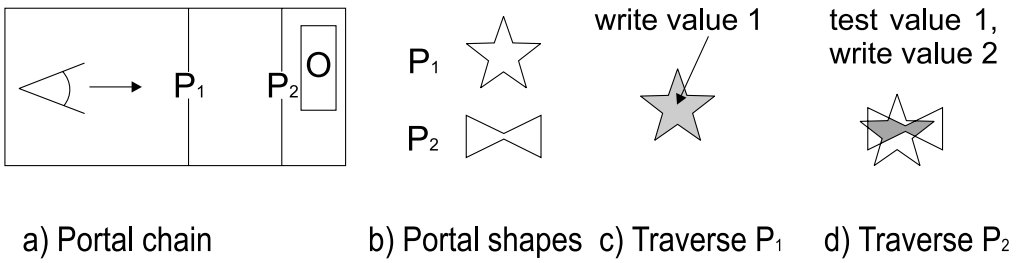

Figure 2.16 Stencil buffers can be used for solving arbitrary portal clipping. Object O is seen through two consecutive portals. Figures c) and d) illustrate the stencil buffer during the traversal and finally O is rendered using stencil test.

Portals can be seen as a way of compactly expressing a PVS. An interesting observation is that portal traversal can be seen as beam tracing (see section 2.3.8). Reflections and approximate refractions of beam tracing are directly applicable to portal traversal.

Overlap Problem

There is no literature giving solutions for certain common problems arising from using portals. Sometimes axis-aligned projections of two portals overlap on the screen. This causes problems when a partially transparent object is seen twice and parts of it are in fact rendered twice. Timestamping the objects is not a generic solution, since there can be any amount of portals between two cells. Merging these two portals into a single bigger portal may fix the problem, but can potentially trigger a chain reaction by overlapping a third portal. This is illustrated in figure 2.15. The only practical solution is to use a stencil buffer [82] to block writes outside the actual portal geometry as illustrated in 2.16. If a stencil buffer is not available, the alpha channel and a destination alpha test can be used instead.

Special Effects

Surprisingly complex visibility relations, such as portable mirrors and reflecting cubes can be modeled with non-physical portals and stencil masks. By adding an off-screen buffer to the list of hard-

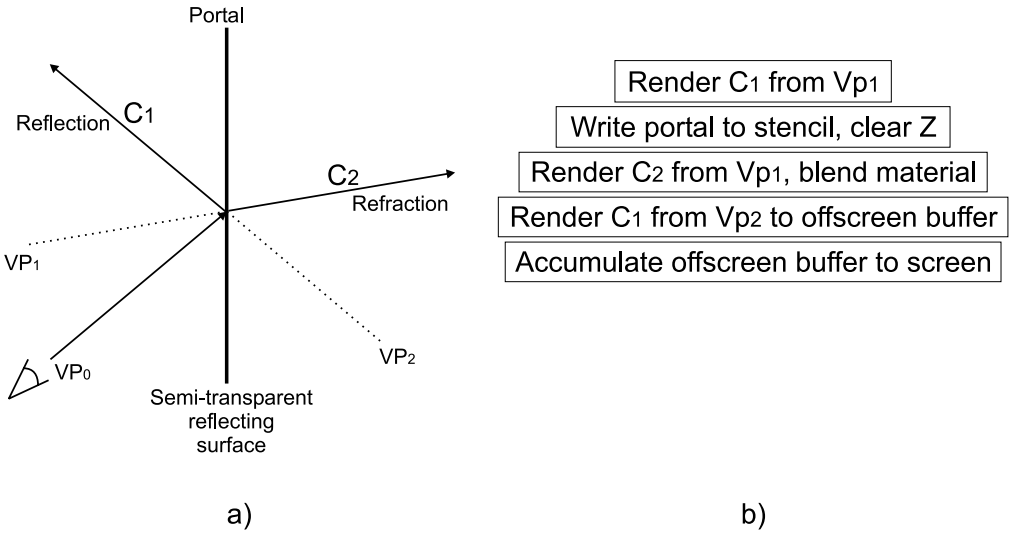

Figure 2.17 Creating simultaneous reflections and refractions using portals

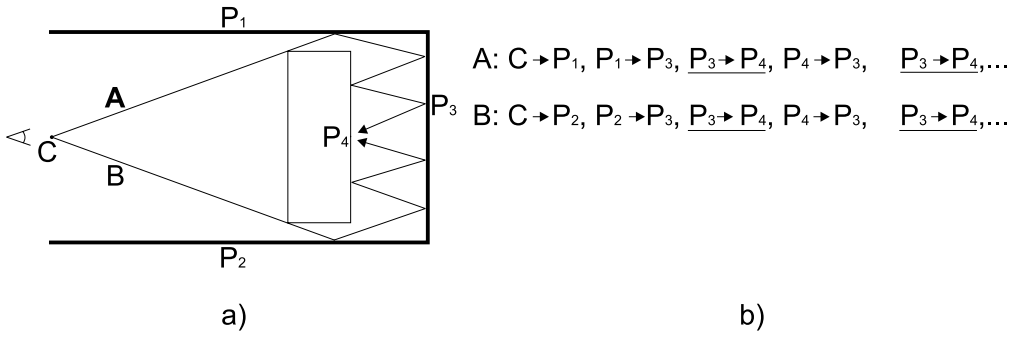

Figure 2.18 Limiting the recursion depth of portal traversal. Assuming the desired recursion depth is 1, the traversal is terminated when the transition $P_3 \rightarrow P_4$ occurs for the second time.

ware requirements, both reflections and refractions[30] can be modeled simultaneously (figure 2.17). The need for an off-screen buffer results from both reflection and refraction requiring their own Z-buffering, while mapping into the same screen pixels.

Infinite Recursion Problem

When using non-physical portals, it is possible to create infinite loops by having mirrors that see each other in a circular fashion. Also, if free transformations are allowed for portals, loops inside the scene are possible. Consider a scene where a character looks into an alien device that transforms the view frustum 5 meters backwards. The character would see himself watching the device, and an infinite recursion follows.

The infinite recursion problem can be solved by writing a recursion solver, which sets the maximum

[30] Refraction calculations are performed on per-portal base only, as in beam tracing.

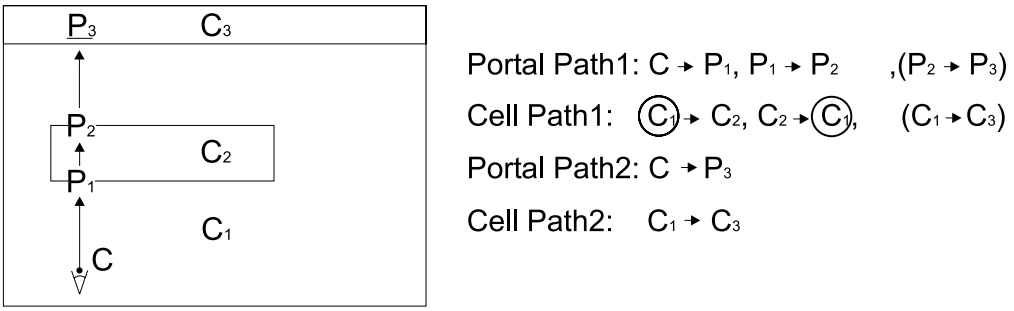

Figure 2.19 The traversal can process the same cell multiple times if nested cells are used. To prevent this, the traversal of "cell path 1" must be terminated since the path already contains C_1.

number of times a portal can be traversed through a *per-portal sequence*. It is important to notice that simply setting the maximum number is insufficient. Recursion solving is illustrated in 2.18. Additionally portals must be back-face culled and two-sided portals evidently result in an infinite recursion[31] in some cases. We made the observation that the portals do not have to be planar. The only condition is the possibility of back-face culling them. Indeed, a cube can be used as a portal. [32]

Multiple Traversals Problem

Nested cells can result in situations where the outer cells are processed several times and therefore the objects inside the cell are also processed several times (see Fig. 2.19).

If the target cell has already been processed during the *portal sequence*, and all the portals were physical, the traversal should be terminated. If any of the portals are non-physical, we cannot guarantee that the transformations of the already processed objects are the same as previously. Thus the traversal cannot be terminated if non-physical portals are involved.

Floating Portals Problem

Special treatment is required if the portal is floating in the middle of the cell. The Z-buffer values may have been set closer than the objects in the destination cell and must therefore be cleared. The proper solution is to clear the Z-buffer if both the stencil test and depth test the for the portal pass. [33]

Warning

There are strong objections against using non-physical portals with free transformation. Physics systems potentially lose the concept of 'up' and the scales get twisted. How should a character passing through a magnifying portal be visualized? To avoid these and countless even trickier problems, the usage of free transformations should not be generally encouraged, except for pure "demo effects".

[31] Which would be solved by the solver explained. Yet, it would result in processing the cells multiple times and possibly incorrect output.

[32] This allows surrounding a floating cell, for example a space station, with a single cube-shaped portal.

[33] This test needs to be done only for the area covered by the floating portal.

2.4.9 Hierarchical View Frustum Culling

View frustum culling is the first operation of a visibility pipeline. Because its input set is usually the whole scene database, the principle of output-sensitivity is of major importance. Naïve view frustum culling tests every object against the view frustum, thus breaking the output-sensitivity. View frustum culling can be done hierarchically [22]. Objects are first combined into bounding volume hierarchies. View frustum tests are then performed starting from the topmost level, and the results are propagated downwards. If a higher-level bounding volume fails the view frustum test, all the objects inside the bounding volume would fail as well, and can therefore be rejected without further tests. This is an important optimization for large scenes and for portalized scenes, where the view frustum is most of the time very narrow. Hierarchical view frustum culling can be seen as a logarithmic search of objects falling inside the view frustum.

View frustum culling can be made temporally coherent by using the algorithm presented in [100] or the one outlined in 3.14.

2.4.10 Hierarchical Back-face Culling

Removal of back-facing primitives can be done hierarchically [69, 123, 63]. This reduces the time complexity from a linear into a logarithmic search of visible primitives. Hierarchical back-face culling of triangles is not widely used in actual rendering, but the same techniques are usable for silhouette extraction.

Kumar and Manocha [68] present hierarchical *backpatch culling* for visibility culling of spline models. El-Sana *et al.* [34] introduce *skip strips* for hierarchical processing of triangle strips.

2.5 Non-Conservative Visibility Determination

Algorithms in this category produce artifacts by removing objects or primitives that should contribute to the output image. This means that the output image is not correct anymore, but in some cases this is considered acceptable. Most importantly, if the camera is moving rapidly, errors of a few pixels will not be noticeable. This principle is widely used in animation compression.

Both conservative visibility determination and non-conservative methods are required for visualization of arbitrarily complex scenes. The most common non-conservative methods include contribution culling (2.5.1), model simplification (2.5.2) and image-based rendering (2.5.3).

Non-conservative methods are not the main focus of this thesis and we mainly list some of the most relevant ideas with references. Nevertheless, we consider non-conservative methods as an important future extension to our library architecture (presented in chapter 4).

2.5.1 Contribution Culling

If an object occupies only a few pixels in the output image, we are tempted to cull it away. The easiest way to achieve this is to estimate the size of the object's bounding rectangle. This is called *screen-size culling* [76]. Zhang [121, 124] offered a more sophisticated method called *contribution culling*, which is able to cull objects according to the percentage of visibility [34]. This method is however more prone to disturbing flickering when the camera moves smoothly. Setting the resolution of image-space algorithms lower than the output image resolution causes *subsampling* and potentially random rejection of slightly visible objects.

2.5.2 Multi-Resolution Presentation

Drawing a lot of detail into a small screen area produces aliasing. Aliasing of distant textures has been successfully solved using an image pyramid[105]-based technique called *mipmapping*[115], which both reduces aliasing and speeds up the rendering. Interpolation between mipmap levels provides additional improvements to the output quality.

It sounds plausible that a similar approach could be used with objects. The object equivalent of a mipmap pyramid is a list of objects with a progressively decreasing amount of detail. Morphing between the levels can be used to reduce visible artifacts. Hoppe [59] introduced the concept of a *Progressive Mesh* (PM) [35], which was later generalized by Popovic and Hoppe [86] to handle non-regular, non-manifold objects.

PM stores the entire process of simplification starting from the full resolution object. A presentation with N vertices can be constructed in linear time. Temporally coherent tracking can be utilized to reduce restoration time considerably. This algorithm is not well-suited for parallel processing and hardware implementation is challenging. PM is extremely well-suited for the progressive transmission of geometry over slow network connections, such as the Internet.

Vast research resources have been put into mesh simplification algorithms. Most of these ignore some or all surface attributes and are only able to handle connected, manifold geometry. This is hardly ever acceptable, and to be of practical use, the algorithm must handle non-manifold geometry

[34] Authors use the term *Levels of Visibility*
[35] Also referred to as continuous level-of-detail

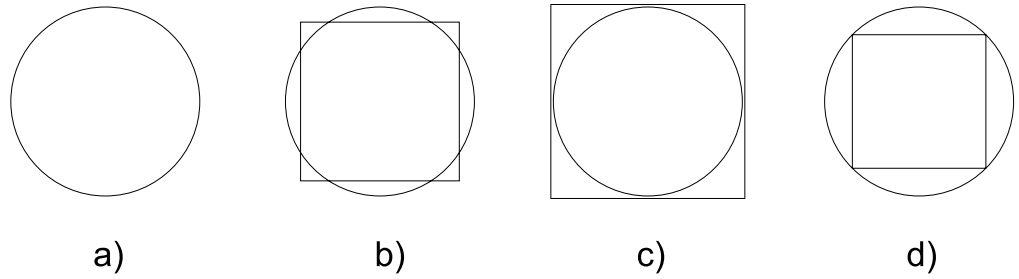

Figure 2.20 Occlusion-Preserving Simplification principle. For visualization purposes a distant sphere could be approximated using an area-preserving square (b). For occlusion culling purposes the approximation must fully contain the sphere when used for testing (c) and be fully inside the sphere when used for writing (d).

and all surface attributes. Proper handling of surface attributes was proposed by Garland and Heckbert [43]. Ericsson's and Manocha's GAPS[35] simplified the work of Garland and Heckbert by introducing *attribute clouds* for robust attribute handling, and also presented an algorithm capable of simplifying unconnected, non-manifold objects.

View-dependent simplification is based on the psycho-visual observation that the silhouette of an object is more important than its internals [66]. This is especially true for models being used in occlusion culling. Hoppe [60] proposed a temporally coherent view-dependent reconstruction algorithm for progressive meshes. Luebke and Ericsson [71] introduced Hierarchical Dynamic Simplification, which tracks view frustum changes and re-tessellates the polygonal environment accordingly. They were also able to exploit temporal coherence.

Occlusion-Preserving Simplification (OPS) Occlusion writes and tests could and should be performed using simplified geometry. The simplified geometry required for this purpose has the additional requirements of being entirely inside or outside the original model. Figure 2.20 illustrates the situation. A simplified test model must fully contain the original model (c) to assure conservative results. Similarly the simplified write model must be fully contained by the original model (d) in order to not cause false occlusion. While the simplified test model can be trivially constructed[36], constructing the simplified write models is a hard problem. Not a single general purpose algorithm solving the OPS for write models has been published. Cohen *et al.* introduced *Simplification Envelopes*[23] for maximum distance bounded simplification. The authors failed to note that setting either the inner or the outer envelope distance to zero indeed gives an occlusion-preserving result. This approach was later used by Zhang[121] to construct simplified geometry for HOM(section 2.4.4). Their simplification algorithm relied on connected, manifold geometry and is inherently hard to implement. Replacing this simplification algorithm with the more tolerant GAPS [35] improves the results, but the hard problem of extracting an envelope of unconnected, non-manifold geometry remains. Hoppe *et al.* [54] also describe an OPS algorithm for manifold objects.

[36] Any bounding volume can be used.

2.5.3 Image-Based Rendering

The process of rendering a frame can be accelerated by reusing the image data from previous frames, instead of rendering the frame from its geometric description. The basic idea is to render one or more objects into a texture, which is then used to represent the objects as long as specific error metrics remain satisfied. *Image warping* is thoroughly covered in [116].

Impostors

Maciel and Shirley [73] assign several bitmaps into each face of every octree node during a pre-process. The authors use image processing techniques to estimate the accuracy of the generated bitmaps when seen from different viewpoints. The results are stored into a table, and at run-time the best-matching bitmap is chosen. The authors define an impostor to be an entity that is faster to draw than the true object, but retains the important visual characteristics of the true object. As all of the impostors are generated during a pre-process, the memory requirements are immense, and the scene is restricted to be fully static.

Schaufler [91] generates impostors dynamically per object. An impostor is always a transparent polygon with an opaque image of the object. The author argues that generating an impostor is hardly more costly than rendering the object into the frame buffer. Rendering into textures was not widely supported at that time, and the argument was at least partially untrue. However technology has advanced, and currently there is quite a bit of truth in this observation.

Aliaga [5] subdivides the scene uniformly into voxels as a pre-process. During navigation, every voxel further than a user-provided threshold is replaced by an impostor. Aliaga and Lastra [6] assign several textures to portals in a pre-processing phase and during navigation they only render the current room and appropriate portal impostors. Sillion *et al.* [99] present a specialized impostor placement algorithm for navigating in a city.

Talisman hardware architecture [109] internally renders objects into several layers and, whenever appropriate, performs affine transformations to layer data to generate subsequent frames. Not much has been said about the error metrics used to determine the needs for re-rendering the geometry. The host is responsible for assigning objects into the image layers and for maintaining a priority list of perceptible errors in the layers. The final output image is generated by a fast layer compositor, which collapses the layers from back-to-front with proper handling of transparencies.

Since the impostors are defined with a single polygon, the depth information is largely lost. This leads to visual artifacts when objects that intersect each other are not rendered correctly. Additionally, the parallax effect inside a single object cannot be simulated.

Layered Impostors

Nailboards [92] additionally store the per-pixel depth information of an object into impostor bitmaps. At the generation time, the depth values are read from the depth buffer, and when rendering the frame, the values are used to offset the depth values. With a correct transformation for the depth offset, nailboards can be mixed properly with geometric rendering. Nailboards cannot be used with current consumer-level hardware.

Layered impostors [93] replace an object with several bitmaps instead of a single one. These bitmaps represent different depths. Since the parallax effect can be approximated, layered impostors can be

used much longer than single-layer impostors.

Spatial Impostors

Shade *et al.* [98] assign dynamically generated impostors to nodes of a spatial subdivision instead of assigning an impostor per object. Thus the impostors in the spatial hierarchy represent regions of space instead of objects. If no geometry intersects the node faces, the assigned impostors can be properly sorted, and occlusion is correctly solved. Schaufler [95] describes a similar approach.

3D Image Warping

To overcome the lack of parallax effect, out-of-plane displacement can be assigned to each impostor pixel. The idea is completely different from nailboards, where the depth was only used to offset the depth buffer. 3D warping equations [116] are unfortunately not invertible like conventional texture mapping. This means that it is impossible to analytically calculate the texture coordinates from the screen coordinates. As the mapping has to be done from texture coordinates to screen coordinates, holes are unavoidable and have to be filled by *ad hoc* methods. The WarpEngine [85] is a hardware architecture solely based on 3D image warping.

Oliveira *et al.* [81] have factored the 3D image warping equations into a 1D pre-warp followed by the conventional texture mapping. It should be possible to implement their *relief texture mapping* in hardware.

2.6 Data Organization and Traversal

Scene Graph vs Spatial Database

Traditionally applications use scene graphs to describe transformation hierarchies and optionally to limit the effects of lights, shaders etc. The scene graph does not contain proximity information, which makes it impossible to efficiently find objects located spatially close to each other. A *spatial database* (or *spatial index*) organizes the data spatially and is therefore suited for proximity queries and can generally provide approximate front-to-back traversal of the scene data. Any performance-optimized implementation of a spatial database is hierarchical.

Data Organization

Axis-aligned nodes of a hierarchical presentation are called *voxels* regardless of the particular data structure in use. Objects can usually belong to several voxels to avoid expensive clipping operations.

Bounding Volume Hierarchy

Rubin and Whitted [87] used bounding volume hierarchies to reduce intersection calculations in ray tracing. If the scene has a high amount of dynamic objects, the maintenance of the bounding volume hierarchy is a challenging task.

Regular Grids

Regular grids are perhaps the simplest spatial subdivision scheme. The drawback of regular grids is that dense and sparse areas of the scene are presented with identical subdivision. The consequence is that voxels in densely populated areas contain a high amount of objects while the ones in sparse areas are nearly empty. As a result, a regular grid is efficient only if the scene contents are uniformly distributed. For example, many terrains are like this.

Octree

Glassner [45] introduces the use of a hierarchical subdivision of a box called *octree*. Each voxel is divided into eight sub-voxels. Finer subdivision is performed in densely populated areas and consequently the amount of objects inside a voxel is nearly constant. The scene traversal is more expensive than with regular grids. The recursive subdivision procedure itself is independent of the data, and causes the octree to adapt rather slowly to any irregular scene structure. Since the octree is a subdivision of a box, it adapts rather poorly to flat scenes; for example, most urban scenes are relatively flat. Some volumetric visualization systems use octrees as the scene description itself [75].

Recursive Grids

Jevans and Wyvill [62] extend the idea of octrees by introducing a more generic subdivision procedure. Whereas octree subdivision always generates eight child voxels, the recursive grid can generate

any amount of child voxels. Usually this amount is higher to reduce the depth of the hierarchy. The optimal number of sub-voxels created is approximately the same as the number of objects in the parent voxel [17]. If even subdivision is used, each axis would be subdivided $\sqrt[3]{N}$ times.

Hierarchical Uniform Grids

Cazals *et al.* [17][18] propose a new approach to spatial subdivision. They first cluster together the objects that are spatially close to each other and have similar sizes. A separate grid is used for each cluster and finally the grids are inserted into higher-level grids. Hierarchical Uniform Grids adapt to an irregular scene structure much quicker than octrees.

BSP

BSP tree [41] is fundamentally different from other spatial subdivision schemes. This is due primarily because it is not axis-aligned and the objects have to be splitted according to the partitioning planes. Splitting creates a considerable amount of *subpolygons* and makes BSP trees suitable mostly for static scenes. Timestamps can be used in many applications to avoid physical splitting of polygons.

Chen and Wang propose the *Feudal priority* algorithm [19] to reduce the amount of splits compared to BSP.

Axis-aligned BSP

If the objects are not clipped according to the planes, and plane orientation is limited to being axis-aligned, also dynamic scenes can be expressed easily. Axis-aligned BSP subdivision adapts to an irregular scene structure faster than octrees. We have selected axis-aligned BSP as our spatial database subdivision scheme.

