Online Occlusion Culling

Thesis Pro Gradu

Johannes Staffans
Department of Information Technologies
Faculty of Technology
Åbo Akademi
Åbo 2006

Supervisors:
Jan Westerholm
Jukka Arvo

Abstract

A computer application that renders a virtual environment, for example an architectural model for use in a walkthrough scenario, can suffer from low performance due to the complexity of the environment. Visibility algorithms are concerned with determining the part of the environment that is visible and therefore necessary to render while discarding the rest, thereby reducing the workload of the graphics hardware. One group of visibility algorithms is concerned with determining which objects are blocked from view by other objects; this is called occlusion culling. Occlusion culling can be applied to for example the interiors of buildings or the contents of containers. In this thesis, the general problem of visibility in three-dimensional scenes will be presented as well as algorithms that attempt to solve it, especially methods to perform occlusion culling and in particular algorithms that perform occlusion culling *online*, without significant precalculations. Among these, several algorithms that take advantage of a recently developed graphics hardware extension will be described in detail, implemented and their performance and complexity analyzed.

Keywords: real-time rendering, visibility, occlusion culling

Acknowledgements

Many thanks go to Jan Westerholm and Jukka Arvo for the original idea for this thesis, valuable comments and fruitful discussions. A special thanks to Ákos Balázs for making available the original implementation of the NOHC algorithm discussed in Chapter 5 - without this reference, the tests and measurements that this work relies on could not have been made.

Åbo, December 2006

Johannes Staffans

Contents

List of Figures

Chapter 1

Introduction

One of the fastest-evolving areas in computer science is the field of computer graphics and in particular three-dimensional graphics, where applications are concerned with providing a two-dimensional image of a virtual, three-dimensional object or set of objects. The process of converting a three-dimensional entity into pixels on the user's screen is called *rendering*. The objects to be rendered are often collectively called the *scene*. The term *scene complexity* is used to describe how much detail the scene contains, generally in terms of number of triangles. The complexity of a scene has virtually no upper limit; a scene can always contain more objects and objects can be made more detailed. For example, laser scans of marble statues can easily provide models with several million triangles [37].

While the processing power of graphics chips has increased at a tremendous rate over the recent years [23], the equal and often greater growth in scene complexity makes the need for algorithms that can speed up the rendering process very actual. The goal is always to provide the user with a real-time, interactive experience.

Visibility algorithms are central to real-time rendering; in virtually any scene (especially in applications such as first-person games, where the user sees the world through the eyes of a character than can move freely in a three-dimensional world) there are many objects that, from some given viewpoint, are not visible and so can be temporarily discarded. Imagine, for instance, moving through the streets of a city; whole city blocks, as well as the building interiors, are often blocked from view by the walls of the houses. Another example would be a model of a normal copy machine where the complicated machinery inside is hidden from view by panels. Clearly, rendering these invisible objects is a waste of processing power.

Normally, a computer system delegates the rendering of screen content to a special part of the computer hardware, the *graphics accelerator*. The graphics hardware contains

a Graphics Processing Unit (GPU) which is similar to the computer's Central Processing Unit (CPU) but designed with the sole purpose of rendering graphics very quickly. Simply put, an application running on the CPU decides what should be rendered and the GPU handles the task of actually displaying something on the screen.

The architecture of modern GPU's is often heavily pipelined. This means that the raw data that enters the pipeline will be sent through a number of different pipeline stages, one after another, before the image is ready to be displayed on the user's screen. If the non-visible geometry can be discarded already before entering the GPU pipeline significant speedups can be gained. Achieving better performance in this manner is the aim of the field of visibility research in computer graphics.

It has been noted that current visibility research suffers from a lack of objective comparisons of the many different existing approaches to the visibility problem [13]. The purpose of this thesis is to alleviate this situation somewhat by describing and comparing several different methods. In particular, a set of interesting algorithms will be implemented and performance benchmarks provided. The idea is to enable developers of real-time rendering software to more clearly see which approaches are worth investigating further.

The focus of this thesis will be on a family of algorithms that take advantage of a recently developed graphics hardware extension that has been specifically developed to handle visibility tests. A hardware solution tends to be faster than a similar software one because it is easier to optimize the static hardware and hence it is logical to assume that these algorithms have a certain advantage right from the start. The required hardware extension is readily available in current graphics accelerators.

The algorithms that will be compared all strive to cover the general case; there are many visibility algorithms that work well provided some special circumstances prevail, for example that the scene is an architectural model that consists of static rooms with interconnecting doors and hallways. Often involving heavy precalculation procedures, these specialised algorithms are largely omitted in favour of a set of more implementation-friendly methods that perform visibility determination *online*, that is, without significant preprocessing. For the interested reader, a good survey of recent visibility algorithm developments can be found in [13].