Chapter 3

Contributions

In this chapter we introduce the new contributions and analyze them in detail. First, we state some of the basic concepts. A preliminary taxonomy of conservative visibility algorithms is given in 3.2. We review the contributions by following the flow of figure 3.1.

Occluder selection is reviewed in 3.3, and a new incremental algorithm with feedback is analyzed in detail in 3.3.2. Sections 3.4 and 3.7 introduce the Visible Point Tracking and Occlusion Write Postpone Queue algorithms respectively. Section 3.8 considers using silhouettes as rendering primitives. The Hierarchical Depth Estimation Buffer (HDEB) is introduced in 3.11. An incremental variant of Hierarchical Occlusion Maps is presented in 3.9. The use of dynamic virtual occluders is outlined in 3.13. Dynamic portal PVSs are briefly considered in 3.14. Finally, conclusions are presented in 3.15.

The camera model used in this chapter is the "standard" view frustum consisting of six planes unless stated otherwise.[1] For example, a portal chain can result in a view frustum with more (or less) than six planes. In such cases, the image-space algorithms use a scissor rectangle and the object-space algorithms use the frustum planes.

3.1 Building a Framework of Inter-Connected Algorithms

Bidirectional Visibility

In the real world, it is always true that if point $\mathbf{B}$ is directly visible from point $\mathbf{A}$ the opposite holds as well. Unfortunately in computer graphics it is possible to create objects that violate this fundamental property. One-sided primitives are commonly used to allow back-face culling. We reserve the right to assume that the above relationship always holds. By doing this, we acquire the right to reverse the visibility calculations whenever appropriate. To cope with violating objects we offer a possibility to manually disable the use of these objects as occluders. Furthermore, the objects that penetrate the viewport may result in situations where the interior of an opaque object is seen. Such cases can usually be considered a bug in the application code, but to avoid ambiguities we do not use such objects as occluders.

[1] left, right, top, bottom, near and far.

Duality of Visibility

We believe that no single visibility algorithm will ever solve all application types satisfactorily. Therefore we propose to use a framework of inter-connected algorithms. Having multiple algorithms working together, a fine-grained subdivision of the problem domain is desired. The first, admittedly obvious, observation is that an object's visibility can be determined in two passes.

1. Prove that the object is visible.
2. Prove that the object is hidden.

This division may seem more than obvious, but it implicitly possesses an important property. An object is visible if *any* part of it is visible. On the contrary, the object is hidden only if *all* parts are hidden. The first test is an early-exit test and we thus first perform a series of tests trying to prove that the object is in fact visible. If all of these tests fail, we are almost sure that the object is hidden, and dare to spend substantially more time trying to prove this. The process is backed up by history data, so that we use results of the previous tests to make our initial guess.

Adaptive Hierarchical Testing

As proving either one of the above conditions is generally a difficult problem, we always use hierarchies of tests organized in order of increasing computational complexity. Whenever a simple test is able to answer the question, significant savings result by not executing the more involved tests. Whether a more involved test should be executed is context-dependent:

1. How much could be won by executing this test?
2. How much does the test itself cost?
3. Is it likely to succeed?

These are very important questions, and to be able to answer we must have some information on which to base the conclusion on. The full potential of an inter-connected framework can only be exploited by building feedback mechanisms to serve data for adaptation and learning. Due to our goal of building a general purpose library, we desperately need systems adapting to the problems at hand. To be able to answer the question "Is it wise to use this object as an occluder?" we should have some knowledge of the history. It is not difficult to construct scenes where the optimal amount of occluders ranges from zero to thousands. As user intervention or parameter tweaking is clearly out of the question, our framework must automatically adapt to the input data. However, rapid adaptation is a double-edged sword, because faster adaptation means having a shorter memory. This duality can be triggered by moving either swiftly or slowly in the same scene.

Resonance

A hierarchy of multiple adaptive algorithms can end up in a situation where algorithm $\mathcal{A}$ performs adjustments that cause $\mathcal{B}$ to adapt, which in turn causes $\mathcal{A}$ to adjust backwards. This is an unstable situation we refer to as *resonance*.

On the positive side, when $\mathcal{B}$ performs poorly, $\mathcal{A}$ will adapt to $\mathcal{B}$'s poor feedback and the effect is partially canceled out. This makes the adapting framework much more tolerant to problem cases than any single algorithm. Unfortunately, this is also true when there are bugs in some particular algorithm in the hierarchy. A sophisticated run-time debugging environment is required to locate such problems, otherwise a malfunctioning part might go unnoticed. This was indeed true with SurRender Umbra. Several bugs were extremely hard to find because other systems "fixed" them.

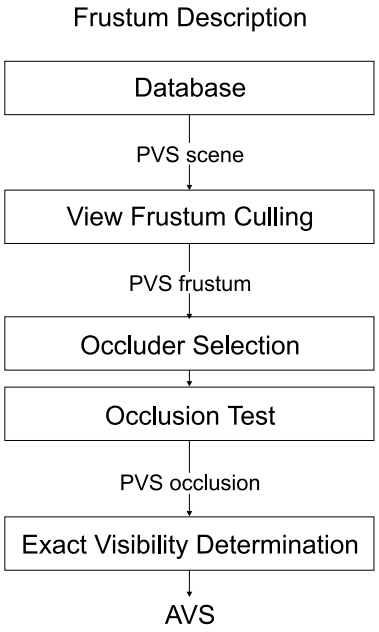

Figure 3.1 Data flow of conservative visibility determination algorithms

3.2 Preliminary Taxonomy of Conservative Visibility Algorithms

The data flow of conservative visibility determination algorithms is shown in figure 3.1. The pipeline begins when a frustum description is received.

Dynamic vs. Static Algorithms

Dynamic algorithms execute the entire pipeline of figure 3.1, whereas static ones do not. The static algorithms can be divided into the following two categories:

1. Skip *occlusion test* and *occluder selection*
2. Skip *occluder selection*

An extensive amount of category 1 algorithms have been published. These include aspect graph variants [84, 31, 32], original PVS implementations [3, 108] and other from-region PVSs. All these algorithms are limited to static environments and are highly memory-intensive. Some authors have proposed efficient compression methods [46, 111].

An alternative is to find only the good occluders in a pre-processing stage [26, 72]. This has the advantage of not requiring substantial amounts of memory. Parts of the scene can usually be considered static, and information amount good occluders can be stored. Dynamic portal (section 2.4.8) variants belong to this category.

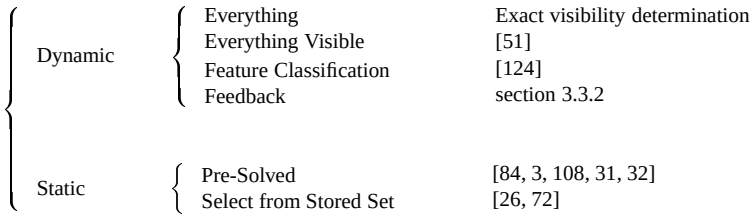

Figure 3.2 Taxonomy of occluder selection

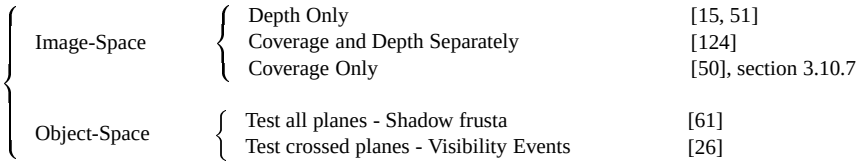

Figure 3.3 Taxonomy of dynamic occlusion culling

The Spatial Database

The spatial database holds the entire scene and the format is usually one of the formats listed in section 2.6. The database can be either fully static or dynamically adapting. Most of the current implementations are limited to static scenes and are therefore not suitable for general purpose use. A notable exception is [102].

View Frustum Culling

Performance-optimized view frustum culling is always done hierarchically [22]. Dynamic Portals, Beam Tracing and Frustum Casting belong to the view frustum culling category. Optimized view frustum culling algorithms can be found in [10, 120]

Occluder Selection

Occluder selection can be either dynamic or static. Figure 3.2 presents a preliminary taxonomy of occluder selection. If every object is used as an occluder, the algorithm is an exact visibility algorithm. The first dynamic approach is not to select occluders, or in other words, to select everything potentially visible as an occluder. Alternatively, only some of the closest and simplest objects can be selected as occluders. Finally, the occluder selection process can rely on feedback information from the previous frames. Static occluder selection either does not select occluders, and thus stores the list of potentially visible objects, or selects good occluders from a pre-processed list, possibly with spatial information.

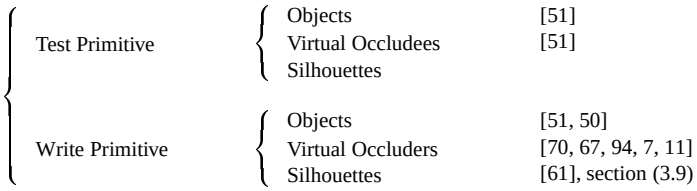

Figure 3.4 Taxonomy of primitives used in occlusion test

Occlusion Test

There are several fundamentally different ways for performing occlusion culling. Figure 3.2 presents a preliminary taxonomy of occlusion culling. The first question is whether the culling takes place in image-space or object-space. Image-space algorithms are limited to a fixed resolution and can be seen analogous to *point sampled* exact visibility algorithms, whereas object-space algorithms are analytical and thus *continuous*. Additionally, object-space algorithms can potentially exploit temporal coherence unlike any currently known image-space algorithm.

Image-Space Occlusion Culling can be carried out in three different ways. Performance-optimized implementations are always hierarchical. Perhaps the most obvious approach is to perform depth tests using the output resolution [15, 51]. An alternative is to perform either the coverage or the depth test at the output resolution and to quantize the other one. Depth representation requires more memory than coverage. Therefore it is a better alternative to quantize the depth as in HOM [124] (see section 2.4.4). If strict front-to-back traversal is available, either coverage or depth alone is a sufficient representation. Since the depth value can be quantized into a single bit, these two tests become the same. Coverage-only representation was used in "Hierarchical Polygon Tiling with Coverage Masks" [50] (see section 2.4.3). The Validation Buffer introduced in section 3.10.7 also works in dynamic environments whenever possible, and uses other algorithms to handle the problematic cases.

Object-Space Occlusion Culling can be carried out in two fundamentally different ways. The first alternative is to construct shadow frusta of the occluders and use those to cull the geometry [61]. As a result, all the occluding planes are tested every frame, and culling with shadow frusta is not temporally coherent. An alternative is to track visibility events when specific planes are crossed [26]. This is temporally coherent, since tests are only performed when visibility events occur.

Primitives Used in Occlusion Culling

Occlusion culling can be implemented using several different primitives in both occlusion writes and tests. Using simplified geometry is theoretically the same as using the object and hence is not listed separately.

The basic approach is to use the object itself as the occlusion write primitive [51, 124]. Alternatively, any hidden object can be used as an occluder. See section 2.1 for a description of virtual occluders. Finally, an object's silhouette can be used instead of the object itself. In object-space this approach has been taken by Hudson *et al.* [61] and in image-space by us (section 3.9).

Occlusion tests are usually performed at a somewhat lower precision than writes. The object itself can be used as a primitive [51, 50]. An alternative is to perform the test with *anything that encloses the object*. We use the term *virtual occludee* for such representations. Most commonly these are the nodes of the spatial hierarchy [51] or the bounding volumes of the objects [124]. In image-space algorithms the virtual occludee is often further conservatively simplified into an axis-aligned rectangle with a single depth value.

Exact Visibility Determination

Any exact visibility determination algorithm can be used. Taxonomy of exact algorithms is show in figure 2.3.2. Almost every currently available graphics hardware uses the Z-buffer [15] (outlined in section 2.3.4).

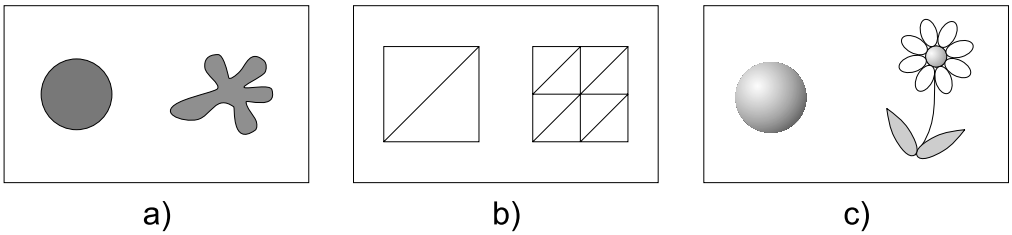

Figure 3.5 Compactness as a tessellation and shape descriptor. The figures on the left are compact whereas the ones on the right are not.

3.3 Occluder Selection

Instead of just selecting *simple* and *large* occluders or relying on *a priori* information, a more analytic approach is needed. The concept of being a good occluder is entirely context-dependent and has little to do with an object's projection size. Before proceeding with a new approach to occluder selection, we briefly explain some of the object characteristics we tried to use.

3.3.1 Potentially Good Occluders

Without context-dependent information we can calculate some descriptors that provide information whether an object *might* be a good occluder.

Visible Object

The most important rule is that a hidden object is generally a bad occluder. [2] An exception to this are Virtual Occluders (see section 2.1).

Normalized Occlusion Property

Every object has a fixed occlusion property regardless of its transformation. When the scales are normalized, a coffee cup is generally a better occluder than a tree. *Compactness* is a transformation-independent shape descriptor widely used by the pattern recognition community [101]. Figure 3.5 illustrates the concept.[3]

$$\frac{edgelength^2}{area} \tag{3.1}$$

$$\frac{area^{3/2}}{volume} \tag{3.2}$$

[2] This information can be found in incremental algorithms.
[3] The shapes on the left are more compact that the ones on the right.

Equations 3.1 and 3.2 represent two and three dimensional compactness descriptors respectively. When used for triangle meshes, 2D compactness estimates the tessellation level. 3D compactness estimates how well the object occupies its bounding sphere. Therefore the global minimum is achieved with a highly tessellated sphere. Calculating an object's volume is a linear operation for closed manifold objects, but a hard and ambiguous problem for objects that are not closed.

Combining both 2D and 3D compactness formulas yields an estimate that tries to balance between low tessellation and a compact shape. According to our psycho-visual tests, the compactness measure reacts expectedly to holes and other problem areas, and sorts objects according to the normalized occlusion property.

Projection Size

From any viewpoint, the objects with a large projection area are likely to be better occluders than the smaller ones. The size of an object's screen projection can be quickly estimated from its bounding volume. Together with the Normalized Occlusion Property, it provides a reasonable estimate of the object's occlusion power. Unfortunately, this measure does not take into account the geometry actually being obstructed by the object. A building is a good occluder, but only if there is something behind it.

Static Context Dependency

In most cases, a small clock on a wall is a redundant occluder. Obtaining such information is hard unless provided by user. Static context-dependencies can be calculated automatically by selecting a large amount of viewpoints, and trying a large amount of occluder combinations to see which ones worked out the best [29]. Such extensive pre-processing is unacceptable for our purposes.

Dynamic Context Dependency

No proposed occluder selection algorithm can properly handle dynamic context-dependencies. Analyzing the results of different occluder sets can be used to tune the occluder selection process [121]. The accuracy obtainable with this approach is fairly limited.

3.3.2 Incremental Occluder Selection with Feedback

As pointed out, a potentially good occluder in the wrong place is a bad occluder. The problem of constructing the optimal occluder set is NP-hard[4] and can only be approximated dynamically. Our goal is to estimate benefits of each selected occluder, and to use the results to predict the future context-dependent occlusion powers dynamically. This is appropriate, since the set of good occluders is usually temporally coherent. The spatial database provides approximate front-to-back traversal and the write queue (see 3.7) postpones the occluder writes up to the point the results are really needed. The resulting effect is mostly similar to the PLP-algorithm [65].

Let $\mathbb{F}$ denote the set of objects inside the view frustum, $\mathbb{O}$ the set of objects used as occluders, O_{base} the base cost of using occlusion culling and R_i, T_i and W_i the costs of rendering, performing an

[4] With N objects there are O($N!$) different occluder sets. Due to *occluder fusion* the occlusion power of an object depends on (possibly all) other objects. Thus finding the optimal set is a combinatorial problem.

occlusion test and using the ith object as an occluder respectively.

The cost of rendering a frame without occlusion culling:

$$\sum_{i \in \mathbb{F}} (R_i) \tag{3.3}$$

The worst-case cost of rendering a frame with occlusion culling:

$$\sum_{i \in \mathbb{F}} (R_i + T_i) + \sum_{j \in \mathbb{O}} (W_j) + O_{base} \tag{3.4}$$

Let $\mathbb{B}$ denote the set of objects culled away so that current object's contribution was necessary and $\mathbb{N}_i$ the number of objects participated in culling the ith object. Since the O_{base} is independent of the occluders selected, we can omit it from our cost-benefit evaluation. Then the total benefit of the current object is:

$$\underbrace{\sum_{i \in \mathbb{B}} (\frac{R_{\mathbb{B}_i} + T_{\mathbb{B}_i}}{\mathbb{N}_i})}_{gain} - \underbrace{W_{object}}_{lose} \tag{3.5}$$

Constructing $\mathbb{B}$ requires the use of a full-resolution Z-buffer and an item buffer, which is an expensive operation and therefore impractical[5]. Let us define $\mathbb{C} \subseteq \mathbb{B}$ as the set of objects culled away, so that the current object contributes to the coverage buffer before the coverage test. Constructing this new set requires neither a Z-buffer nor an item buffer, but the evaluation still has to be performed with full resolution. Finally, we declare $\mathbb{D} \subseteq \mathbb{C}$ as the set, where the previous condition holds but the contribution area is a rectangle instead of a single pixel. It holds that $\lim_{area \to 1} \mathbb{D} = \mathbb{C}$. As an obvious consequence, we declare the size of this rectangle to be equal to the tiles used in our write queue. In general this would be 64x64 or 32x32 pixels. More accurate tiles have been observed to cause increasing overhead, while not providing significantly better results.

We believe that $\mathbb{D}$ is the closest possible approximation to $\mathbb{B}$, so that the time spent in building the set does not become a dominant cost. Construction of $\mathbb{D}$ proposes strict requirements for the occlusion sub-system:

1. Approximate front-to-back traversal.
2. Occlusion writes must be postponed until their contributions are needed. [6]
3. Incremental occlusion buffer.
4. Feedback from the coverage buffer, if the object contributed to the area in question.

[5] Except for testing the algorithm.
[6] This would be unnecessary if strict front-to-back traversal and exact depth writes were available.

The object test cost $T_{\mathbb{B}_i}$ in equation 3.5 is dominated by the rendering cost and cannot be reliably estimated due to hierarchical testing[7]. Omitting the term is analogous to slightly overestimating the rendering cost. So the final benefit for an object is:

$$\underbrace{\sum_{i \in \mathbb{D}} \frac{R_{\mathbb{D}_i}}{\mathbb{N}_i}}_{gain} - \underbrace{W_{object}}_{lose} \tag{3.6}$$

Implementation

We can estimate W_{object} from the silhouette cache's hit ratio, silhouette extraction cost, screen projection area and silhouette complexity. $\mathbb{N}_i$ can be maintained incrementally inside write queue tiles by appending contributing occluders individually to a *benefit receiver list*. The coverage buffer must supply the information about whether the object contributed inside the tile area. $R_{\mathbb{D}_i}$ is a function of an object's geometrical and material complexity. Unfortunately this can only be approximated. However, occluder selection is rather insensitive to errors in calculation of either one of the costs. Both the *gain* and *lose* parts of equation 3.6 tend to get approximated incorrectly in the same direction, and what really matters is that the correct occluders get the benefit.

Whenever an occludee is found to be hidden, its rendering cost is divided among all tiles it touches. For each tile, we divide the assigned benefit with the number of occluders currently in the benefit receiver list and accumulate the value. When the entire frame has been processed, the benefit receiver lists of all tiles are collapsed and the benefits are assigned to the objects. The proposed two-pass approach is of linear complexity unlike the direct assignment, which grows quadratically with the amount of receivers.

Each processed object stores the benefit it received during the previous frame. Additionally, a fixed-size history of N previous benefits is maintained to stabilize the prediction. We call the balance between these two terms *responsivity*. Responsivity is used to indicate how quickly the predictor reacts to changes. Putting more weight to history results in a predictor that is less prone to accidental changes in occlusion power, but reacts slower to new potentially good occluders.

Outdated and Insufficient Information

We are able to get rather reliable information about the true occlusion power of the occluders used. Since data from previous frames is used to guide the selection process, problems arise from old information and from not getting information about objects that have not been used as occluders lately. In such cases, we must somehow try to use the object as an occluder to get updated information. The following cases arise:

1. The object has recent information and estimates are valid.
2. The object has been hidden for N frames, but has become visible during this frame, and its information is therefore outdated.
3. The object has performed poorly during the previous times it was used as an occluder, and has therefore a very low benefit estimate.

[7] Visible Point Tracking (section 3.4) diminishes the cost of occlusion testing considerably.

How should we react to objects becoming visible? Since there is no information about their potential occlusion power, or it is outdated, our best bet is to use them as occluders to get the information updated. However, if the object smoothly slides into the view frustum, it almost certainly receives very little or no benefit during the first frame it is visible. This is clearly unfair. Even if an object has been determined to be inadequate from place A, nothing guarantees that it could not perform extremely well from place B.

Including a random term into the selection procedure gives a possibility to distribute the chance of a new object getting selected to be an occluder over N frames. The expected time before selecting the object can be bounded when the repeated execution of the random function is considered to be a cumulative probability function. The *cost-benefit* function for occluder selection is:

$$\underbrace{\alpha\mathcal{P} + (1-\alpha)\mathcal{A} + \frac{\mathcal{W}}{q}}_{benefit} - \underbrace{\mathcal{W} * rnd()}_{cost} \tag{3.7}$$

The first two terms denote the blend between the previous benefit ($\mathcal{P}$) and the N-frame average benefit ($\mathcal{A}$) using responsivity (α). The third term of the benefit part assures that objects of categories 2 and 3 (above) get a chance to prove their occlusion power during a bounded frame interval. The basic idea is to increase an object's possibility of getting selected along with the time passed since the object was previously visible. An estimate of the write cost ($\mathcal{W}$) is present to scale the values to match the cost side. Scaling value q is selected so that by executing the selection code N times, the cumulative probability of an object getting selected reaches 0.99. To enable this, the cost side of the equation includes $rnd()$, which returns a pseudo-random number in range $[0, 1[$. For example, if $q = 500$, the condition is fulfilled when $N = 30$. In other words, an object has a high probability to get a chance as an occluder during a period of 30 frames regardless of its observed occlusion power.

Let λ denote the number of frames elapsed since the object was last visible. The objects of category 3 usually have $\lambda = 1$. As the objects sliding into the view frustum tend to have limited occlusion power during the first few frames, and as the equation tends to select such objects straight away, we utilize a trick to assure they get another chance in a short while. When an object with $\lambda > 15$ is observed, we overwrite its average history

$$\mathcal{A} = \mathcal{W} \tag{3.8}$$

before executing equation 3.7 in order to give the object an increased chance to prove its abilities during the next few frames.

Global Knowledge

If several occluders are required to hide a single object, the algorithm proposed can fail to select all the required objects simultaneously, and to conclude that they are all bad occluders. We call the operation of using every visible object as an occluder a *flash*. Flashing increases the amount of occluders used every N frames by I percent. Typically we have observed $30 - 100\%$ of visible objects being used as occluders (v). The overall cost of flashing is

$$I = \frac{100\%}{v\%} * \frac{1}{N} \tag{3.9}$$

which is 6.7% more occluders for typical values ($N = 30$ and $v = 50$). This is the cost of fixing the occluder selection information of every visible object and effectively prevents the convergence to a bad occluder set. Flashing increases the processing time of a single frame and the operation cannot be amortized over several frames. We have not been able to witness significant performance degradation in actual scenes during the flash. This is mainly due to lower amount of visible objects reported to the rendering subsystem during the flash. Umbra uses only a limited portion of the CPU time, and consequently the peaks are not that big. One of the reasons why we can use flashing is that we always select only the visible objects as occluders. Therefore the flash operation remains output-sensitive.

Another possibility is to use the flash only when there seems to be a need for it. When the increase in the amount of visible objects between two subsequent frames is more than a fixed threshold, say 30%, it is probably an indication of an abrupt change in the camera motion or the scene geometry. It may or may not be that the occluder selection algorithm has performed poorly. Regardless of the actual reason, we can guess that there was something wrong with the occluder selection, turn on the flash and re-resolve the visibility.[8] Current implementation of SurRender Umbra uses both fixed frame (once every 32 frames) flashing and 'panic' flashing.

Discussion

The occluder selection algorithm proposed finds the occluders that actually hide other objects. It completely ignores previously used heuristics of using 'large' and 'simple' objects as occluders. Instead of such *a priori* classification, our algorithm dynamically learns to use the correct occluders, and we have observed the expected behavior with scenes ranging from a few objects to scenes where the fusion of thousands of occluders is required.

It is theoretically possible for our occluder selection algorithm to conclude that there are not enough good occluders and therefore infinite lines of sight can exist. Such a case can be a big problem when visualizing very large environments. Fortunately it can be detected by utilizing a heuristic that compares the amount of visible objects of the previous and the current frame. If the amount of visible objects increases by more than 40%, it is likely that infinite lines of sight exist and therefore we perform the query again with the flash enabled.

[8] Activating the flash for the *next* frame can cause the application to fetch unnecessary data to the memory.

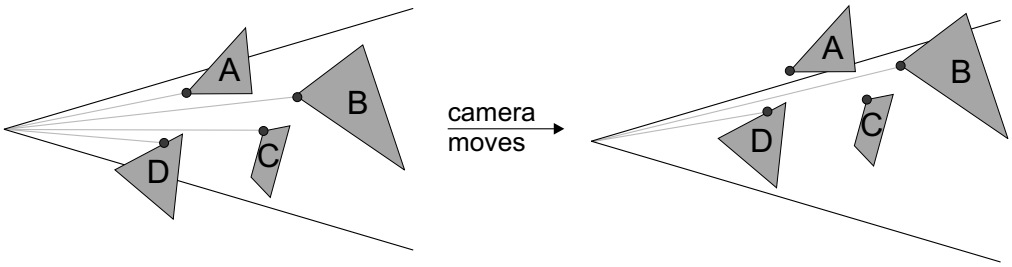

Figure 3.6 Principle of Visible Point Tracking. After the camera has moved, objects B and D can be proved visible with a single ray cast. Visible point candidates of objects A and C are hidden and therefore a more involved visibility test is required.

3.4 Visible Point Tracking

Any visible object can be theoretically proven visible with a single ray cast. If we always knew the correct point to cast the ray into, none of the visible objects would require further testing. Thus the visibility determination problem would be reduced into proving that the *nearly visible* [9] objects are hidden. Being backed up by an image-space algorithm, we can replace the ray cast with a constant time image-space test. If the selected visible point candidate was visible, the object is visible. If the object gets determined visible by another more involved test, we should try to find a better visible point candidate. The problem is solved if the image-space algorithm used can supply coordinates of some actually visible point, otherwise we randomize a new location inside the object.

Figure 3.6 a) illustrates the concept of Visible Point Tracking. In 3.6 b) object A's visible point is no longer visible even though the object is. Therefore a new visible point should be calculated. Visible points expire quicker when the camera motion is rapid. Figure 3.7 shows an example of Visible Point Tracking in action. The camera is located at the origin of the rays. In this case only one of the objects fail the ray cast and therefore requires more involved visibility tests.

The same technique can be used to speed up view frustum culling. We have been able to find on average 80% of the visible objects using Visible Point Tracking. If the camera motion is subtle, the figure steadily approaches 100%. Additionally, we can expect that the objects passing the test really are hidden. Therefore we dare to use significantly more effort to prove this, and even if the tests fail, we get a better estimate for a new visible point candidate, and are possibly able to skip the more expensive tests next frame. While Visible Point Tracking provides temporal coherence into determining that the object is *visible*, it provides no solid information about objects being hidden.

[9] Nearly visible objects are the ones that are hidden, but need to be processed as their parent bounding volume (voxel) cannot be determined as hidden.

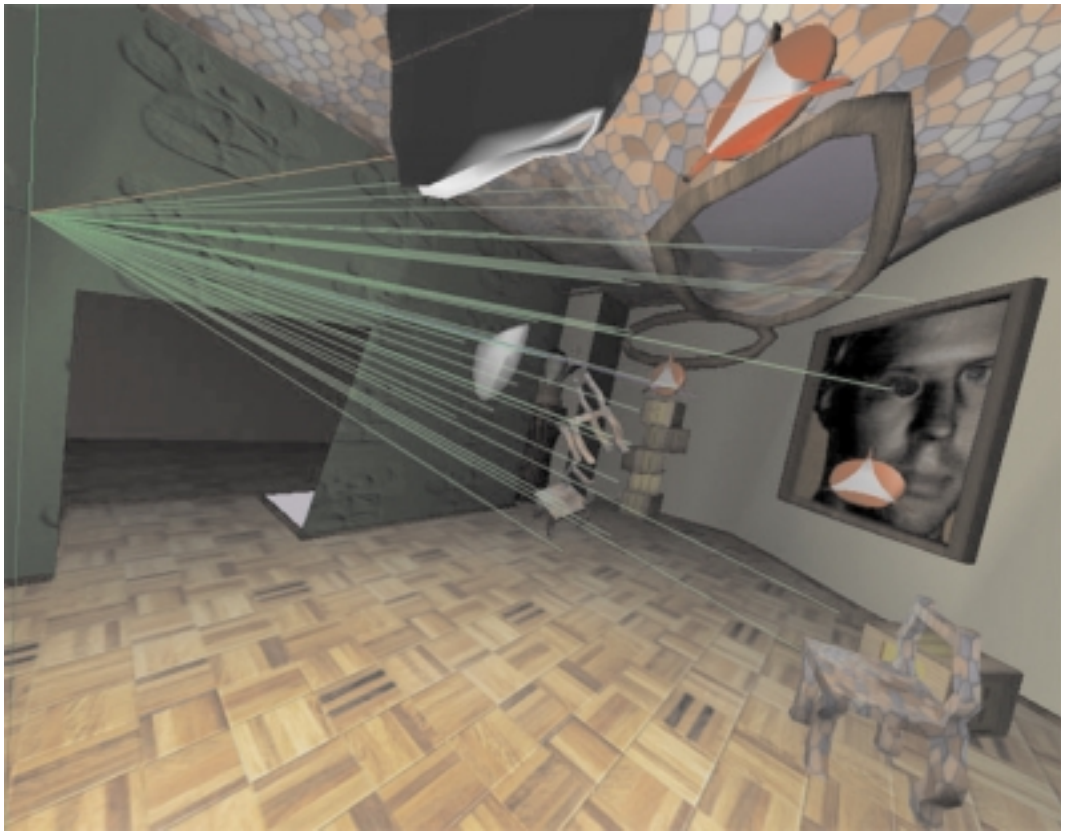

Figure 3.7 Example of Visible Point Tracking. The camera is located at the origin of the rays. In this case only the topmost ray (shown in red) fails to hit its target. That particular object is a *moving* Umbra logo and therefore its visible point candidates expire quicker than the ones used for the static objects.

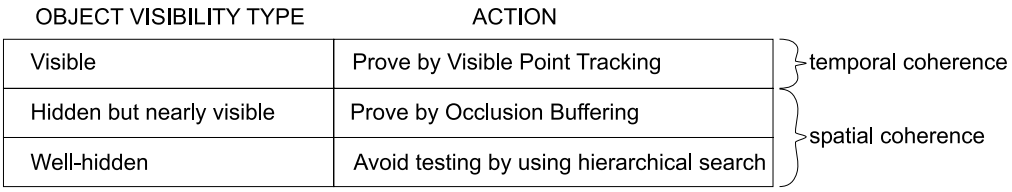

Figure 3.8 Object visibility types and corresponding actions

3.5 Object Visibility Types and Actions

An object's visibility type can be classified into the three categories shown in figure 3.8. The action taken is based on the visibility type. Temporal coherence is exploited by Visible Point Tracking and spatial coherence by occlusion buffering and the hierarchical spatial database search.

3.6 Occlusion Buffer and Plan of Development

Occlusion Buffer is a common term for image-space occlusion culling data structures. The most common occlusion buffer algorithms include hierarchical Z-buffer[51] and hierarchical occlusion maps (HOM)[121]. Non-hierarchical variants, such as 1bpp coverage buffers and span buffers exist, but are not used in speed-optimized implementations.

We present our incremental occlusion buffer in section 3.9.

Coverage Buffer is a part of the occlusion buffer that stores coverage information. In addition to coverage information, an occlusion buffer generally requires some representation of depth. The resolution of the depth representation can be chosen independently of the coverage buffer and is usually smaller in order to obtain lower memory consumption and faster processing speed. The most common representations include Z-buffer and depth estimation buffer (DEB).

We proceed by introducing an occlusion write postpone algorithm (3.7). Silhouettes as occluder primitives are considered in 3.8 and the results are used in section 3.9 for creating an incremental version of HOM. Hierarchical version of DEB is reviewed in section 3.11.

3.7 Occlusion Write Postpone Queue

With *incremental* occlusion buffer algorithms it is possible to delay writes to the occlusion buffer up to the point when their result is required. This is not important with very high depth complexity scenes, but a useful optimization with scenes of modest depth complexity. Practically all real-time graphics falls into the latter category. By using an Occlusion Write Postpone Queue the following useful optimizations are introduced:

1. The depth estimation buffer contains better (i.e. less conservative) values during the occlusion tests as the non-contributing occluders have not yet “trashed” the depth estimation buffer values.
2. The write queue working together with the incremental occlusion buffer can be used to calculate context-dependent occlusion powers. Section 3.3.2 discusses this in detail.
3. The depth complexity of the occlusion buffer is reduced as far-away occluders that cannot occlude anything are not rasterized
4. When combined with portal rendering, the write queue removes a large amount of unnecessary writes. In the best case, only the objects that overlap subsequent visible portals are rendered.
5. Improves CPU cache-coherence, because consecutive writes and reads are subjected to the same part of the screen, and thus to the same memory area.

Since occluder rasterization may not be a by-product of the actual scene rasterization [10], it is necessary to rasterize only useful occluders to minimize the rasterization costs. The properties listed above all help to reduce rasterization costs *and* to improve the quality of the occlusion and depth estimation buffers.

Our current implementation subdivides the screen into tiles (axis-aligned rectangles) and the write queue is locally *flushed*, if a visibility test passes and there are pending writes with closer depth values than the visibility test depth value. As a result, the contents of the occlusion buffer are not updated, unless there is a possibility to hide the object.

Flush Tabus

If we expect an object to be visible [11], we do not allow the object to flush the write queue. This rule decreases the amount of write queue flushes immensely. As we want every adapting algorithm to contain a small amount of random behavior, the objects that are expected to be visible have a 6% chance of flushing the write queue. Objects that are expected to be hidden can always flush the write queue.

[10] For instance, in HZ-buffer[51] the occluder rasterization is a by-product of the rendering algorithm, in HOM[121] it is not.

[11] Based on history data.

3.8 Silhouettes as Occluder Primitives

The silhouette of an object represents exact coverage information. This is easily proven by observing that drawing the object with a single color in fact creates the object's silhouette. Therefore it is not necessary to consider the actual object geometry when performing coverage writes [12]. HOM (2.4.4) renders the actual geometry to produce an image of silhouettes. This is unnecessary and we take an alternative approach and omit the phase of rendering the objects. Instead we extract the silhouettes in object-space, and directly use them to create the coverage buffer. This has the substantial advantage of reducing the edge complexity from O(n) to approximately O($sqrt(n)$) in many typical models [89].

In order to efficiently use silhouettes as occluder primitives, we must be able to extract, clip and render them as quickly as possible. We proceed by first optimizing silhouette extraction, then rendering and finally clipping. Algorithm for converting alpha textures into silhouettes is given in 3.8.4.

3.8.1 Temporally Coherent Silhouette Extraction

Brute-Force Extraction

Brute-force silhouette extraction is a linear algorithm and would be a dominating cost in our occlusion buffer algorithm (section 3.9). Occlusion-preserving simplification algorithms are still inadequate to be used in a commercial library, and usually the actual geometry has to be considered when rendering the occluders.

Output-Sensitive Extraction

Most output-sensitive silhouette extraction algorithms only work under orthographic projection and are thus not useful for our purposes. Sander *et al.* [89] describe an algorithm that also works with perspective projection. According to their results, their algorithm is faster than the brute-force algorithm, if the object has more than a 1000 edges.

Occlusion-Preserving Temporally Coherent Extraction

Theorem 1. *Silhouette of an opaque mesh $\mathcal{M}$ extracted from location $\mathcal{L}_0$ is a valid occluder when observed from $\mathbf{R}^3$ if a) $\mathcal{M}$ is two-sided or if b) $\mathcal{M}$ is a closed surface and only observed from the outside.*

Proof. Back-face culling $\mathcal{M}$ from location $\mathcal{L}_0$ produces a list of N opaque triangles T_i. Outlines of T_i are a piecewise representation of the silhouette of $\mathcal{M}$. Each one of the triangle outlines accurately bounds a piece of the actual surface, and is therefore a valid occluder when observed from $\mathbf{R}^3$. The above is true only if all of the triangles T_i are opaque from both sides.

If two triangle outlines share an edge, the edge can be removed to merge the triangles into a single outline, which is guaranteed to be a valid occluder when observed from $\mathbf{R}^3$. This relation holds when all shared edges are recursively removed. The operation of removing the shared edges is silhouette extraction. Therefore the relation holds for a silhouette.

[12] This is not of course true for the depth tests.

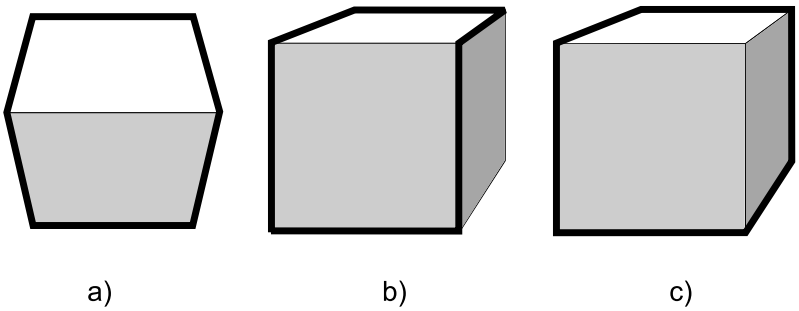

Figure 3.9 Re-transforming a previously extracted silhouette (a) to a new orientation (b) produces a conservatively wrong silhouette compared to (c).

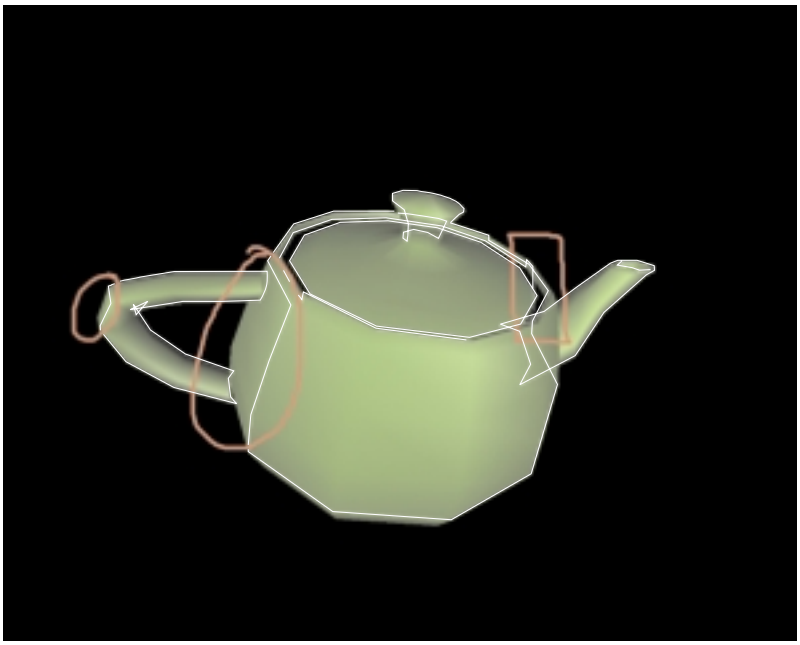

Figure 3.10 An example of a re-transformed silhouette. Two marked areas on the left are conservatively wrong, but the one on the right can produce false occlusion due to issues explained in the text.

The triangles T_i do not have to be opaque from both sides if only one side of each triangle can be observed. This is only true if the mesh is closed and observed from the outside. □

Any silhouette of an object is always a valid occluder. As long as the specified angular distortion constraint remains satisfied, the same silhouette can be re-transformed instead of extracting a new one. The re-transformed silhouette is an *occlusion-preserving impostor*. Figure 3.9 *a)* illustrates the original silhouette *b)* the re-transformed conservative silhouette and *c)* the correct silhouette.

We have implemented a silhouette cache, which stores object silhouettes and either finds the best existing match or creates a new silhouette depending on object and camera transformations. The re-transformed silhouette in figure 3.10 is extremely conservative. Most notably this can be seen in places marked with rounded symbols. The occurrence marked with a square symbol is the only true defect of re-transformed silhouettes. The teapot model violates the principle of bidirectional visibility and the re-transformed silhouette can therefore cause false occlusion. If that is not acceptable, such models should not be used as occluders or the models should be fixed.

We have witnessed silhouette cache hit ratios ranging from 50% to 100% depending on camera and object motion. On average we are able to skip approximately 95-100% of the silhouette extractions[13]. These figures could be further enhanced by allowing more distortion, which would in turn decrease the occlusion power of an object. Where the break-even lies is scene- and platform-dependent and we are currently adjusting the limits at run-time using simple feedback heuristics. To prevent all sudden moves from invalidating the caches, we allow the objects to initially use worse silhouettes and tighten the constraint according to time the silhouette has been used. [14]

The silhouette cache is independent of the actual extraction algorithm and effectively multiplies the performance. For relatively simple meshes with less than ˜2000 triangles we opt for using brute-force extraction to save memory. We have not implemented an output-sensitive extraction algorithm but currently the most appealing approach seems to be the *search tree* by Sander *et al.* [89]. It remains to be seen whether the output-sensitive extraction is actually needed. The options to avoid it are to split the model into several parts, use simplified geometry, or to disable the occlusion writes of a complex model.

[13] Silhouettes are cached on a per-model basis, thus many objects sharing a model can share a cached silhouette.
[14] We have also limited the total size of the silhouette cache and perform aggressive cleaning of the cache.

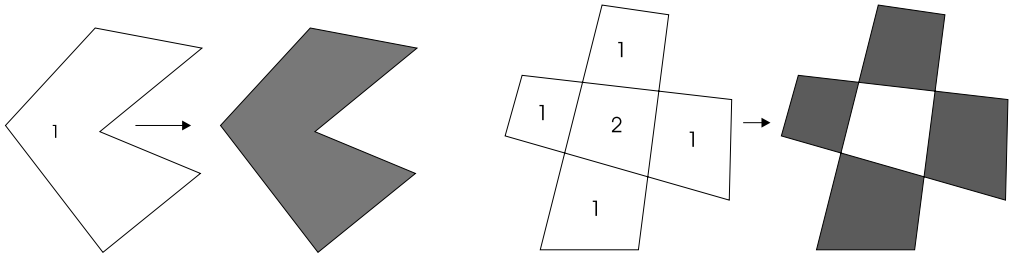

Figure 3.11 XOR filler handles correctly a single overlap (a), but fails when the overlap count is two or more (b). The numbers indicate the overlap count.

3.8.2 Silhouette Rasterization

A single bit per pixel is a sufficient representation of coverage. To alleviate the need for graphics hardware and downloading the frame buffer we create a specialized 1 bpp silhouette renderer, which needs a very high fill rate to compete against hardware rasterization.

Scan-line Filler

The usual approach [38][15] for silhouette filling is to assign all active edges into scan-line buckets. Value $+1$ is assigned to left edges and -1 to right edges. For each scan-line, the entries are sorted from left to right and $depth$ is initialized zero. For each entry in the scan-line, $depth = depth + entry.value$ and if $depth > 0$, a span is drawn from the previous to the current position-1. Though very simple to implement, this algorithm is not very efficient in practice. The need for sorting and the high amount of conditional processing limit the performance.

XOR Filler

Commodore Amiga 500® presented all graphics with bit planes[16]. Up to 5 bit planes could be displayed simultaneously and the final color of a pixel was indexed from a 32-entry palette. Amiga 500 also had a co-processor called *blitter*, that performed simple combinations of logical operations between two sources. Simple filled 3D objects were rendered by first drawing the outlines to the bit planes in XOR mode. Then the blitter was configured to perform operation 3.10 for each scan-line [17].

$$scanline_i = scanline_{i-1} \text{XOR } scanline_i \tag{3.10}$$

This approach only works for silhouettes without overlaps (Figure 3.11), but has the very desirable property of being totally unconditional and highly parallel.