Before investigating the actual algorithms somefundamental concepts in computer graphics have to be introduced. Again, the research in the field is vast and the focus will be on the issues vital for understanding the later chapters.

Chapter 2 describes how a three-dimensional scene can be set up and what it consists of. Spatial data structures, used for organizing the objects present in the scene, are also

introduced. These structures are essential for any good visibility algorithm.

In Chapter 3 the terms and concepts of visibility for computer graphics applications are elaborated. The foundations on which the algorithms presented later in the thesis are built are introduced.

Chapter 4 covers some of the earlier work on *occlusion culling*. Occlusion culling involves removing objects which are invisible because they are located behind other objects (like the engine of a car being invisible because the bonnet is down) and is currently the primary area of interest for visibility researchers. In this chapter, the hardware extension mentioned earlier is also introduced.

Several algorithms that make use of the previously introduced hardware extension are introduced in Chapter 5 and subjected to analysis concerning their strengths and weaknesses.

In Chapter 6, we present the algorithms selected for comparison and discuss their implementation. The simulation environment is reviewed along with the other algorithms and solutions involved, such as how the test scenes were constructed. The simulation results are also presented in this chapter.

Chapter 7 concludes the thesis.

Chapter 2

Three-dimensional scenes

In three-dimensional computer graphics, a scene consists of a number of *objects*, such as houses, cars or trees. These objects might have been created using some kind of three-dimensional modeling tool or they might be the end product of some data aquisition process, such as a medical magnetic resonance imaging (MRI) scan, a range scan or satellite imagery.

The objects, in turn, consist of *faces*, generally triangles (the word *primitive* is also used when referring to these basic building blocks). The faces make up the surface of the object. More faces means more detailed objects. Assuming a face is represented as a triangle, its extent can be completely defined using only three points in three dimensions, called *vertices* (sing. vertex). The positions of the vertices and the faces that are constructed from them make up the minimal amount of information we need in order to display an object. In reality, more information is often stored, such as the color of the faces, normals for computing the intersections between the vertices and light sources and so on.

In addition to the vertices, which define the extent of the object in *object space*, an object is often associated with a transformation matrix that defines its position, scale and rotation in *world space*. Using transformation matrices for positioning objects in our three-dimensional scene implies that the same raw object data can be reused many times; it is possible to display copies of the same object in different positions in the three-dimensional scene by simply using a new transformation matrix.

All the objects that make up a scene are collectively called the *scene geometry*. This can be further split into *static* and *dynamic* geometry. The static geometry is the part of the scene that does not move, such as roads and buildings. The dynamic geometry can change shape or move around in the scene; an example would be a car driving around on in the streets of a city, the streets being static geometry. The focus of this thesis is mainly

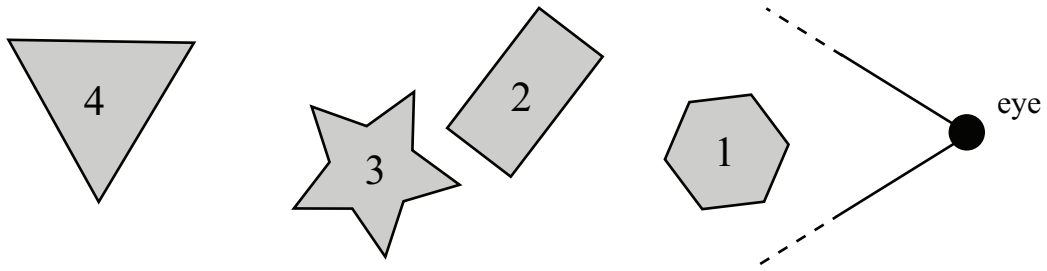

Figure 2.1: A sample scene containing four objects. The numbers indicate the render order when performing a front-to-back traversal of the scene.

on static geometry.

When a scene of some complexity has been created a need arises for a way to organize the objects that make up the scene. The structures that are used to organize the geometry are called *spatial data structures*.

2.1 Spatial data structures

A spatial data structure organizes the scene geometry in either two or three dimensions and usually in a hierarchical fashion. These ideas are extensible to higher dimensions as well, but usually this does not make much sense in a real-time rendering application. Spatial data structures are important because they enable the application developer to treat the scene geometry on a more abstract level than on the level of individual objects or primitives. It is important to realize that the spatial data structure does not make up a scene by itself, it merely processes the underlying objects in some fashion.