[15] Active edge table.
[16] bit plane = 1 bit per pixel buffer
[17] In fact the blitter did not perform the operation vertically but horizontally.

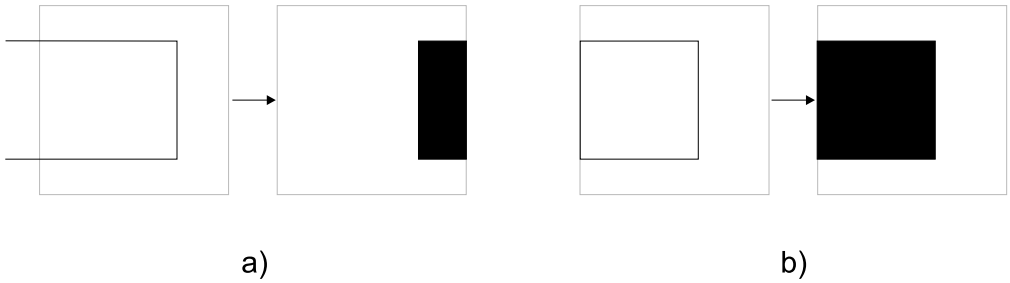

Figure 3.12 A silhouette clipping to the left plane. Clipping produces an incorrect result (a) whereas clamping produces the visually correct filler output (b).

N-bit Parallel Filler

We merge two previous algorithms by employing the algorithmic outline of the XOR filler and extending the logical operation to handle multiple overlaps. The basic idea is to store the per-pixel overlap counts into N bitplanes. The bitplane$_0$ stores the lowest bits, bitplane$_1$ the next bits and so on. The logical operations needed are identical to a hardware ALU accumulator. We first accumulate the left edges with value $+1$ and then the right edges with value -1. As our N-bit numbers are unsigned, adding -1 is equal to adding $2^N - 1$. The filler output bit is set if any of the bitplanes have a set bit. This operation reduces to a logical OR among all bitplanes.

Technically this never guarantees the correct output, since a silhouette can potentially have an unlimited number of overlaps. In practice, the amount seldom reaches $2^3 = 8$, which is the reason we are currently using $N = 3$. One key observation is that even if the capacity of the accumulator is exceeded (N>=8) the output is only conservatively wrong. This can be verified by observing that $M * 8 \Rightarrow 0$, which is incorrect, but no input sequence can result in $0 \Rightarrow !0$. Thus we ignore the problematic cases.

Each pixel on the scan-line is processed independently of each other and we only need logical operations. Therefore we can fit the bitplane data of 64 pixels into N 64-bit registers and are able perform the accumulation in parallel. On Intel Pentium III ®-based machines a 3-bit 64 pixel parallel accumulator executes in approximately 16 cycles, which gives the peak fill rate of $64/16 = 4$ pixels per cycle. Running at 700MHz this sums to 2.8 billion pixels per second. Even if such figures cannot be demonstrated in real-life scenarios, these numbers indicate that the filler is unlikely to become a bottleneck of the system.

3.8.3 Silhouettes of Clipping Objects

Our silhouettes are a simple collection of edges. The mesh facing is always clockwise and therefore a single comparison of y-coordinates suffices to indicate whether an edge is a left or a right edge. An arbitrary silhouette can have any amount of distinct closed regions and we do not require a specific ordering of silhouette edges. The silhouettes are defined with (x, y, z)- coordinates and are generally not planar before perspective projection.

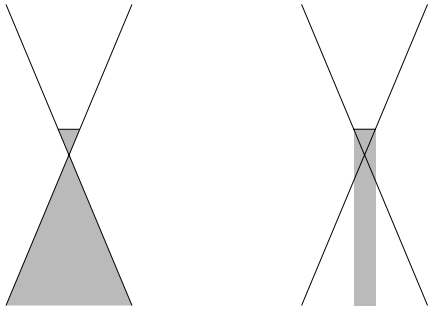

Figure 3.13 The gray area indicates the behind-the-viewport region of space from where a perspective projection (a) and a parallel projection (b) can project points into the viewport. Such a projection causes incorrect visual output and cannot thus be allowed. As can be seen from the figure, parallel projection has a significantly smaller gray region and consequently a larger valid backprojection area.

Spatial Clipping

Since our silhouettes do not have connectivity information, the clipping procedure only needs to consider a single edge at a time. Clipping to top, bottom and right planes of the view frustum are simple rejections. If the edge is completely outside any of these planes, it can be rejected. The edges partially outside are clipped, and the parts outside the planes are rejected.

On the left border the rejection is not possible. To assure correct filler functionality none of the edges outside the left plane must be discarded. Instead the edge should be clipped to the left plane and the clipped part of the edge should be clamped to the leftmost column. This is illustrated in figure 3.12 where a) shows the consequences of discarding an edge and b) shows the clamping and subsequent correct filler output.

Silhouette Splatting

Very often the best occluders are the ones that intersect the front clipping plane. By intersecting the front plane we do not mean objects that penetrate the viewport, but the ones that intersect the infinite plane outside the viewport. For example, most walls and floors in an architectural scene intersect the front plane.

An obvious approach is to clip the object to the front plane, and then to extract the clipped silhouette. This suffers from the fundamental flaw of not being able to utilize temporal coherence. [18]

The key observation is that the front clipping operation does not need to produce correct output, since it is only performed outside the screen. If an edge, or a part of an edge, is behind the front plane, we can *clamp* it to the front plane by using a parallel projection. A point can only get parallel-projected inside the viewport if it lies inside the gray area of figure 3.13 b). We call this gray area *back projection shaft*. An object intersecting the back projection shaft is considered an invalid occluder. Since anything *outside* the back projection shaft gets projected *outside* the viewport, the observed filler output is always correct *inside* the viewport.

[18] Silhouette cache in our case.

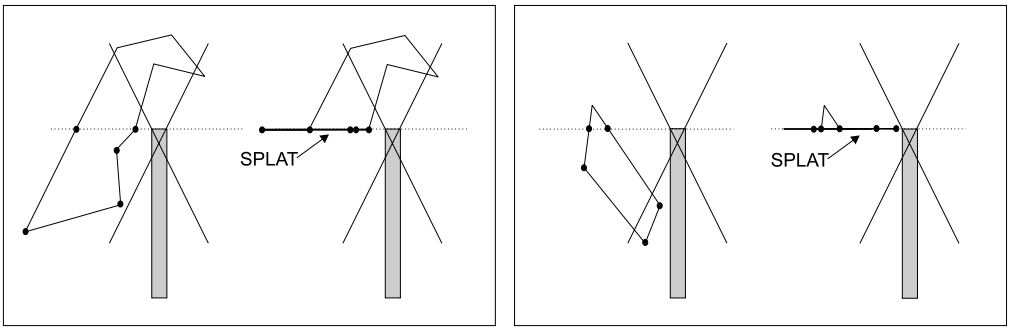

Figure 3.14 Silhouette splatting. If an edge of a front clipping silhouette is partially outside the front plane, the intersection point is calculated and the outside part is parallel projected to the front plane. Note that the silhouette on the right intersects invalid backprojection region of perspective projection but is a valid occluder if parallel projection is used (see figure 3.13).

What actually needs to be done is a parallel projection of the *front clipping parts of edges of the silhouette* to the front plane. If the edge is partially outside the front plane, the intersection point is calculated and the outside part is parallel projected to the front plane. We call this operation *silhouette splatting*. The operation is illustrated in figure 3.14.

For comparison, if perspective projection is used, the points inside the gray area in figure 3.13 a) can get projected inside the the viewport. Normally the view plane distance is very small [19] and the *back projection shaft* of b) converges to a line. Therefore the invalid gray area in b) is much smaller than the one in a) and thus more front clipping occluders can be accepted.

The final rules for using cached silhouettes are:

1. If the *object* does not intersect the front plane, it is a valid occluder.
2. If the *object* does not intersect the back projection shaft, it is a valid occluder.
3. The front clipping silhouettes of valid occluders must be splatted.

Discussion

Not having to reject front clipping occluders more than doubled the performance of some of our test applications. During the development of Umbra this has been performance-wise the single most important issue. The silhouette splatting algorithm currently used can still reject some of the good occluders, but we consider the ability to use temporally coherent silhouette extraction more valuable than the rare unnecessary rejections. In fact, such rejection can only occur when an object intersects the back projection shaft without penetrating the viewport.

3.8.4 Silhouettes of Alpha Textures

Rendering the silhouettes directly loses the possibility of using primitives with alpha textures as occluders. This may or may not be an important issue depending on the application type. If alpha

[19] Optimally it would be zero, but is usually set to > 0 in order to get floating-point math working and to avoid certain singularities.

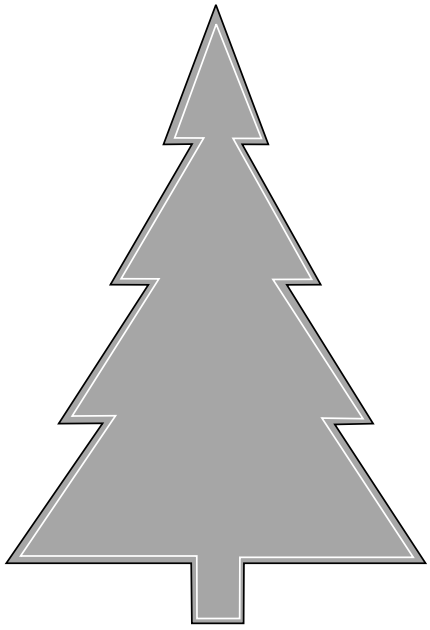

Figure 3.15 Converting an alpha texture into a silhouette. Any picture can be converted into a silhouette according to the alpha channel. The silhouette can be simplified as long as every edge is contained by the opaque area.

textures contribute significantly to occlusion, the texture can be converted into a simplified silhouette according to the alpha channel of the texture[20]. Figure 3.15 illustrates the general idea. The job is easy to perform manually, but in the case of extensive use of alpha textures as occluders, an automatic extraction is preferable.

The general outline of a brute-force algorithm goes as follows:

1. Separate the opaque and transparent areas with flood fill.
2. Construct closed boundary silhouettes of all opaque areas.
3. Store vertices in a priority queue, sorted by *opaque area loss*.
4. Simplify boundary by removing one vertex at a time.
5. Allow the removal only if both new edges are fully within the opaque area.

The algorithm outlined is suitable for pre-processing purposes. Whereas the occlusion-preserving simplification of 3D meshes is a challenging problem, in 2D it is mostly trivial.

Consider the case of a checkerboard texture where the black texels are opaque and the white ones are transparent. Such a texture map can be tiled arbitrarily many times over a single polygon. Thus, the exact alpha mesh of the polygon can be arbitrarily complex. Therefore meshing according to alpha channel should not attempt to be exact; occlusion losses should be accepted in areas where the edge complexity would become too high.

[20] A texture mapped on a curved surface can create a very complex silhouette and the approach presented here is mainly suitable for planar surfaces.

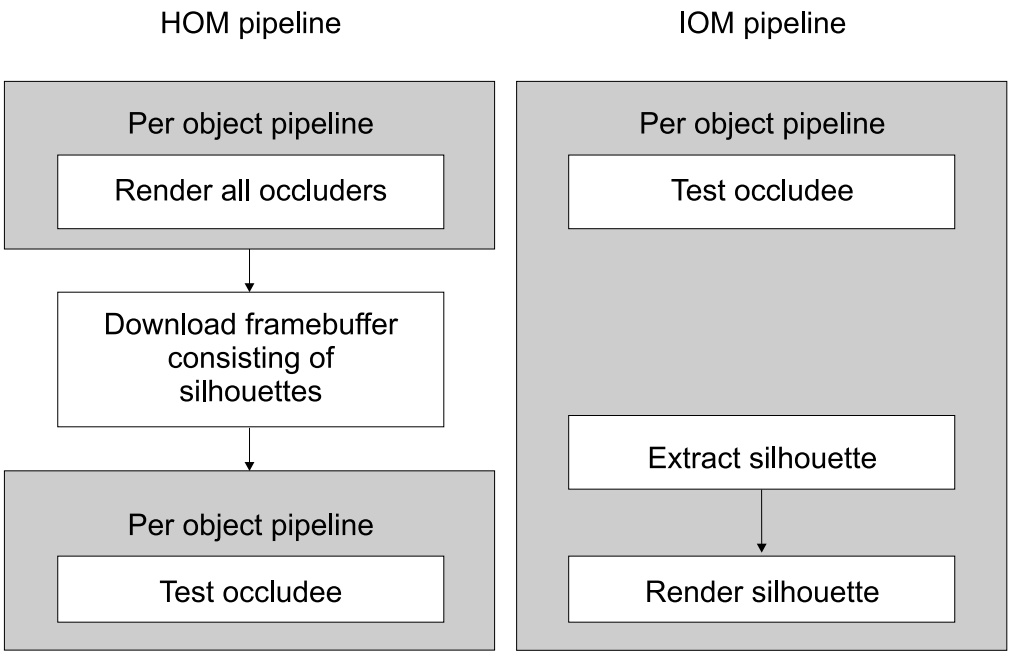

Figure 3.16 Comparison of HOM and IOM pipelines shows that IOM pipeline only includes per-object operations whereas HOM pipeline has a pipeline wreck in the middle.

3.9 Incremental Occlusion Maps (IOM)

3.9.1 HOM and IOM Pipelines

HOM (section 2.4.4) and IOM pipelines are shown in 3.16. The fundamental difference is that HOM first renders all occluders and then downloads the frame buffer consisting of silhouettes of the occluders. Downloading the frame buffer is not a per-object operation. This pipeline interrupt is important, since it disables the possibility of getting valuable feedback of the usefulness of the occluders.

IOM pipeline only consists of per object operations and makes it possible to incrementally add more occluders when needed. This in turn allows us to get estimates of the occlusion powers of the individual occluders. Occluder selection algorithm that uses this information is explained in section 3.3.

3.9.2 Occluder Fusion

Initially the *coverage buffer* is empty and the DEB is initialized to nearest depth value. We store a single floating-point depth value for each 8x8 *block* in the coverage buffer. This corresponds to a memory consumption of 0.5 bits per pixel.

For each occluder we fill the silhouette to a *cache* and then perform occluder fusion by ORing the cache to the coverage buffer. If the block in the coverage buffer changes during the occluder fusion,

we update the corresponding depth value.[21] As a result, the depth updates are limited to the blocks that satisfy both of the following conditions:

1. The silhouette at least partially covers the block.
2. The silhouette actually contributed to the corresponding coverage buffer block.

Comparison with HOM

HOM updates the depth values of all blocks that are at least partially covered by the *axis-aligned rectangle*, regardless of the actual contribution to the coverage buffer. As a result, our depth estimation buffer contains less conservative depth values. Since we can collect the coordinates of changed coverage buffer blocks into a list, we can optionally derive more accurate depth values only for the required blocks.

[21] Depth update only occurs if the current depth value is *bigger* than the value stored for the block.

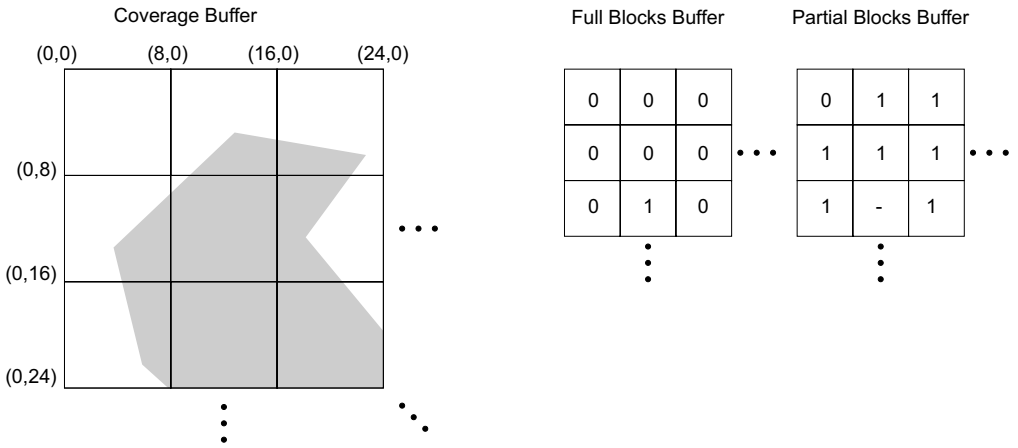

Figure 3.17 The coverage information of IOM is stored in three buffers. The Coverage Buffer contains the actual coverage information. The Full Blocks Buffer has a set bit if the corresponding 8x8 region in the Coverage Buffer is full and the Partial Blocks Buffer has a set bit if the corresponding 8x8 region is partially full. If the Full Blocks Buffer has a set bit, the Partial Blocks Buffer value is redundant.

3.10 Implementing Incremental Occlusion Maps

This section outlines an efficient implementation of incremental occlusion maps. The implementation uses a custom three-level hierarchy of the coverage buffer. We call these levels *bit*, *block* and *tile* resolution buffers respectively. Bit resolution equals output resolution, block resolution is $\frac{1}{8x8}$th and tile resolution is $\frac{1}{32x64}$th of the output resolution.

Each tile has a corresponding tile in the write queue (section 3.7) and stores information whether the corresponding coverage buffer area is completely covered. When the coverage buffer is cleared, the *Pending Clear Flags* are set to the tiles instead of actually executing the clear operation.

Whereas HOM uses a flat depth estimation buffer, IOM uses a hierarchical presentation in order to have faster depth queries for objects with a large screen coverage.

We begin by introducing the different buffers and proceed with tiled silhouette rasterization and hierarchical construction. Occlusion queries are explained in section 3.10.4. An alternative contribution culling technique, called *construction-time contribution culling*, is introduced in section 3.10.5. Incorporation with a stencil buffer and a useful optimization algorithm called *validation buffer* are explained in 3.10.6 and 3.10.7 respectively. Finally, conclusions on IOM are given in 3.10.8.

3.10.1 Buffers

Contents of the buffers to be explained next are illustrated in figure 3.17.

Coverage Buffer is a bit resolution 1 bpp coverage buffer, where each output pixel has a corresponding “coverage bit”. The bit is set if the pixel is covered.

Full Blocks Buffer is a block resolution buffer. A bit is set if all of the corresponding 8x8 pixels in the Coverage Buffer are set.

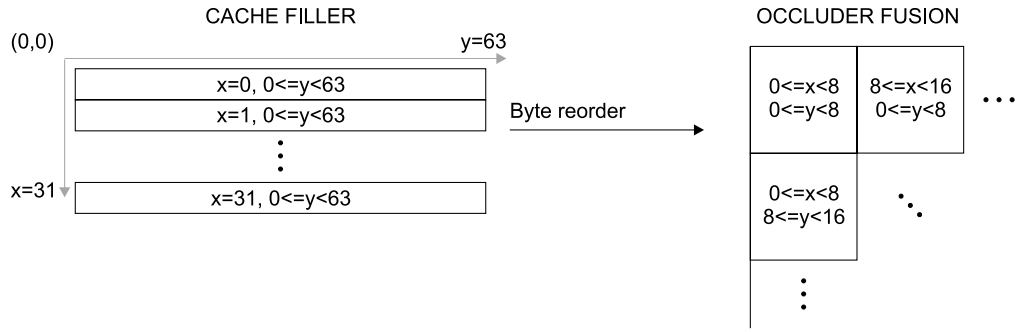

Figure 3.18 "Byte reorder" operation arranges filler output so that the data of each 8x8 blocks is consecutive in the memory. After the byte reorder operation each 8x8 block can be very efficiently manipulated and on native 64-bit architectures it fits into a single register.

Partial Blocks Buffer is a block resolution buffer. A bit is set if any of the corresponding 8x8 pixels in the Coverage Buffer are set.

Hierarchical Depth Estimation Buffer is a block resolution buffer at the lowest hierarchy level. The higher levels are updated only when needed (see section 3.11).

3.10.2 Tiled Silhouette Rasterization

We have chosen to subdivide the output surface into 32x64 pixel tiles and to perform all filling operations inside a tile-sized 3-bit plane cache. We run the parallel filler in the cache from left to right. For vertical lines inside the cache we use the term *column*. Performing the filling operation using a fixed-size cache is straightforward and each tile only requires the last column of the previous tile. We call this last column the `fvalue` .

After the tile is filled into the cache, we perform a *byte reordering* operation to get 8x8 pixel masks into 64-bit registers (figure 3.18). The coverage buffer is updated by performing a logical OR operation with the masks. This results in occluder fusion.

Tiles that are 64 pixels high result in a lot of unnecessary rasterization. However, modern CPUs execute 64-bit operations generally faster than 8-bit operations. On the other hand, the byte reordering and the merging to coverage buffer are costly operations and should be avoided if possible. Thus we need to track *dirty rectangles* for each tile. Vertical bounds are easily obtained by ORing all of the columns together inside the filler. As the *outlines* are drawn into the cache before executing the filler, we can track the horizontal bounds there (figure 3.19).

We know the horizontal bounds before executing the filler and can therefore limit the execution to the dirty region. In a large number of cases, the size of the dirty region is only a fraction of the full tile. A favorable special case is a tile without any outlines crossing it, as large objects usually cover several tiles completely. All we have to do to fill such tiles is to set the current `fvalue` to each column of the cache. If the `fvalue` has all bits set, the tile is fully contained by the silhouette and the cache filler and the byte reordering operations can be skipped.

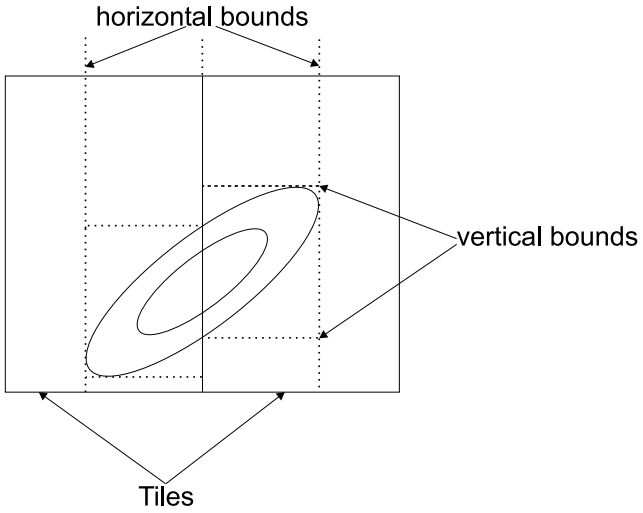

Figure 3.19 Vertical and horizontal bounds inside the tiles of the IOM filler. The horizontal bounds are used to limit the filler work and both of the bounds limit the byte reorder work (see figure 3.18).

3.10.3 Hierarchical Construction

Having introduced the buffer hierarchy, we can construct a faster version of the occluder fusion algorithm that both uses and updates the hierarchical buffers:

1. Skip the tile, if marked as full.
2. If the tile has a pending clear, clear the corresponding coverage buffer area.
3. Execute parallel filler to the cache.
4. For each dirty block inside the cache:
   (a) Skip the block, if corresponding bit is set in Full Blocks Buffer.
   (b) Merge cache block with Coverage Buffer block if no pending clear was set.
   (c) Skip the block, if Coverage Buffer did not change.
   (d) Append block coordinates to dirty blocks list.
   (e) If the block became full, set bit to Full Blocks Buffer.
   (f) If the block is partially full, set bit to Partial Blocks Buffer.
   (g) After all the blocks have been processed, update the HDEB for all blocks listed in the dirty blocks list. Set the modified region to the HDEB to allow lazy updating of the hierarchy.

The first step deserves further investigation. If the processing of a tile is skipped without regarding the outlines inside the tile, the `fvalue` is incorrect when starting to process the next tile. To alleviate the problem, when a tile becomes full, the next non-full tile on the right is searched and set as a *reflection target*. If no such tile is found, NULL is set to indicate that the outlines can be fully ignored. The outlines crossing a full tile are set to the leftmost column of the reflection target instead

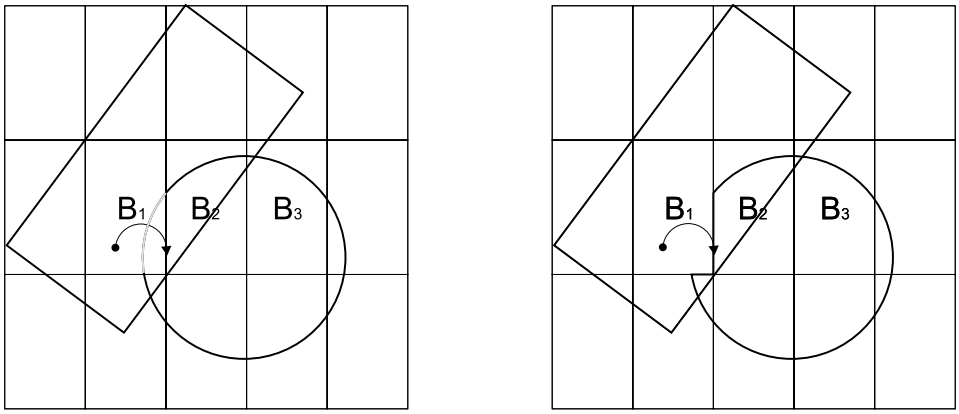

Figure 3.20 The box on the left has already been rasterized and therefore the bucket B_1 is full. A full bucket starts rejecting writes by setting its "reflection target" to the bucket on the right, in this case B_2. When the circle is about to get rasterized, the part marked with lighter gray gets "reflected" to the leftmost pixel column of B_2, as indicated in the figure on the right. After the modified circle has been rasterized, B_2 becomes full and its reflection target is set to B_3. Furthermore, the reflection target of B_1 has to be changed to B_3 as B_2 no longer accepts writes.

of the cache. Figure 3.20 illustrates the process. On the left, the B_1 tile is full and has its reflection target set to B_2. The parts of the circle that overlap B_1 are assigned to the leftmost column of the B_2 tile. Consequently no outlines are assigned to B_1 tile and therefore the tile can be skipped while rasterizing the circle. After the circle has been rasterized, B_2 becomes full and reflection targets of both B_1 and B_2 are set to point into B_3. Had B_3 been outside the screen, both reflection targets would have been assigned NULL instead.

The HDEB is only updated in places where the silhouette alters the contents of the coverage buffer. Therefore the depth estimation buffer has less conservative values than the original HOM approach, which used axis-aligned rectangles to determine the DEB area to update.

When the depth values are written into the HDEB, they are either approximated to be the object's zFar or the conservative Z value obtained by casting a frustum through the object's oriented bounding box. For planar objects, such as walls, this will produce exact depth values.

The algorithm presented skips work at every level. Figure 3.21 shows the actual contents of the Full Blocks Buffer (gray) and the Coverage Buffer (white). It is clear from the picture that our algorithm avoids touching the coverage buffer whenever possible. As a result, the fill rate increases when the screen becomes more covered. First the blocks, then the tiles and ultimately the entire coverage buffer get full, and start rejecting writes.

3.10.4 Occlusion Queries

We support three types of queries:

Figure 3.21 Contents of the Full Blocks Buffer and the Coverage Buffer marked as white and gray respectively. Cache coherence is improved since the higher resolution Coverage Buffer must only be accessed if partial 8x8 coverage is encountered.

Point Query

A point query tests the visibility of a single screen-space point. A depth test is performed prior to the coverage test. The coverage is first tested from the Full Blocks Buffer. Unless full, the Partial Blocks Buffer is tested to make sure there is a chance to succeed, if a full resolution query is used. Last, the partially full Coverage Buffer block is tested for accurate coverage information.

Rectangle Query

A rectangle query tests visibility of an axis-aligned rectangle. The coverage test is first performed using the block (8x8) resolution. If only partial coverage was encountered, a list of partially filled blocks is built and their accurate coverage is tested from the Coverage Buffer. This behavior is identical to the triage masks used by Greene in [50]. Finally, a depth test is carried out using HDEB.

Each bit of the Full Blocks Buffer encodes the visibility status of 8x8 pixels. A single test of 32/64 bits can be carried out in approximately 2 cycles[22], which results in a peak test rate of 1024/2048 pixels per cycle[23] and gives 0.7/1.4 TeraPixels at 700MHz. This clearly indicates that coverage testing will hardly be a bottleneck of the overall system.

Silhouette Query

A silhouette query tests the visibility of a *convex* silhouette. Each block that is even partially covered by the silhouette is tested for both coverage and depth. As the silhouettes are convex, we can use the XOR filler (see section 3.8.2) for additional fill rate.

All occlusion queries first execute a point test, then a rectangle test, and finally a silhouette test. The test silhouette for an object is extracted from its OBB. Spatial database nodes are tested one face at a time, and each node has 1-3 visible faces.[24] Since the faces are always planar, we employ a more accurate depth test for database nodes. If the normal depth test that uses the closest depth value of the face fails, we evaluate a more accurate depth value for each 8x8 block using the plane equation of the face.

3.10.5 Construction-Time Contribution Culling

Traditionally contribution culling has been done in the occlusion *test* (section 2.5.1). If contribution culling is done at construction time, there is no need for transparency values and a single bit per pixel representation is sufficient for testing.

We normally set a bit in the Full Blocks Buffer when all 64 (8x8) pixels are set. We might just as well set the bit when a certain amount of pixels are set. For example, setting the bit into the Full Blocks Buffer when 60 of the 64 corresponding pixels are set, results in potential rejection of objects with less than 6% visible pixels.

[22] On current x86 processors

[23] For a 32-bit CPU: $32bits * 64pixels/2cycles = 1024pixels/cycle$.

[24] If the camera is located inside a voxel, we always consider the voxel visible.

Error Bounds

With HOM (section 2.4.4) the *levels of visibility* are calculated hierarchically and the values in the higher levels provide no information about how the vacant pixels are distributed. In the worst case all of the vacant pixels can be consecutive. If contribution culling is done on an 8x8 block basis, the visible error is tightly bounded.

3.10.6 Stencil Buffer

In order to support complex portal sequences, the hardware must have a stencil buffer. Similarly the occlusion buffer must support stencil buffers to allow occlusion writes behind portals that use stencil masks.

A single bit presentation is sufficient for portal traversal purposes. We can implement the stencil buffer very efficiently by storing it in the same format as the cache. Initially the stencil buffer is set to fully covered, and subsequent stencil masks are applied to the stencil buffer by using a logical AND operation. The stencil buffer is applied to the silhouette being rendered by performing a logical AND operation *before* merging the cache contents into the coverage buffer.

The stencil test is performed for 64 pixels at a time and executes virtually for free. Rendering the stencil masks is the same operation as the silhouette rendering and an almost identical code is executed.

3.10.7 Validation Buffer

Our database traversal can guarantee that objects that are in front of a database node are processed before the node itself. In other words, when a node is processed it is known that nothing that is processed later can be closer to the camera than the node.[25] This opens a new possibility.

Whenever a pixel of a node is found to be hidden, a bit can be set to the *validation buffer*. A set bit in the validation buffer indicates that *no depth comparison is required* for the pixel. The Validation Buffer can be made hierarchical in the same way as the Coverage Buffer (see 3.10.1).

Whenever a coverage test is being made, the Validation Buffer is used to mask parts of the primitive that are hidden regardless of the depth. As the implementation is hierarchical, the rejections become very fast when the *occlusion frontier* is reached. Ultimately the entire validation buffer becomes full, and all subsequent occlusion queries succeed without actual tests.

The Validation Buffer essentially converts a HOM into a coverage buffer of “Hierarchical Polygon Tiling with Coverage Masks” [50], but unlike Greene’s algorithm, it can also be used without strict front-to-back traversal.

The Validation Buffer has the following important properties.

1. Reduces depth tests.
2. Reduces HDEB subsampling loss.[26]

[25] In the node’s screen projection

[26] If a lower-resolution depth estimation buffer is used, occluders that are rasterized into the occlusion buffer "leak" their depth values to nearby pixels. After a pixel becomes validated, it cannot be affected by such leaks.

3. Can be maintained with very little extra cost.

3.10.8 Conclusions on IOM

Advantages

1. Provides incremental occluder fusion with feedback and therefore enables sophisticated occluder selection.
2. No hardware rendering is required and therefore the need to download the frame buffer is alleviated. Performance of our software implementation is comparable to current high-end hardware rasterization.
3. Uses full output resolution and is therefore truly conservative, unlike HOM. Subsampling can be performed internally to reduce rasterization costs at very high resolutions.
4. More accurate updates of the depth estimation buffer compared to HOM.

Drawbacks

1. Cannot directly handle alpha textures as occluders.
2. A lot more complicated than HOM. Our current implementation of the IOM pipeline is about 10.000 lines of C++ code.

Additionally static coverage masks can contribute to occlusion. This is useful when the application has a static head-up display (HUD) or other kind of on-screen display that can cause occlusion.

3.11 Hierarchical Depth Estimation Buffer

Depth Estimation Buffer (DEB)[121], reviewed in section 2.4.4, is particularly suitable depth representation for software implementation. The data structure of the *hierarchical depth estimation buffer* (HDEB) is very similar to the Hierarchical Z-buffer (section 2.4.2). The lowest level of the HDEB pyramid is identical to DEB. For practical reasons we limit the horizontal and vertical resolutions of higher pyramid levels to be powers of two. The buffer does not need to be square. The highest-precision level, however, can be of arbitrary resolution. Thus memory is only wasted in the much smaller hierarchical layer structures.

Construction

Lazy evaluation is used to postpone and to combine update operations. This postponing/combining makes the HDEB updates feasible. A similar idea was proposed by Xie and Shantz [119] for a hardware implementation of HZ-buffer. They construct the HZ-buffer from the real Z-buffer several times per frame. The main difference is that we never get worse results than by using a true Z-buffer. In addition to the depth values (32-bit floats) a single-bit *modification mask* is maintained per pixel. Whenever the contents of the real Z-buffer are modified, the modification masks of the screen-space bounding rectangle of the object are set. Additionally a single-bit mask is maintained for each horizontal scan-line of each layer. This bit contains the logical OR of all modification bits in the scan-line. Thus if the *scan-line modification bit* is not set, none of the pixels in the scan-line have been modified.

Whenever a query is made, depth buffer levels are updated based on the modification masks. Levels are updated bottom $\rightarrow$ up and modification masks are propagated upwards by performing a logical OR of the 2x2 modification bits when changes happen. Lookup tables are used to perform the bit-swizzling required in this process. Only the modified pixels are updated, and only true modifications are propagated upwards. The HDEB depth values are never cleared. Instead the modification bits of the bottom level are all set at the beginning of each frame. Thus the HDEB contains a lot of the time "false depth information".[27] This does not matter, since the depth values will become correct when an actual query is made. Conditional logic in updates can be removed by using the conditional move and maximum instructions available in newer processors.

Queries

A depth query resolves whether the current object is entirely behind the depth values stored into the HDEB. The queries are either point or axis-aligned rectangle queries. Point queries, that in our occlusion culling system mainly originate from Visible Point Tracking and silhouette queries (IOM), are performed directly on the lowest pyramid level. Rectangle queries are performed in a fashion similar to the HZ-buffer[28]. Even large objects can thus be tested extremely quickly. For each query we determine the correct *starting level* based on log_2 of the smaller spatial dimension of the query rectangle. Only levels up to the query start level need to be updated. For small objects (small query rectangles) we perform the queries using the original depth buffer. This means that the HZ-buffer does not need to be updated for small objects, which is an important saving. Also, it is faster to sample the depth buffer directly than to use the more complicated traversal when there is only a few pixels to test. Currently we use an area of 32 pixels as the break-even point.

[27] See section 6.3 for discussion about shadow acceleration. In that case the invalidations are only needed for moving visible objects.

[28] logarithmic search.

Results

The HDEB works very well in practice. The time required to perform the updates is insignificant in all of our test applications, typically being 1-2% of Umbra's CPU consumption [29]. In worst-case situations, a time usage of 4% has been measured. In many test scenes the updates are performed only ˜1-4 times per frame on average. In worst-case scenarios over 80 updates/frame have been measured. However such updates are very localized, and we need to update only a very small portion of the HDEB at a time. The queries are very efficient: time consumption is typically $< 1\%$ of overall Umbra CPU usage. Major reasons for efficiency are that relatively low resolution HDEBs are used and that queries are made only for objects that have passed the coverage test. Also, the depth buffer is updated only for pixels that actually contributed to the depth buffer.

[29] CPU consumption of the visibility query, not rendering time.

3.12 Object-space Techniques

Object-space techniques can be exploited in order to better utilize temporal with occlusion buffer algorithms. In the next two sections we briefly review two optimization techniques that can be used to increase the performance of occlusion buffer algorithms.

3.13 Virtual Occluders

The research on this subject continues and Umbra does not currently use virtual occluders (VO). Virtual occluders are introduced and their short history is outlined in section 2.1.

Our virtual occluders will be the *nodes of the spatial database* as in [11]. Thus we can recursively combine virtual occluders into bigger entities. Since we track how long objects and nodes have been hidden, we have a rather good initial guess for virtual occluder candidates. We also know if an object is *currently* static and consequently accept only static occluders to hide virtual occluders.

Whenever a virtual occluder is created, we store the set of occluders used to hide it. If any of the objects used to hide a virtual object becomes dynamic, this *agreement* expires and the VO is destructed. Our agreements are similar to those in [26][30] except that we create and delete the agreements dynamically.

When the database traversal encounters a virtual occluder, that VO is rasterized into the occlusion buffer. VOs that cannot be merged in object-space are thus merged using image-space processing. The occluder selection algorithm used in Umbra can automatically adjust its selection process when VOs are introduced. The main performance improvement that can be achieved from using VOs is the diminished occluder rasterization cost, as the real geometry is not necessarily needed.

VOs fit extremely well into the framework, and we are currently trying to find a generic algorithm for proving that a virtual occluder is hidden from a region. Our goal is to be able to prove the existence of one new virtual occluder per frame.

[30] Used to assign personal occluders for detail objects.

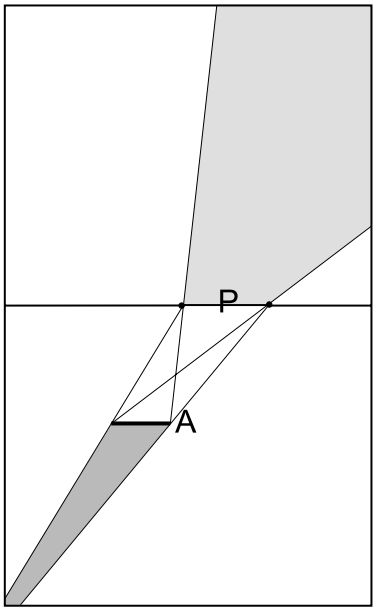

Figure 3.22 The Potentially Visible Set marked in light gray is conservatively valid for all viewpoints in the dark gray area. The portal is marked with the letter *P* and the area light source polygon with the letter *A*.

3.14 Portal PVS

All view frustum culling algorithms can exploit temporal coherence by constructing and caching Potentially Visible Sets of objects using temporal bounding volumes (TBVs) of view frusta over a period of time. Such an algorithm periodically constructs new PVSs and performs exact view frustum culling only for objects in the PVS. The PVS can be constructed and stored hierarchically, if the scene is organized in a spatial database.

Because of the similarity between portal culling and view frustum culling, the same method can be used for constructing *portal PVSs*. The Potentially Visible Set is created by finding all objects that intersect the penumbra formed by an area light source and a convex portal. See figure 3.22 for an example. The light gray area is the PVS penumbra formed by the portal *P* and the area light source *A*. The dark gray area is the corresponding *view cell* of the PVS.

The area light source is a convex polygon, having the same shape as the portal and located perpendicular to the portal. The size and exact position of the polygon may vary. [31] The *view cell* for the PVS is the open-ended frustum formed by connecting each edge of the area light source with the corresponding portal edge, and then *capping* the other end by the area light source polygon.

The penumbra frustum is created by constructing planes between each edge in the area light source and the furthest vertex [32] in the portal, and then capping by the portal polygon. [33] Finding the objects intersecting the penumbra is trivial, as the intersection test is an ordinary view frustum culling test.

A new PVS is constructed whenever the old one becomes invalid, i.e. when the camera moves

[31] This choice of shape and orientation avoids generating quadratic surfaces for the view cell boundaries.
[32] in the negative half-space of the plane
[33] This overestimates the size of the PVS frustum, but avoids constructing quadratic surfaces and keeps the frustum convex.

outside the associated view cell. The size and actual positioning of the area light source can be selected based on the velocity of the camera and by deciding an *expected validity period*. The algorithm can utilize additional coherence by caching several PVSs with view cells of different sizes. A larger PVS can then be used as a starting point for constructing a tighter one.

The portal PVS algorithm can be used to store Potentially Visible Sets of arbitrarily long portal sequences. Because the view cell is represented in the portal's space, the portal PVS can be also used for storing PVSs of virtual portals, such as mirrors.

Although dynamic objects cannot be stored into the portal PVS, they can be easily used by storing visible voxels of the spatial database into the PVS - the dynamic objects inside these voxels must then be tested each frame. By storing volumetric information rather than objects in the PVS, the PVS stays valid regardless of object motion.

The portal PVS is not particularly effective when the camera comes very close to the portal, as the view frustum becomes then extremely wide, and the PVS converges into the full set of objects in the target cell. When such a case is detected, the portal PVS should be temporarily disabled. [34]

Note that back-face culling can be used when creating the PVS. If a polygon is back-facing when viewed from all of the vertices of the area light source, it can be removed from the PVS.

[34] Alternatively, if volumetric information is stored rather than object lists, a pointer to the root node can be stored. This way the algorithm requires no special treatment of the bad cases.

Feature	Virtual Occluder	OcclusionBuffer	Portal
Resolution-Independent	+	−	+
Can handle transparencies	−	some can	−
Can use dynamic occluders	+	+	−
Static occluder fusion	+	+	+
Dynamic occluder fusion	−	+	−
Temporally coherent	+	−	section 3.14
Source area culling	+	−	−
Destination area culling	subdivision	subdivision	cell
Needs pre-processing	−	−	+
Contribution culling	−	+	−
Depth estimation loss	small (view indep)	high (view dep)	small
Memory consumption	agreements	view size	small
Occluder selection	complex dynamic	dynamic	static
Object processing order	arbitrary	front-to-back	portal order
Occludees processed	once per frame	once per frame	once per portal

Figure 3.23 Comparison of algorithmic properties of Virtual Occluders, Portals and Occlusion Buffer.

3.15 Conclusion

No single algorithm provides all desirable features, and a combination of some sort seems appealing. Combination of the occlusion buffer and virtual occluders forms a nice algorithm handling everything but transparencies correctly (Fig. 3.23). Neither of these algorithms require pre-processing, although both can benefit from it. Portals require pre-processing, but are extremely powerful in architectural scenes, and can additionally be used to create a wide variety of special effects. Portals are essentially a view frustum culling algorithm, and can thus be easily incorporated into a visibility determination framework.

Portal culling and occlusion culling are complementing algorithms, since portal culling can be used to reduce occlusion culling work, and occlusion culling can be used to cull portals.

Chapter 4

SurRender Umbra Architecture

In this chapter we list the algorithms used inside the library, outline the SurRender Umbra™ architecture and introduce the Umbra Visualizer™ run-time debugging tool. Accuracy and robustness issues are considered in section 4.4.

The information presented in this chapter is intended to help the reader to understand the library architecture - not how to *use* the library. How the library can be integrated into applications is discussed in the “Programmer’s Guide”. The API documentation is publicly available at **www.surrender3d.com**.

4.1 Algorithms Used in SurRender Umbra

Our implementation is approximately 70.000 lines of C++ code. The primary reason for the amount of code is the number of algorithms used. We have also tried to make the framework as extendable as possible in order to integrate impostors and other future extensions (see chapter 6) into the architecture with modest effort. The following algorithms are currently used inside the library:

- Hierarchical View Frustum Culling (section 2.4.9)
- Visible Point Tracking (section 3.4)
- Incremental Occlusion Maps (section 3.9)
- Hierarchical Depth Estimation Buffer (section 3.11)
- Occlusion Write Postpone Queue (section 3.7)
- Silhouette Cache (section 3.8.1)
- Incremental Occluder Selection with Feedback (section 3.3.2)
- Portals (section 2.4.8)
- Dynamic Spatial Subdivision and Traversal (section 4.6)
- Temporal Bounding Volumes (section 2.2)
- Validation Buffer (section 3.10.7)
- Construction-Time Contribution Culling (section 3.10.5)
- Dynamic Virtual Occluders (section 3.13)

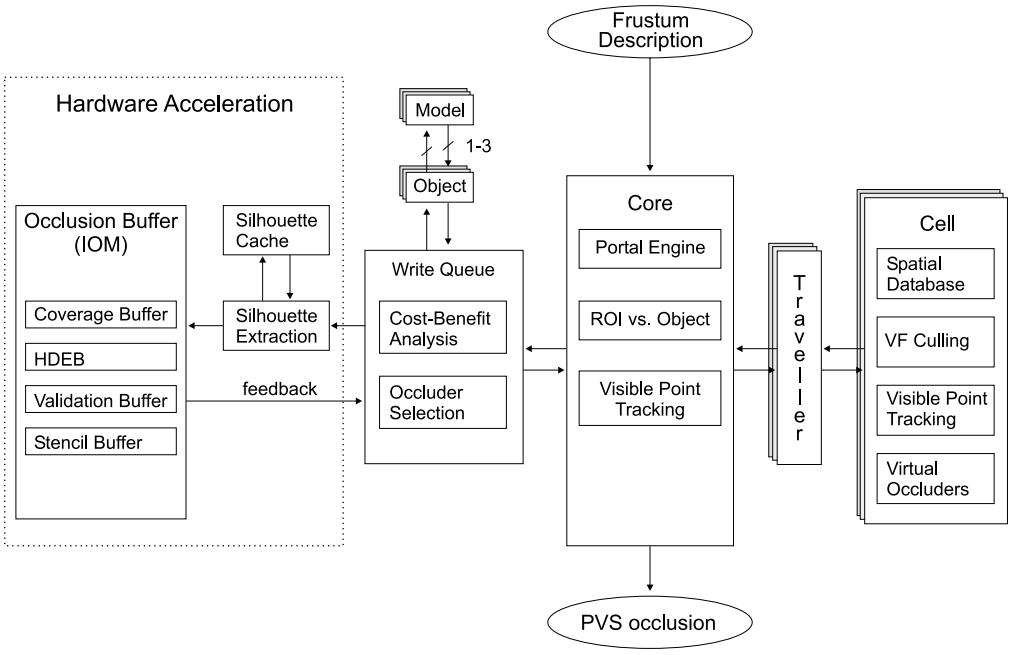

Figure 4.1 Block diagram of SurRender Umbra

4.2 A Block Diagram of SurRender Umbra

Figure 4.1 shows the block diagram of SurRender Umbra. There are a few issues in the figure that need further explanation.

4.2.1 Concepts

Visible Point Tracking

Visible Point Tracking (VPT), see section 3.4, is performed in two different places (core and cell). This is because VPT is used for both view frustum culling in the spatial database and for occlusion culling in the core.

Traveller

Traveller is an interface class between the core and a cell. The current view frustum description is set to the traveller before the traversal of the cell begins. All information exchange between the core and a cell is handled through a traveller object.

ROI vs. Object

Region of influence (ROI) is a term used for a "thing" that affects other objects somehow. The most obvious usage are light sources. Thus "ROI vs. Object" stands for resolving which lights affect which objects. Umbra defines ROIs as special objects that are handled as normal objects all the way through the visibility pipeline. Before informing the user about visible objects, the intersections between ROIs and other objects are resolved. Since the visibility pipeline of normal objects is used, ROIs can be eliminated by VF and occlusion culling. As a result, Umbra only reports ROIs that are visible or almost visible. Umbra does not concern different light types or lighting models in any way. Since the ROIs are objects, the test model can be anything from a sphere to a mesh. For example, point light sources can be modeled by setting a sphere as the test model and spot lights by submitting a cone-shaped mesh.

The basic algorithm used for object versus ROI intersection testing is sweep and prune. The optimal sorting axis is found by incrementally calculating the standard deviation of ROI center-point locations in three dimensions and normalizing the result to get the direction vector. The choice of sorting axis has been found very important, and the maximum variance algorithm being used is extremely good in practice. To further enhance the performance of our sweep and prune implementation, bucketing is performed in the remaining two dimensions to eliminate redundant overlap tests. [1] The number of buckets is chosen based on the number of objects/ROIs. [2] The total number of object/ROI overlap tests converges to a linear function when objects become "well-separated".

The communication to the user is handled through the *commander interface* [3]. A "ROI active" command is executed whenever a ROI becomes active. The ROI reported stays active until a "ROI inactive" command is executed. OpenGL has a similar light manipulation mechanism.

Theoretically the region of influence can be, for example, a sound emitter. The implementation is generic, and we are expecting to encounter surprising uses for regions of influence in the near future.

Hardware Acceleration

Should occlusion culling become available in consumer-level hardware, the part marked with a dashed line could be accelerated. As a hardware implementation will almost certainly not use silhouettes but triangle meshes as rendering primitives, the silhouette extraction would not be needed.

An important question is the feedback information from the image-space occlusion system [4]. Our occluder selection algorithm relies solely on this feedback, and if not available, some other occluder selection algorithm should be used. As the required feedback information is only a single bit per screen-space tile, and a boolean value whether the object is visible, it should be possible to implement the hardware so that the required information could be obtained. From our point of view, this is very important, since other occluder selection schemes have shown drastically worse performance figures in our experiments.

Another important consideration is the *latency*. If it takes a significant amount of processor cycles to get the feedback information from the hardware, we would need to adjust the granularity of the visibility queries to be somewhat higher. That is an awkward thing to do and basically the time consumption of the query itself and the feedback latency has to be less than when implemented with

[1] Our implementation can also be seen as performing the sweep and prune simultaneously in all three dimension by using bucket sorting.

[2] We use $N^{2/3}$ buckets, where N is the number of visible objects and ROIs.

[3] see "Programmer's Guide" for explanation of the Commander class

[4] See 3.3.2 for details of our occluder selection algorithm and the feedback information required.

software. HP Visualize-fx workstations have occlusion culling in the hardware, but they can only issue 1000-6000 queries per second, which is considerably less than what we are able to handle with our software. As a conclusion we must say that the latency of a hardware implementation must be low.

4.3 Visibility Pipeline

The visibility determination pipeline is divided roughly into two categories: view frustum culling and occlusion culling. View frustum culling is managed by the spatial database and a new view frustum is created every time we pass through a portal. The resulting view frustum is the intersection of the portal chain. Basically the same pipeline is executed for objects and for the nodes of the spatial database.

For each object and each node of the spatial database:

1. If VPT is inside the view frustum → test occlusion.
2. If AABB intersects the view frustum → test occlusion.
3. If convex hull intersects the view frustum → test occlusion.

For each object and each node inside the view frustum:

1. If VPT is visible → visible.
2. If axis-aligned rectangle is hidden → hidden.
3. If silhouette of OBB is hidden → hidden.
4. Flush the write queue, if expected to be useful. Re-execute tests 2 & 3.

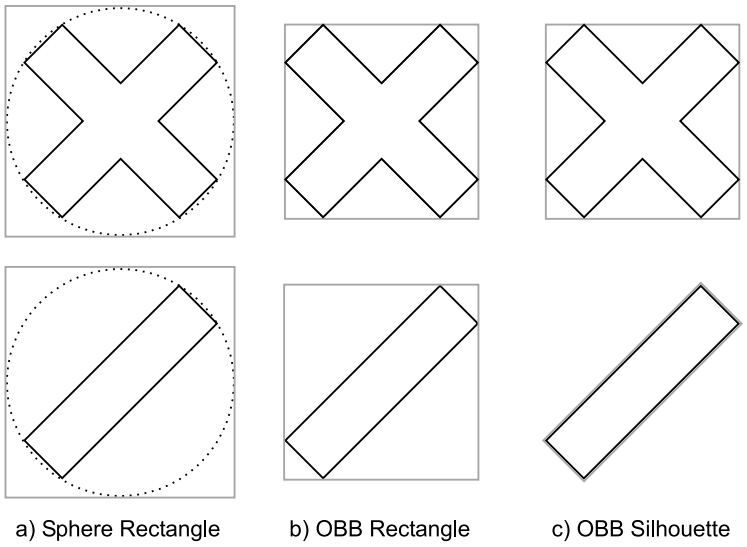

Figure 4.2 Different coverage estimation schemes. An axis-aligned rectangle constructed from a bounding sphere approximates most objects rather poorly. An axis-aligned rectangle constructed from an OBB is a suitable presentation for the upper object but very poor for the lower one. A silhouette constructed from OBB is a more expensive test primitive, but provides good coverage estimations for a wide variety of objects.

4.4 Accuracy and Robustness Considerations

In this section we first explain several accuracy issues of Umbra. After that we list known robustness issues and problem scenarios.

4.4.1 Accuracy

Granularity

Umbra performs culling at per-object precision. Polygon-precision rejection is clearly out of the question, primarily because it would be too slow and altering the model description causes bandwidth problems.[5] Consequently Umbra cannot occlude parts of a model and therefore spatially large models should be splitted into several sub-models.

Coverage

Stored coverage information is pixel-exact by default. Optionally sampling density can be increased or decreased by setting the image-space scaling to something else than 1.0. An object's screen coverage is first estimated with an axis-aligned rectangle and then with a silhouette of the OBB. Figure 4.2 illustrates occlusion test accuracies obtained by using a) rectangle of a bounding sphere, b) rectangle of an OBB and c) silhouette of an OBB. From the figure it becomes prominent that

[5] If even a single triangle is removed, the model has to be re-uploaded to the graphics hardware.

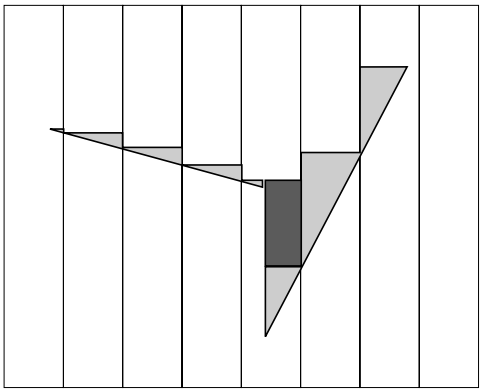

Figure 4.3 Effects caused by a low depth buffer resolution. Occlusion volume losses are marked with light gray and the area marked with dark gray is caused by several primitives getting mapped into the same 8x8 block and the furthest value getting selected.

the upper shape is relatively well presented with all three approximations, but the lower shape is captured well only by the OBB silhouette. As shapes such as the lower one are not rare, we are first using a rectangle, and only if the test fails, do we use the OBB silhouette.

Depth

Umbra stores a single depth value for each 8x8 block of the image-space resolution. Such low resolution depth values are useful only if the *greatest* depth of the corresponding 64 pixels is stored. This causes depth representation occlusion volume loss. Figure 4.3 illustrates occlusion volume losses (light grey) caused by modest and huge depth slopes. The area marked with darker gray is caused by several values mapping into the same 8x8 block and the greatest value getting selected. These effect have been found rather insignificant in actual scenes both by Zhang [121] and by us in our empirical experiments. The primary reason for this is the accurate coverage information.

Primarily due to performance reasons, the depth values used both in occlusion tests and occlusion writes are usually conservative. Figure 4.4 illustrates depth values obtained from $a)$ the nearest and the farthest points of an OBB and b) interpolated depth values of an OBB. As can be seem from the figure, in a) the depth values fail to separate objects A and B, and therefore the object B is determined visible. On the contrary, in b) the more accurate depth values succeed in separating the objects and therefore B is determined hidden. The first version of Umbra uses approach a) for objects and approach b) for spatial database nodes. Future versions are expected to use approach b) for both objects and nodes in order to further improve the accuracy.

4.4.2 Robustness

All image-space algorithms are numerically stable as they use integer math. Silhouette extraction is considered stable in the sense that degeneracies may produce non-optimal, but still closed silhouettes. We use OBBs internally and the OBB construction code itself is prone to certain degeneracies and cannot therefore be considered robust. If the OBB creation fails, we use AABB instead. Portal traversal is prone to mathematical inaccuracies when the portal chain is very long. This effect cannot fully be eliminated, but we expect no “feasible” cases to show visible artifacts. Should the problem

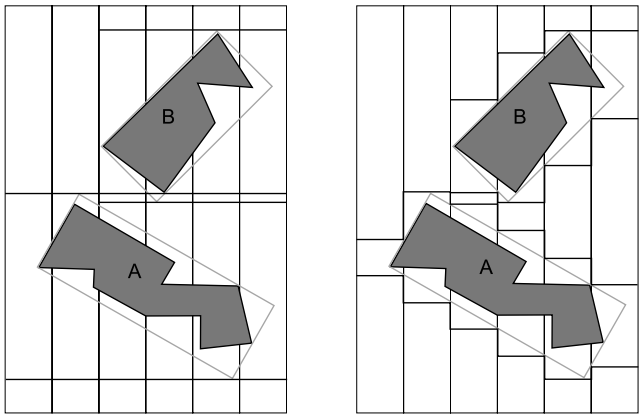

Figure 4.4 Different depth estimation schemes. If the depth values are obtained from the nearest and farthest points of the OBBs (a), object A fails to hide object B. If the depth values are obtained from interpolated depth values of the OBBs (b), object B is hidden.

become visible, double precision matrices will be used internally. Infinitesimal numerical values are explicitly flushed to zero in order to avoid underflows and inaccurate results.

4.5 Data Organization and Traversal

In this section we briefly list the primary concepts that are related to data organization and traversal.

Traversal

Objects and *cameras* are assigned into *cells*. The traversal starts from the cell into which the camera is assigned. The database is traversed hierarchically, and whenever a portal is found visible, a new view frustum is created and the target cell is processed. The traversal proceeds in depth-first order so that the current cell is always processed fully before continuing to the next cell.

Cells

Cells are object containers. A typical application only defines a single cell that contains all objects. Each cell has its own local coordinate system and a spatial database. Certain types of applications [6] divide the scene into several cells that are connected through *portals*.

Portals

We perform portal traversal computations in 3D [7], in the coordinate system of the cell. Image-space tests are supported in the form of scissoring [82]. We support both physical [8] and virtual [9] portals. Portals are treated the same way as other objects and therefore our occluder selection algorithm automatically takes portals into account.

Objects

Objects are nodes of the scene graph that are organized into the spatial databases.

Models

An object can have separate occlusion test and an occlusion write *models*. The test model is used when the object is tested for occlusion. In many cases a simple bounding volume is a sufficient representation. The write model is used when the object is being used as an occluder. Typically the write model is either the original model or simplified [10] geometry. The same model can be assigned to any number of objects.

[6] Mostly architectural scenes
[7] Luebke and Georges [72] performed portal intersection calculations in screen-space.
[8] Continuous in space.
[9] Free transformation between two portals.
[10] Simplified with Occlusion-Preserving Simplification.

Visibility Parent

An object can have a *visibility parent*. If the visibility parent is determined hidden, Umbra can cull the child as well without performing more expensive occlusion tests. *On-demand loading* can be implemented using unnecessarily large *bounding volumes* that are visibility-parented to actual objects. Visibility parents can be seen as a generalized method for expressing bounding volume hierarchies.

Bounding Volumes

Whenever Umbra needs a bounding volume, it creates it internally. The interface does not define the concept of a bounding volume, but the supported model types include, for example, a sphere. Certain complex entities such as particle systems should be inserted to Umbra as bounding volumes.

4.6 Spatial Database

4.6.1 Construction

A separate spatial database is created for each cell. The database is an axis-aligned BSP tree. Thus each non-leaf cell has either 1 or 2 children[11]. Objects are originally inserted into the database based on their AABBs[12]. All database updates are performed during visibility query traversals. This means that hidden portions of the database are not updated. Objects are always inserted into the database root node and "pushed down" during the traversal. If an object is pushed down into a node that is found hidden, it will not be pushed further down until the node becomes visible. Nodes containing more than N objects (N = 12 currently) are split into two child nodes. The splitting plane is always axis-aligned, but can be in any position inside the node. The algorithm used for finding the splitting plane location is presented in[77]. The algorithm attempts to minimize the cost function

$$L.surfaceArea * L.numObjects + R.surfaceArea * R.numObjects \tag{4.1}$$

Where L and R stand for the left and right child respectively. This algorithm works very well in practice and also attempts to minimize object splits. If more than 30of the optimal subdivision, the voxel is not split. Objects are never split, but can belong to several nodes instead. A mailbox mechanism is used during the traversal to ensure that objects are accessed only once per traversal.

When an object moves, it is not deleted and re-inserted. Instead, the update is performed locally by moving the object upwards in the tree until it fits completely inside a node. The object is then potentially later pushed down by a traversal. In many cases, an object does not exit the node where it is, and thus no updates need to be performed. Objects are pushed downwards until either a leaf node is reached or the volume of the node is smaller than the volume of the object. Leaf nodes containing no objects are destroyed.

A *tabu counter* is used for node updates. This distributes the database construction work over multiple traversals and avoids sudden hits. For example, the time to create a 10,000 object database was reduced from a 3 second start-up time to 0.3 secs for each of the first ten frames.

4.6.2 Visibility Query

The query traverses the database in an approximate front-to-back order. View frustum tests are applied to all nodes. A hierarchical frustum mask (see [10]) is propagated downwards to limit the VF tests only to the planes that clip the parent node. The AABB test used is an optimized version of Greene's / Hoff's AABB[58] test using N and P points[13]. Non-leaf nodes use their full AABB, leaf nodes use *tight bounds*, i.e. the total AABB of the objects inside it. This can be a considerably smaller volume than the node itself.

An occlusion test is performed for all potentially hidden nodes. All nodes containing objects are potentially hidden. Also nodes that had only hidden children during the previous query are potentially hidden. This is the same as Bittner's "node skipping" [12] algorithm and reduces tests by approximately ˜50%. Bittner's probabilistic node skipping [12] was tried, but not found useful. Occlusion

[11] left/right children.

[12] That can be temporal bounding volumes for moving objects. Later, if an object is determined to be static, a more accurate triangle-level intersection test is made to place the object into the correct voxels.

[13] $d.x * ||p.x|| + d.y * ||p.y|| + d.z * ||p.z|| + m.x * p.x + m.y * p.y + m.z * p.z + p.w \leq 0$; where p = plane equation of the clip plane, m = center of the bounding box and d = half-diagonal of the bounding-box (all components positive)

tests are performed using silhouettes of the visible faces of the node. A single face is tested at a time. If a node is found visible, all objects in it are VF tested and occlusion tested. The VF test uses the frustum mask of the node. Objects inside a node are locally sorted (quicksort) to front → back order to improve occlusion tests. Visible point tracking (page 54) is used in node occlusion tests, object VF tests and object occlusion tests. This reduces the amount of work considerably and is the most important optimization of the database. Object VF testing uses initially bounding spheres, then visible point tracking, then a more exact convex hull vs. view frustum test [14]. Spending additional time in VF testing to get rid of objects that are "just outside the view frustum" is extremely important. In our implementation VF testing takes only ˜10% of the time used by the occlusion culling, so reducing the number of objects to be tested and subsequently applied as occluders is crucial.

4.6.3 Optimizations

Custom pool allocators are used throughout the database system to ensure data coherence and fast memory allocations/frees. Most of the optimization work is memory-related: packing data, accessing cache lines coherently, avoiding unnecessary writes into memory. The actual traversal and testing takes very little time and the system is clearly bounded by the memory bandwidth. Originally other database formats were tried, but the axis-aligned BSP trees were found to be by far the most flexible mechanism for dynamic worlds. The lazy evaluation and postponing strategies used ensure that database updates are output-sensitive.

[14] Vertices of the convex hull are stored in a random order to provide early exits.

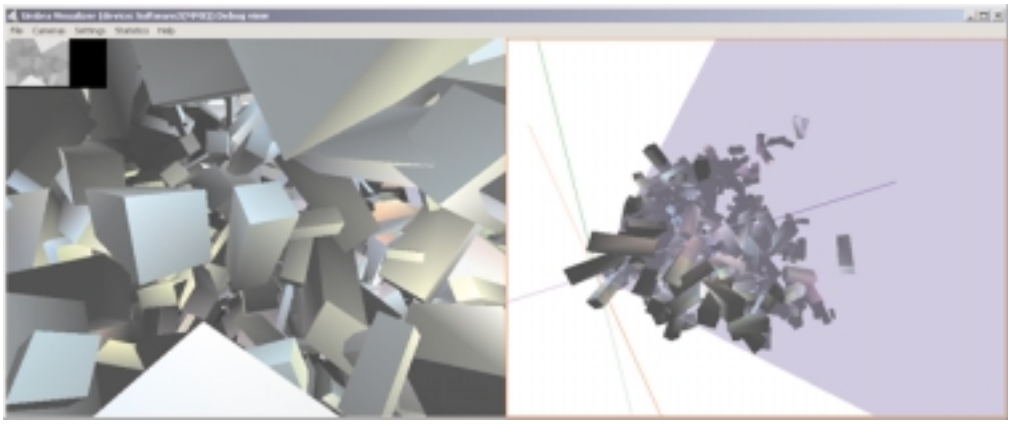

Figure 4.5 A scene with 32000 random boxes and 1000 point light sources rendered with occlusion culling.

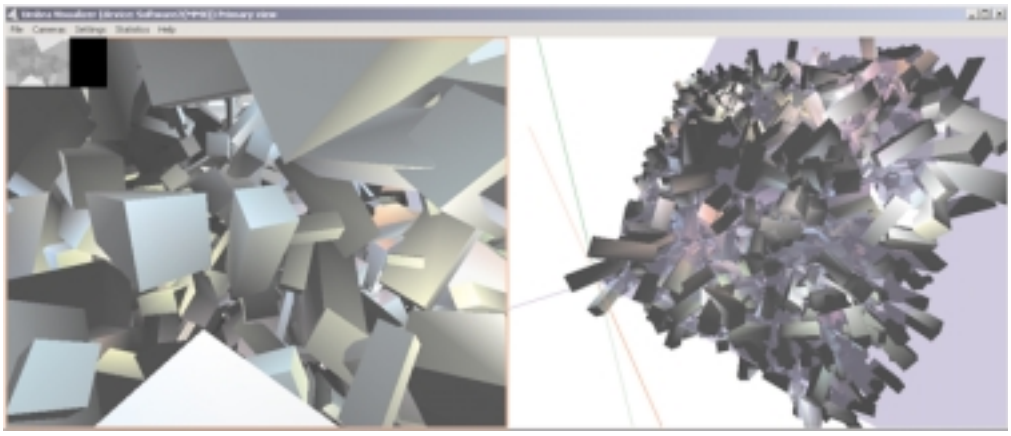

Figure 4.6 A scene with 32000 random boxes and 1000 point light sources rendered without occlusion culling (view frustum culling is enabled).

4.7 Umbra Visualizer

SurRender Umbra Visualizer is a run-time debugging and performance analysis tool. Debugging features include a wide variety of statistics (4.7.2) and visual debugging information (4.7.1). Figures 4.5,4.6,4.7 and 4.8 are screenshots taken using the Umbra Visualizer.

4.7.1 Visual Debugging Information

The visual debugging information available includes the following:

Secondary View (figure 4.5 can be used to observe the culling process from any location in the scene. This is highly useful for observing what gets processed and drawn.

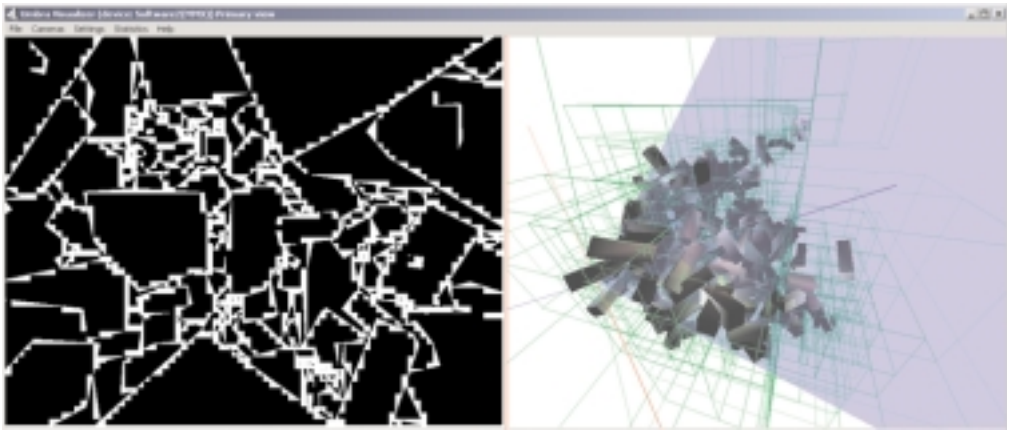

Figure 4.7 Occlusion buffer and voxels for the scene in figure 4.5

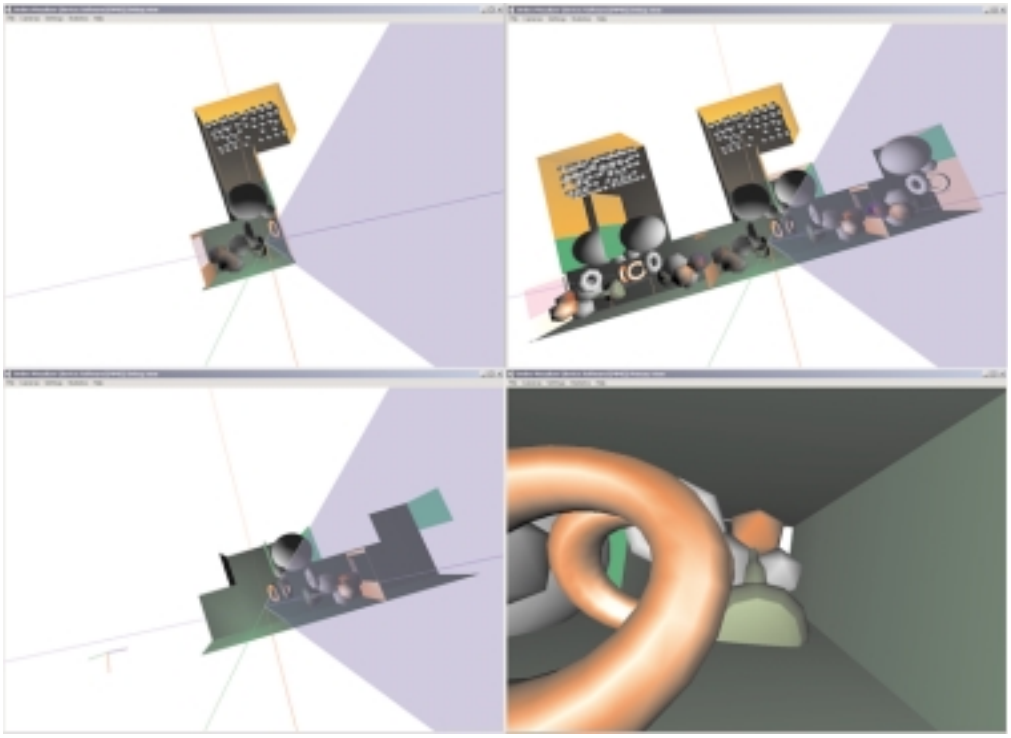

Figure 4.8 A scene with circular mirrors. The upper left figure is the scene without active mirrors. The upper right figure shows the result of enabling mirrors both on the right and on the left (the recursion depth is limited). When both view frustum culling and occlusion culling are enabled, most of the objects get culled away and the result is shown in the lower left figure. The lower right figure shows the first-person view of the final scene with two circular mirrors. Notice that only the first torus is a real object, all others are reflections.

Camera Slots: Up to 10 configurations can be stored/loaded in order to ease run-time debugging.

Crash Recovery: Umbra Visualizer can store all parameters to a disk every frame in order to trap severe bugs that cause a crash. The crash situation can thus be quickly located and consequently debugged.

Hierarchical View Frustum Culling can be disabled to simulate a basic rendering engine.

Occlusion Frustum Culling can be disabled to test the performance of the scene with and without occlusion culling.

Traversed Database Voxels can be visualized (figure 4.7) to get precise information about the traversal depth. If it seems that a traversed voxel should be occluded, the reason is an unacceptable amount of occlusion volume loss. Using this debug information we observed that depth testing the voxels with a single depth value is not accurate enough, and reports a considerable amount of voxels as visible, even though a more accurate depth test indicates that they are occluded.

Occluder Silhouettes can be visualized. A different color is used for the contributing and non-contributing silhouettes. A large amount of non-contributing silhouettes indicates that the coverage tests should have been made using a higher precision. If the same non-contributing objects are continuously being selected as occluders, there might be a bug in the occluder selection algorithm, as such objects should receive no benefits.

Occlusion Test Rectangles can be visualized to observe how conservative the axis-aligned rectangles actually are.

Occlusion Buffer can be shown. The options available are the full-resolution Coverage Buffer (figure 4.7), the Full Blocks Buffer and the Depth Estimation Buffer (figure 4.5).

Virtual Portals can be disabled in order to disable all mirrors and other special effects.

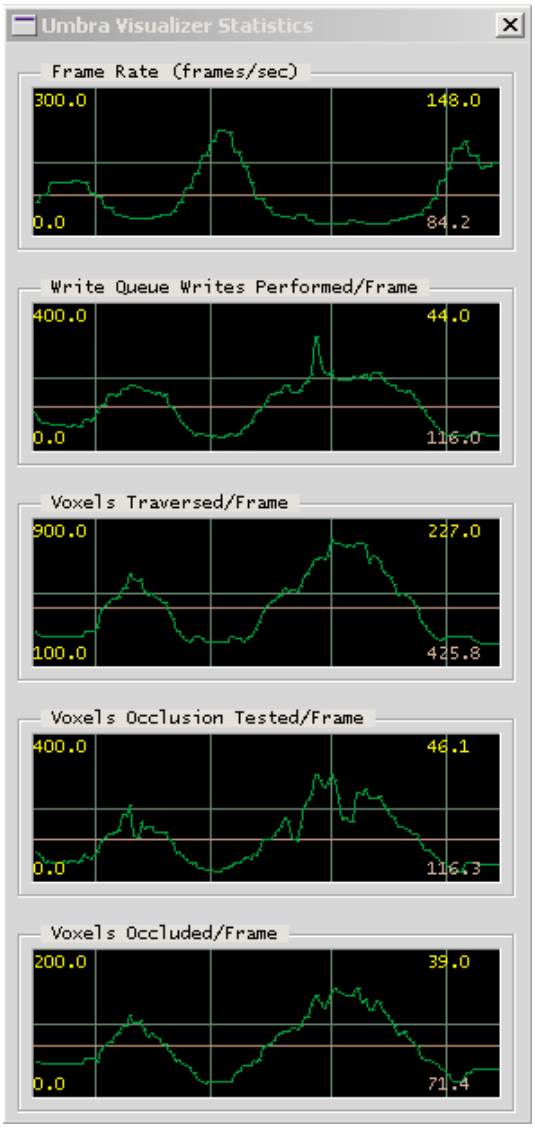

Figure 4.9 Example of the statistics output from Umbra Visualizer

Number of silhouettes created and inserted into the cache
Number of silhouettes removed from the cache
Number of silhouette queries
Number of silhouette query inner iterations
Number of silhouette cache hits (misses = queries−hits)
Number of bytes used by the silhouette cache currently

Figure 4.10 Available silhouette cache statistics.

Write queue point visibility queries
Write queue silhouette visibility queries
Write queue object visibility queries
Writes submitted to write queue
Writes submitted to occlusion buffer
Writes postponed by the write queue (bad Z)
Insufficient allocation space in write queue (expensive)
Insufficient allocation space in write queue (even more expensive)
Depth buffer writes performed by write queue
Depth buffer clears performed
Number of write queue flushes performed
Number of buckets traversed in flush work

Figure 4.11 Available write queue statistics.

4.7.2 Statistics

SurRender Umbra maintains approximately 80 different run-time statistics. Umbra Visualizer can show any combination of the statistics available in the fashion presented in figure 4.9. Additionally derivatives such as ratios between two statistics are available. More expensive statistics such as conservativity (section 2.4) require additional processing, and are optionally calculated by the Umbra Visualizer, not inside the library.

Figures 4.10, 4.11, 4.12, 4.13, 4.14, 4.15 and 4.16 contain the list of currently available statistics.

Number of hidden occluders used
Number of front clipping occluder tests
Number of front clipping occluders used
Number of points for which a full-precision query was performed
Number of blocks for which a full-precision query was performed
Number of points for which a full-precision query was useful
Number of blocks for which a full-precision query was useful
Number of times occlusion buffer buckets were cleared
Number of times occlusion buffer buckets were processed

Figure 4.12 Available occluder statistics.

Number of top level traverse calls
Number of nodes created
Number of nodes removed
Number of nodes dirty updated
Number of nodes traversed
Number of leaf nodes traversed
Number of nodes node-skipped during traversal,
Number of nodes view frustum tested
Number of nodes view frustum culled by primary test
Number of nodes view frustum culled by secondary test
Number of nodes occlusion tested
Number of nodes occlusion culled
Number of nodes visible after VF/occlusion culling
Number of nodes split
Number of times a new root node has been created (i.e. upward collapses)
Number of front clipping nodes traversed

Figure 4.13 Available database statistics.

Number of times object's visibility status has changed from visible to hidden or vice versa
Number of objects inserted into the database
Number of objects removed from the database
Number of object update calls
Number of times object update calls really caused work
Number of times an object instance has been touched
Number of objects traversed
Number of objects view frustum tested
Number of objects view frustum culled
Number of objects view frustum culled by exact tests
Number of objects view frustum culled by exact tests
Number of objects determined visible
Number of objects culled by relation to visibility parents
Number of objects whose occlusion test was skipped
Number of objects occlusion culled
Number of objects occlusion culled
Number of times a new "visible point" has been computed
Number of instances inserted
Number of instances removed
Number of instances moved (i.e. skipped a removal/insertion pair)

Figure 4.14 Available object-related statistics

Number of regions of influence found visible
Number of region of influence state changes (active $\leftrightarrow$ inactive)
Number of ROI vs. object overlap tests performed
Number of ROI vs. object overlaps

Figure 4.15 Available ROI statistics.

Number of bytes used by the silhouette cache currently
Number of times object*camera matrices have been updated
Number of times sphere bounding rectangles queried internally
Number of times `Umbra::Camera::resolveVisibility()` has been called
Number of times camera has performed visibility callbacks
Number of times a HDEB level has been updated
Time in seconds since last reset (timer accuracy is platform-dependent)
Conservativity (Visualizer only)

Figure 4.16 Available miscellaneous statistics.

4.8 Portability

SurRender Umbra does not use any rendering hardware and is not dependent on a specific platform or an operating system. Consult the “Programmer’s Guide”-part of this book for a detailed description of portability issues.

Our implementation is written in C++, but the interface can be easily wrapped to support a number of languages, including C, Java and Python.

Occlusion Buffer and Hierarchical Depth Estimation Buffer are the only algorithms prone to little/big endian issues. The issue is solved with a `SELECT_ENDIAN(little_expr,big_expr)` macro that executes the correct expression based on a global setting.

Some of the data structures use 64-bit elements. We have implemented a `QWORD` class that emulates a subset of 64-bit arithmetics on 32-bit processors. The only operations we need are logical operations and the pre-increment (`++`) operator. These operations can be emulated very efficiently without conditional code.

Chapter 5

Results and Conclusions

5.1 Test Results

The results presented in most of the occlusion culling literature have very little practical value. As there are *no* standard test scenes, “Authors report over 90% culling ratios” - style results are not very useful.

Conservativity (see section 2.4) is a useful measurement for estimating how conservative the algorithm is. Additionally some kind of measurement is needed for estimating how “difficult” the scene is.

Due to the above reasons we have no intention to release specific numbers. Additionally the end result, SurRender Umbra, is a commercial library, and stating anything like “culls 75% of the hidden geometry in typical scenes” would not be very wise. Every scene is different and there is no such thing as a typical scene. We proceed by examining a few scenarios and analyzing their results.

A Large Scene with Simple Models

In order to simulate an urban scene, we have run tests with a scene that consists of 30.000 - 100.000 randomly oriented and scaled boxes and 0 - 1000 local light sources (see figures 4.5 and 4.6). The optimal occluder count varies from zero to hundreds of objects depending on the camera transformation.

When starting from the center of the scene with 30.000 objects and 1000 lights (figure 4.6), we had to wait approximately 5 minutes to see the first frame, if occlusion culling was turned off. The first frame took approximately 0.2 seconds to render when occlusion culling was enabled. No preprocessing stage was involved. Figure 4.9 shows the frame rate when occlusion culling was enabled and actual rendering was disabled. Enabling the rendering slows down the frame rate approximately from $30 \rightarrow 20$ fps. Increasing the object count to 100.000 while keeping the same object density has no effect on the frame rate, if occlusion culling is used. If the density of the scene is decreased, and consequently we see much further, the frame rate decreases expectedly. According to our tests, the frame rate follows the size of the output set, not the size of the input. Average conservativity value is slightly below 2.0.

A Large Scene with Complex Models

Replacing the boxes of the previous scene with teapots (a few thousand triangles each) and other equally complex objects pushes the total polygon count to over 100 million triangles. Without stating exact numbers, we can say that we have observed Umbra's CPU time consumption growing slower than rendering engine's[1]. When rendering is disabled, Umbra is able to solve the visibility several frames per second while retaining the average conservativity value of 2.0. Rendering slows the frame rate down considerably, but that is pretty much out of our hands, since the conservativity value is rather good. There is simply too many polygons visible on the screen, and simplification or impostors would be needed for interactive navigation. We have not tried to turn off occlusion culling, as the result would not be seen anytime soon.

A Large Scene with Very Complex Models

We are currently using temporally coherent silhouettes, but have not implemented an output-sensitive extraction algorithm. Very complex models would suffer from substantial extraction penalty, and the guideline is to split complex models into sub-models and to use *visibility parents*[2] in this sub-model hierarchy. We have not performed tests with very complex models.

A Simple Scene with Nothing to Occlude

Umbra is constantly adapting to its input data, but does occasionally try to use objects as occluders even when it knows there is hardly a chance to succeed. The performance penalty in simple scenes with nothing to occlude has been measured to be a few percents of the CPU time.

Portal Sequences

We have experimented with physical and virtual portals. Figure 4.8 shows our test scene that contains two cells connected by a physical portal, and two mirrors on opposite walls[3]. If virtual portals are disabled in the Visualizer, no reflections are generated. As the mirrors see each other in a circular fashion, infinite recursion follows, unless addressed properly. Umbra uses both maximum recursion depth and importance decay terms to terminate recursions. Objects can cull the reflections in the mirrors. Portals are also occlusion culled. The light sources are correctly reflected by the mirrors and properly culled. Stencil models were assigned to some or all of the portals to test functionality of the stencil buffer. The rendering speed was increased by occlusion culling whenever there was something to occlude, otherwise the frame rate was practically intact.

[1] OpenGL and NVIDIA GeForce256

[2] If an object's visibility parent is hidden, Umbra can cull the child without additional occlusion tests.

[3] Application code does not need to consider mirrors, it is purely an internal issue.

5.2 Conclusions

We have presented a framework for conservative visibility determination in dynamic scenes. Our framework is able to find visible geometry by logarithmic search and is therefore output-sensitive. The framework was designed so that no platform- or hardware-specific features were used, and therefore we are able to use the library on basically any platform. We have been able to show significant acceleration of rendering of both relatively simple and very complex scenes.

To alleviate the need for pre-processing, we developed several new algorithms. Our occluder selection algorithm is adaptive, and estimates the expected occlusion powers of objects at run-time. The spatial subdivision scheme used adapts dynamically to changes in the scene structure. It internally classifies objects to be either static or dynamic and selects optimal execution paths accordingly.

Our variation of Hierarchical Occlusion Maps is incremental and does not require hardware acceleration. The framework has feedback paths between every algorithm, and most of the algorithms used perform dynamic adaptation to the input data. Even though the adaptation processes are generally very simple, the framework “learns” to process each scene in a slightly different way.

Future extensions, such as image-based rendering, have been taken into account when developing the framework.

Arbitrarily complex scenes pose difficult challenges to real-time visualization. In order to render such scenes at interactive frame rates, both conservative visibility determination algorithms and non-conservative methods, such as impostors and model simplification, are required. Although output-sensitivity can be reached in the visualization process, the challenging problems of avoiding physics, collision detection, AI calculations etc. in the hidden regions of the scene remain.

Chapter 6

Future Work

6.1 Algorithms

Umbra has a very solid image-space solver, but does not utilize temporal coherence maximally. Every known object-space algorithm is too limited for general purpose use. The concept of virtual occluders is very promising, and should an efficient algorithm to construct them be published, it would certainly be added to Umbra . The current implementation is ready to use virtual occluders as soon as an algorithm for proving region-to-virtual occluder visibility is found (see section 3.13).

6.2 Pre-Processing Tools

Occlusion-Preserving Simplification

Not a single general purpose algorithm has been proposed for Occlusion-Preserving Simplification (OPS). Should such an algorithm be published, the occlusion write models used in Umbra could be replaced with simplified versions.

Automatic Portal Generation

Being able to automatically create cell-portal subdivision of scenes would extend the usability of portals. Currently only custom modeling software have cells and portals as built-in primitives. Being able to automatically create such structures from arbitrary models would result in substantial savings of time and money in the games industry. A special-case solution for axis-aligned scenes has been proposed by Teller [107].

Alpha-Meshing

Umbra cannot handle alpha maps and automatic alpha-meshing code would simplify their use.

6.3 Accelerating Shadow Generation

Shadow generation is a problem tightly connected to visibility determination. Umbra can be used to accelerate the shadow generation by tracking the geometry seen by the light sources. Since most light sources are static, unlike cameras, a specialized version that only updates the changed portions of the coverage buffer could be used. Furthermore, stencil shadows work with object silhouettes. Since we have already optimized the silhouette extraction, the same algorithms could be utilized to accelerate the generation of shadow volumes.

6.4 On-Demand Loading

Modern consoles have a very high polygon drawing capability compared to the amount of memory available[1]. This imbalance forces the scenes to be unreasonably small, unless the data can be loaded from the DVD drive when needed. Umbra tracks the potentially visible set of objects at any given time, so it could be used to predict which objects are likely to be needed in the near future. Similarly, it would be able to predict which objects are unlikely to be needed for a while, and to give hints to swap these out in order to free resources.

In a similar fashion, the clients of massive online games could only request the data actually needed from the server. Such an optimization would result in remarkable savings in the bandwidth required [36].

6.5 Image-Based Rendering

Umbra could be extended to create and handle bitmapped impostors. A large percentage of the code and data structures needed has already been implemented. The current interface was designed with impostors in mind, so they will fit into the framework very well. The first step is most likely generic reflections and refractions that require rendering into an off-screen surface. The actual rendering will be done outside the library, just as the rendering of the primary view. Umbra would be used to handle distortion calculations, redraw commands and texture management commands. Distortion calculations are in fact very similar to the current silhouette cache distortion calculations. Umbra could then transparently replace distant scene geometry with bitmapped impostors for increased rendering speed. Our impostors will be mapped to the voxel faces of the spatial database, thus combining spatially close objects and avoiding depth overlap problems. Should an impostor be completely opaque, the voxel face can be automatically determined to be a virtual occluder.

6.6 Interaction Pre-Processor

Umbra already solves ROI vs. object interactions and the logic could be easily extended to handle object-object interactions. Internally, we know whether an object is *currently static* and can thus eliminate the interaction tests between static objects. To maintain output-sensitivity, interactions must only be processed for the visible portion of the scene. It is a challenging task to design physics and collision systems so that physics in the hidden parts of the scene can be postponed until needed.

[1] Especially Sony PlayStation2®

On the other hand, if a character becomes visible after a long while, the exact animation frame and position may not be important.

Appendix A

Glossary

This section contains short definitions and explanations for terms and acronyms used in this manual.

AABB	Axis-Aligned Bounding Box
Accumulation Buffer	Additional high-resolution buffer for combining color buffer images. Can be used for antialiasing or effects such as motion blur or depth of field
Aggressive Culling	Culling of hidden objects as well as some visible objects (compare to conservative culling)
AI	Artificial Intelligence
API	Application Programming Interface
Area Visibility	Determination of visibility from a volume in space rather than a single point
Axis-Aligned	Something that is aligned with the three primary axis (X,Y,Z)
Back-Face Culling	Culling of polygons or objects that are completely facing away from the camera
Bounding Box	A box that fully contains an object
Bounding Sphere	A sphere that fully contains an object
Bounding Volume	Any volume that fully contains an object
Bounding Volume Hierarchy	A set of bounding volumes organized hierarchically and used to improve various tests
BSP	Binary Space Partitioning
BSP Tree	A tree structure where each node partitions a set of objects or polygons into two halves
BVH	Bounding Volume Hierarchy

Cell	A part of the world connected to other cells by portals
Conservative Culling	Culling of (most of the) objects that do not contribute to the output image
Contribution Culling	Culling of objects that have little significance to the output image
Convex Hull	Vertices on the convex boundary of the object
Culling	Removal of objects that do not contribute to the output image
CW	Clockwise
CCW	Counter-clockwise
Depth Buffer	A data structure used for pixel-level hidden-surface determination
DHS	Definitely Hidden Set
DLL	Dynamic Link Library
DOF	Direction Of Flight (or View Plane Normal)
Floating Portal	A virtual portal that is not on a cell boundary (such as a hand mirror)
FPS	Frames Per Second
Frame Coherence	Coherence between subsequent frames
Frame Rate	Number of frames rendered per second
Free Clipping Plane	Any clipping plane beyond the initial six planes forming the view frustum
HOM	Hierarchical Occlusion Map
Hz	Hertz (unit of frequency). Measures number of times something occurs per second
HZB	Hierarchical Z-Buffer
HWTL	HardWare Transform and Lighting
IBR	Image-Based Rendering
Image-Based Rendering	Creating the output image from previously rendered images
LOD	Level Of Detail
Motion Prediction	Heuristic prediction of future locations and orientations of objects
OBB	Oriented Bounding Box, i.e. a bounding box where the axes are not aligned to the three primary axes
Occludee	An object obstructed from the view by other objects
Occluder	An object used for obstructing geometry from the view

Occluder Fusion	A property of certain occlusion culling algorithms of merging the umbrae of multiple occluders
Occlusion Culling	Culling of objects that are hidden by other objects
Octree	A spatial data structure where each node has eight child nodes
OPS	Occlusion-Preserving Simplification
Penumbra	The partial shadow region of an area light source
Physical Portal	A real-life portal where no warping of light rays is performed
Point Visibility	Visibility from a single point in space
Portal	A connection (either physical or virtual) between two cells
Potentially Visible Set	List of all visible and some hidden objects
PVS	Potentially Visible Set
Recycler	A memory-management structure for reusing allocated memory used to reduce memory allocations/frees
Region Of Influence	The volumetric region in space where an object (such as a light source) can influence other objects
ROI	Region Of Influence
Screen-Size Culling	Removal of objects that have an axis-aligned screen projection smaller than some specified size
Spatial Database	A data structure where objects are organized based on their spatial extents
Spatial Index	Spatial Database
Stencil Buffer	A buffer used for enabling or disabling frame buffer writes on a per-pixel level
Tabu Counter	A counter used to prevent some action for a period of time
TBV	Temporal Bounding Volume
Temporal Bounding Volume	A bounding volume for an object that is valid for a certain period of time
Temporal Coherence	Coherence in the objects' positions and orientations over long periods of time
Test Model	A model used for determining the visibility of an object
Umbra	Portions of the scene that are fully hidden from a light source
Vertex Indices	Indices to a vertex position array. Used for describing triangle mesh data. Each triangle is specified using three indices
VF	View Frustum
View Frustum	Portion of the scene visible from a camera
View Frustum Culling	Culling of objects that fall completely outside the view frustum

Virtual Occluder	A hidden region of space that can be used as an occluder
Virtual Portal	Non-physical link between two portals
Warp Matrix	A matrix used for bending light rays when they travel through a virtual portal
Winding	The order in which a polygon's vertices are defined. Used to indicate which side is considered the front-side of the polygon
Write Model	A model that participates in the occlusion culling process as an occluder
Z-Buffer	A depth buffer where the raster-space Z-values of pixels are used as depth values

Appendix B

Technical Information

Platforms supported

- IBM PC and compatibles (Pentium or newer) running on Win32
- Apple Macintosh (PowerPC) running on MacOS 8.6 or newer
- all other platforms available on request

Minimum Platform Requirements

Umbra can be ported to any platform with the following minimum requirements:

- 32 or 64 bit integer instructions
- hardware single-precision floating-point unit
- double-precision floating-point unit preferable
- in order to compile the examples, OpenGL 1.1 and GLUT 3.x are needed

Compilers Supported

The Umbra library is written in fully portable C++, where many of the newer C++ constructs have been avoided. The code handles endianess issues properly and scales to 64-bit architectures.

- Microsoft Visual C++ 5/6
- Intel C++ W1.2 or newer
- GCC 2.95.2 or newer
- Borland C++ Builder 4.0

- Metrowerks CodeWarrior C/C++ version 2.2 or newer
- library files for other compilers available on request
- C, Java and Python wrappers available

Minimum Compiler Requirements

Umbra can be ported to any modern C++ compiler satisfying the following minimum requirements:

- support for namespaces
- support for basic templates. No member function templates needed.
- no support for exception handling needed
- no support for RTTI needed
- neither STD nor STL needed

API

- all classes area inside a common namespace - Umbra does not conflict with other libraries
- public interface consists of 9 header files, about a dozen classes, and about one hundred public member functions
- no global functions, variables or enumerations
- all API functions are fully documented
- consistent naming conventions for enumerations, classes and member functions
- most of the complicated concepts are internal and not exposed in the API
- no implementation details exposed in the API

Debugging

- separate debug build containing assertions for `NULL` pointers, invalid data and other user errors
- symbolic debugger info available in debug build
- most internal components perform self-tests in debug build
- system tested with a number of test applications and error-detection tools such as BoundsChecker
- entire source code compiles with zero warnings at maximum warning levels on all supported compilers
- Umbra Visualizer tool with external correctness tests of the system

Profiling

- API contains functions for querying a large amount of internal statistics
- Umbra Visualizer tool with statistics graphs

Resources Used

- no pre-processing of data required
- library object file is ˜150 kB
- no disk space needed directly as the library does not perform any file I/O
- memory consumption is linear to input data
- small consumption of CPU resources

Input Data

- models used as occluders are defined as triangle/vertex arrays
- models used as occludees are defined either as triangle/vertex arrays or bounding volumes
- objects can be instantiated, i.e. the same model can be shared by multiple object instances

Multi-Threading Issues

- only a single thread can access any components of the library at a time
- compiled with multi-thread version of the standard C library
- does not use non-thread safe C library functions such as `rand()`

x86

- detects and utilizes MMX instructions if available; speeds up occlusion buffering
- detects and utilizes SIMD FP instruction sets if available; speeds up floating-point processing

Index

Bibliography

[1] http://www.minolta3d.com.

[2] John M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. Technical report 90-027, Department of Computer Science, University of North Carolina at Chapel Hill, July 1990.

[3] John M. Airey, John H. Rohlf, and Fredrick P. Brooks Jr. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):141–150, March 1990.

[4] John Alex and Seth Teller. Immediate-mode ray casting. Technical Report 784, MIT LCS, 1999. *http://graphics.lcs.mit.edu/˜seth/pubs/MIT-LCS-TR-784.ps.gz*.

[5] Daniel G. Aliaga. Visualization of complex models using dynamic texture-based simplification. In *IEEE Visualization '96*, October 1996.

[6] Daniel G. Aliaga and Anselmo A. Lastra. Architectural walkthroughs using portal textures. In *IEEE Visualization '97*, October 1996.

[7] Carlos Andújar, Carlos Saona-Vázquez, Isabel Navazo, and Pere Brunet. Integrating occlusion culling with levels of detail through hardly-visible sets. In *Proceedings of Eurographics '2000*, 2000.

[8] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan - Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[9] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS Conference Proceedings*, volume 32, pages 37–45, 1968.

[10] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithyms for bounding boxes. Technical report, Department of Computer Engineering, Chalmers University of Techonology, Sweden, 2000. Accepted for publication in Journals of Graphics Tools.

[11] Fausto Bernardini, Jihad El-Sana, and James T. Klosowski. Directional discretized occluders for accelerated occlusion culling. In *Proceedings of Eurographics '2000*, 2000.

[12] J. Bittner. Global visibility computations. Master's thesis, Department of Computers, Czech Technical University, 1997.

[13] Karsten Bormann. An adaptive occlusion culling algorithm for use in large ves. In *Proceedings of the IEEE Virtual Reality 2000 Conference*, 2000.

[14] Loren Carpenter. The a-buffer, an antialiased hidden surface method. In *Computer Graphics (SIGGRAPH '84 Poceedings)*, volume 18, pages 103–108, July 1984.

[15] Edwin Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[16] Edwin Catmull. A hidden-surface algorithm with anti-aliasing. *Computer Graphics (Proceedings of SIGGRAPH 78)*, 12(3):6–11, August 1978. Held in Atlanta, Georgia.

[17] Frédéric Cazals, George Drettakis, and Claude Puech. Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. *Computer Graphics Forum*, 14(3):371–382, August 1995. ISSN 1067-7055.

[18] Frédéric Cazals and Claude Puech. Bucket-like space partitioning data structures with applications to ray-tracing. In *Proceedings of the 13th ACM Symposium on Computational Geometry*, pages 242–248, New York, June 1997. ACM Press.

[19] Han-Ming Chen and Wen-Teng Wang. The feudal priority algorithm on hidden-surface removal. *Proceedings of SIGGRAPH 96*, pages 55–64, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[20] Lih-Shyang Chen, Gabor T. Herman, R. Anthony Reynolds, and Jayaram K. Udupa. Surface shading in cuberille environments. *IEEE Computer Graphics and Applications*, 5(15), December 1985.

[21] Milton Chen, Gordon Stoll, Homan Igehy, Kekoa Proudfoot, and Pat Hanrahan. Simple models of the impact of overlap in bucket rendering. In *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH workshop on Graphics hardware*, pages 105–112, August 1998.

[22] James H. Clark. Hierarchical geometric model for visible surface algorithms. *Communications of ACM*, 19(10):547–554, October 1976. Reprinted in [117].

[23] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *SIGGRAPH '96 Proc.*, pages 119–128, Aug. 1996. *http://www.cs.unc.edu/~geom/envelope.html*.

[24] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann Publishers, 1993.

[25] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 95–102, July 1987. Held in Anaheim, California.

[26] Satyan Coorg and Seth Teller. Temporally coherent conservative visibility. In *Proceedings of the Twelfth Annual Symposium On Computational Geometry*, pages 78–87. ACM Press, t'May 1996. Held in New York.

[27] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. *1997 Symposium on Interactive 3D Graphics*, pages 83–90, April 1997. ISBN 0-89791-884-3.

[28] Michael Cox and Narendra Bhandari. Architectural implications of hardware-accelerated bucket rendering on the pc. In *Proceedings of the 1997 SIGGRAPH/Eurographics workshop on Graphics hardware*, pages 25–34, August 1997.

[29] Franklin C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH 77)*, 11(2):242–248, July 1977. Held in San Jose, California.

[30] Frédo Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Université Grenoble I - Joseph Fourier Sciences et Géographie, July 1999.

[31] Frédo Durand, George Drettakis, and Claude Puech. The 3d visibility complex: A new approach to the problem of accurate visibility. In *Eurographics Rendering Workshop 1996*, pages 246–256. Eurographics, June 1996.

[32] Frédo Durand, George Drettakis, and Claude Puech. 3d visibility made visibly simple an introduction to the visibility skeleton. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 89–100, 1997. *http://www.acm.org/pubs/articles/proceedings/graph/258734/p89-durand/p89-durand.pdf*.

[33] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. *Proceedings of SIGGRAPH 97*, pages 89–100, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[34] Jihad A. El-Sana, Elvir Azanli, and Amitabh Varshney. Skip strips: Maintaining triangle strips for view-dependent rendering. *IEEE Visualization '99*, pages 131–138, October 1999. ISBN 0-7803-5897-X. Held in San Francisco, California.

[35] Carl Ericsson and Dinesh Manocha. Gaps: General and automatic polygonal simplification. Technical Report 98-033, UNC Chapel Hill Computer Science, 1998. Appeared in Proc. of ACM Symposium on Interactive 3D Graphics, 1999, Atlanta.

[36] Chris Faisstnauer, Dieter Schmalstieg, and Werner Purgathofer. Scheduling for very large virtual environments and networked games using visibility and priorities. Technical Report TR-186-2-00-09, Institute of Computer Graphics, Vienna University of Technology, A-1040 Karlsplatz 13/186/2, April 2000.

[37] E. Fiume, A. Fournier, and L. Rudolph. A parallel scan conversion algorithm with anti-aliasing for a general purpose ultracomputer. *Computer Graphics (Proceedings of SIGGRAPH 83)*, 17(3):141–150, July 1983. Held in Detroit, Michigan.

[38] Foley, van Dam, Feiner, and Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990.

[39] W. R. Franklin. A linear time exact hidden surface algorithm. *Computer Graphics (Proceedings of SIGGRAPH 80)*, 14(3):117–123, July 1980. Held in Seattle, Washington.

[40] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. Predetermining visibility priority in 3-d scenes (preliminary report). *Computer Graphics*, 13(2), August 1979.

[41] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3), July 1980.

[42] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *International Conference on Computer Graphics and Interactive Techniques Conference Proceedings on Computer Graphics*, pages 79–88, Boston, MA, USA, July 1989.

[43] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. *Proceedings of SIGGRAPH 97*, pages 209–216, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[44] C. Georges. Obscuration culling on parallel graphics architectures. Technical Report 95–017, Department of Computer Science, UNC-Chapel Hill, 1995.

[45] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 1984.

[46] Craig Gotsman, Oded Sudarsky, and Jeffrey A. Fayman. Optimized occlusion culling using five-dimensional subdivision. *Computers & Graphics*, 23(5):645–654, October 1999. ISSN 0097-8493.

[47] Charles W. Grant. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. *Computer Graphics*, 19(3), July 1985.

[48] Charles W. Grant. *Visibility Algorithms in Image Synthesis*. PhD thesis, University of California, 1992.

[49] Ned Greene. *Hierarchical Rendering of Complex Environments*. PhD thesis, University of California at Santa Cruz, June 1995.

[50] Ned Greene. Hierarchical polygon tiling with coverage masks. *Proceedings of SIGGRAPH 96*, pages 65–74, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[51] Ned Greene and Michael Kass. Hierarchical z-buffer visibility. *Proceedings of SIGGRAPH 93*, pages 231–240, 1993. ISBN 0-201-58889-7. Held in Anaheim, California.

[52] Ned Greene and Michael Kass. Error-bounded antialiased rendering of complex environments. *Proceedings of SIGGRAPH 94*, pages 59–66, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[53] Eduard Gröller and Werner Purgathofer. Coherence in computer graphics. Technical Report TR-186-2-95-04, Institute of Computer Graphics, Vienna University of Technology, A-1040 Karlsplatz 13/186/2, March 1995.

[54] X. Gu, S. Gortler, H. Hoppe, L. McMillan, B. Brown, and A. Stone. Silhouette mapping. Technical Report TR-1-99, Department of Computer Science, Harvard University, March 1999.

[55] G. Hamlin, Jr., and C. W. Gear. Raster-scan hidden surface algorithms. *Computer Graphics (Proceedings of SIGGRAPH 77)*, 11(2):206–213, July 1977. Held in San Jose, California.

[56] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):119–127, July 1984. Held in Minneapolis, Minnesota.

[57] Poon Chun Ho and Wenping Wang. Occlusion culling using minimum occluder set and opacity map. In *Proceedings of the 1999 International Conference on Information Visualization*, 1999.

[58] Kenny Hoff. A faster overlap test for a plane and a bounding box. Technical report, University of North Carolina, 1996. http://www.cs.unc.edu/~hoff/research/vfculler/boxplane.html.

[59] Hugues Hoppe. Progressive meshes. *Proceedings of SIGGRAPH 96*, pages 99–108, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[60] Hugues Hoppe. View-dependent refinement of progressive meshes. *Proceedings of SIGGRAPH 97*, pages 189–198, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

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

[62] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89*, pages 164–172, June 1989.

[63] A. Johanssen and M. Carter. Clustered backface culling. *Journal of Graphics Tools*, 3(1):1–14, 1998.

[64] Norman P. Jouppi and Chun-Fa Chang. Z3: an economical hardware technique for high-quality antialiasing and transparency. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 85–93, August 1999. Held in Los Angeles, California.

[65] J. Klosowski and C. Silva. Rendering on a budget: A framework for time-critical rendering. *Proceedings of the IEEE Computer Society Visualization '99*, October 1999.

[66] J. Koenderink. What does the occluding contour tell us about solid shape. *Perception 13*, pages 321–330, 1984.

[67] Vladlen Koltun, Yiorgos Chrystanthou, and Daniel Cohen-Or. Virtual occluders: a from-region visibility techniques. In *Eurographics Rendering Workshop*, 2000.

[68] Subodh Kumar and Dinesh Manocha. Hierarchical visibility culling for spline models. In *Graphics Interface*, pages 142–150, May 1996.

[69] 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. Eurographics, Springer Wein, June 1996.

[70] Fei-Ah Law and Tiow-Seng Tan. Preprocessing occlusion for real-time selective refinement. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 47–53, 1999.

[71] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. *Proceedings of SIGGRAPH 97*, pages 199–208, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[72] David P. Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings 1995 Symposium on Interactive Computer Graphics*, pages 105–106, April 1995.

[73] P. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In P. Hanharan and J. Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 95–102. ACM SIGGRAPH, April 1995.

[74] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *CG & A*, 9(4):43–55, July 1989.

[75] D. Meagher. *The Octree Encoding Method for Efficient Solid Modeling*. PhD thesis, Electrical Engineering Department, Renssalaer Polytechnic Institute, New York, August 1982.

[76] Jim Miller. Directmodel 1.0 specification - revision c. Available online: http://www.op.dlr.de/FF-DR/dr_fs/staff/trautenberg/DirectModelspec.pdf, October 1997.

[77] Gordon Müller, , and Dieter Fellner. Hybrid scene structuring with application to ray tracing. In *Proceedings of International Conference on Visual Computing (ICVC'99)*, pages 19–26, February 1999.

[78] A.J. Myers. An efficient visible surface program. Technical Report DCR 74-00768A01, National Science Foundation, 1975.

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

[80] M. E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. In *Proceedings of the ACM 1972 National Conference*, pages 443–450, 1972. Reprinted in [117].

[81] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, July 2000.

[82] Dave Schreiner Opengl Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Addison-Wesley, December 1999.

[83] Mark Peercy, Marc Olano, John Airey, and Jeff Ungar. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, July 2000.

[84] H. Plantinga and C. Dyer. Visibility, occlusion and the aspect graph. *International Journal of Computer Vision*, 5(2):137–160, 1990.

[85] Voicu Popescu, John Eyles, Anselmo Lastra, Josh Steinhurst, Nick England, and Lars Nyland. The warpengine: An architecture for the post-polygonal age. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, July 2000.

[86] Jovan Popovic and Hugues Hoppe. Progressive simplicial complexes. *Proceedings of SIGGRAPH 97*, pages 217–224, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[87] Steven M. Rubin and J. Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics (Proceedings of SIGGRAPH 80)*, 14(3):110–116, July 1980. Held in Seattle, Washington.

[88] D. Salesin and J. Stolfi. The zz-buffer: A simple and efficient rendering algorithm with reliable antialising. In *Proceesings of the PIXIM '89 Conference*, pages 451–466. Hermes Editions, Paris, September 1989.

[89] P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe, and J. Snyder. Silhouette clipping. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, July 2000.

[90] C. Saona-Vázquez, I. Navazo, and P. Brunet. Data structures and algorithms for navigation in highly polygon-populated scenes. Technical Report Technical Report LSI-98-58R, Universitat Politècnica de Catalunya, 1998.

[91] G. Schaufler. Dynamically generated impostors. In D. W. Fellner, editor, *Modeling Virtual Worlds - Distributed Graphics*, pages 129–136. MVD'95 Workshop, November 1995.

[92] Gernot Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In Julie Dorsey and Philipp Slusallek, editors, *Rendering Techniques '97*, Eurographics, pages 151–162. Springer-Verlag Wien New York, 1997.

[93] Gernot Schaufler. Per-object image warping with layered impostors. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98*, Eurographics, pages 145–156. Springer-Verlag Wien New York, 1998.

[94] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and Francois Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, July 2000.

[95] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–236, August 1996. ISSN 1067-7055.

[96] D. Schmalstieg and R. F. Tobler. Real-time bounding box area computation. Technical Report TR-186-2-99-05, Vienna University of Technology, 1999. http://www.cg.tuwien.ac.at/research/TR/.

[97] R.A. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer generated images to visual simulation. Technical Report AFHRL-TR-69-14, U.S. Air Force Human resources Laboratory, September 1969.

[98] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Proceedings of SIGGRAPH 96*, pages 75–82, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[99] François X. Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. In *Computer Graphics Forum*, pages 207–218, 1997.

[100] Mel Slater and Yiorgos Chrysanthou. View volume culling using a probabilistic caching scheme. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 71–77, 1997.

[101] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis and Machine Vision*. International Thomson Computer Press, 1996.

[102] Oded Sudarsky and Craig Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. *Computer Graphics Forum*, 15(3):249–258, August 1996. ISSN 1067-7055.

[103] Ivan Sutherland, Robert Sproull, and Robert Schumacker. Sorting and hidden-surface problem. In *Proceedings for the AFIPS National Conference*, volume 42, 1973.

[104] Ivan Sutherland, Robert Sproull, and Robert Schumacker. A characterization of ten hidden surface algorithms. *ACM Computing Surveys*, 6, 1974.

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

[106] Seth Teller. Frustum casting for progressive, interactive rendering. Technical Report 740, MIT LCS, January 1998.

[107] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, CS Division, UC Berkeley, October 1992. Tech. Report UCB/CSD-92-708.

[108] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):61–69, July 1991. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.

[109] Jay Torborg and James T. Kajiya. Talisman: Commodity realtime 3D graphics for the PC. *Computer Graphics*, 30(Annual Conference Series):353–363, 1996.

[110] Steve Upstill. *The Renderman Companion*. Addison Wesley, Reading, MA, 1990.

[111] Michiel van de Panne and A. James Stewart. Effective compression techniques for precomputed visibility. In *Eurographics Workshop on Rendering*, pages 305–316, June 1999.

[112] John E. Warnock. A hidden surface algorithm for computer generated half-tone pictures. Technical Report RADC-TR-69-249, Dept. of CS U of Utah, June 1969. Reprinted in [38].

[113] K. Weiler and K. Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics (Proceedings of SIGGRAPH 77)*, 11(2):214–222, July 1977. Held in San Jose, California.

[114] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6), June 1980.

[115] Lance Williams. Pyramidal parametrics. *Computer Graphics (Proceedings of SIGGRAPH 83)*, 17(3):1–11, July 1983. Reprinted in [117].

[116] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 1990.

[117] Rosalee Wolfe, editor. *Seminal Graphics: Pioneering Efforts That Shaped The Field*. ACM SIGGRAPH, 1998.

[118] C. Wylie, G.W. Romney, D.C. Evans, and A.C. Erdahl. Halftone perspective drawings by computer. In *Proceedings of AFIPS Fall Joint Computer Conference*, 1967.

[119] Feng Xie and Michael Shantz. Adaptive hierarchical visibility in a tiled architecture. In *Proceedings 1999 Eurographics/SIGGRAPH workshop on Graphics hardware*, pages 75–84, 1999.

[120] Hansong Zhang. Fast incremental transformation of bounding boxes, with generalizations to regular grids like terrain and volume data. Technical report, University of North Carolina, September 1997.

[121] Hansong Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1998.

[122] Hansong Zhang. Personal communication, 2000.

[123] Hansong Zhang and Kenneth E. Hoff III. Fast backface culling using normal masks. *1997 Symposium on Interactive 3D Graphics*, pages 103–106, April 1997. ISBN 0-89791-884-3.

[124] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. *Proceedings of SIGGRAPH 97*, pages 77–88, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.