All spatial data structures work by dividing space into smaller parts, called *cells*. In a hierarchical scheme, the cells are organized into nodes where the top-level node contains the whole scene and nodes at lower levels enclose correspondingly smaller regions of the scene. Hierarchical structures are particularily well suited for visibility determination because when a node at a particular level is found to be non-visible, all of its children can be discarded as well. Hierarchical schemes usually keep the displayable geometry in the leaf nodes.

It is worth noting that most spatial data structures are rather expensive to compute and are therefore usually created in a preprocessing step. Updates to the data structures are generally possible to perform in real-time, however, with some variants being more suitable for handling dynamic updates than others; if for example moving an object means that whole structure has to be recomputed then the structure is not very suitable for dynamic objects.

One of the advantages of using spatial data structures is the possibility to perform a front-to-back scene traversal [4] (see Figure 2.1). This means that the objects that are closest to the viewer are rendered first, then the ones behind them and so on. This is, as will be shown later, very important in order to be able to calculate visibility efficiently. A spatial data structure provides the necessary proximity information, that is, it is easy to find objects that are located close to each other and it is in general possible to find at least an approximate front-to-back traversal [2]. The traversal is not strictly front-to-back since the individual objects inside a node are not ordered; however, while the performance of visibility calculation algorithms might be somewhat decreased by this, visual artifacts such as incorrectly rendered objects are not produced [4].

2.1.1 Object mapping

The step from having a nonordered collection of objects to having the scene geometry organized into a spatial data structure is called *object mapping*. Spatial data structures can usually be divided into two categories: firstly, structures working with the actual geometry of the scene and secondly, schemes which impose some kind of auxiliar structure on the geometry [21]. A bounding volume hierarchy is an example of the first category while grid and tree structures are examples of the second. These different structures are elaborated below.

When using an auxiliar data structure, a decision has to be made about what to do when a piece of geometry is intersected by one of the structure's cell borders. The geometry can either be split or it can be associated with both neighbouring cells. In many schemes the latter is preferred due to the marked increase in polygon count that can result from object splitting.

The scene geometry can be structured on many different levels; inputs to the routine that performs the spatial partitioning can be the logical objects of the virtual world, polygons or vertices, for example [34]. For scenes containing many dynamic objects, it may be more efficient to map the geometry at object level. Very large objects might still be split into a few sub-objects, but on the whole objects are kept intact. For static scenes polygon-level mapping is often done; this means that the cells of the spatial data structure are not necessarily aware of what the objects actually look like or what their attributes are as they deal directly with the polygons of which the object is made. The polygons that make up a wall may be split between two neighbouring cells, for example.

Regardless of which scheme is used, care must be taken to have the number of primitives contained in a leaf node stay above some threshold. This is due to the fact that modern graphics hardware deals more efficiently with large batches of polygons than

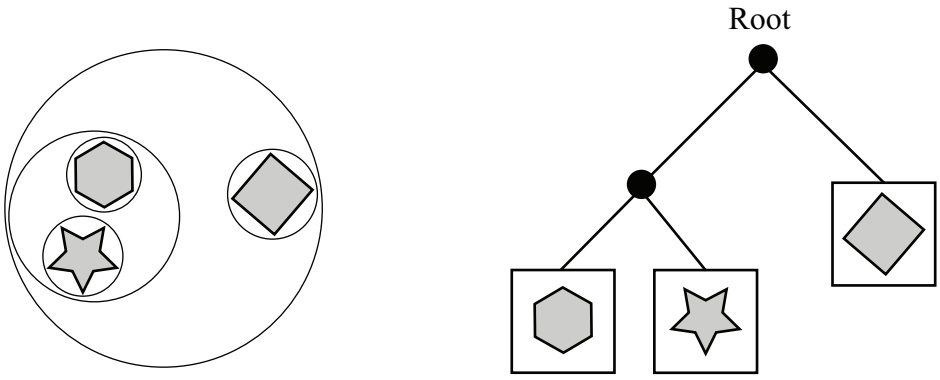

Figure 2.2: A sample scene and its corresponding bounding volume hierarchy. Spheres are used as bounding volumes.

with small clusters.

We will now continue by introducing seven different types of spatial data structures, namely bounding volume hierarchies, regular grids, octrees, loose octrees, kD-trees, p-HBVO's and adaptive binary trees.

2.1.2 Bounding volume hierarchy

Introduced already in 1980, a bounding volume hierarchy [33] provides a single, simple and hierarchical solution to the spatial subdivision problem. The basic idea is to replace the objects in a scene with simpler representations of the geometry, be it bounding spheres, axis-aligned bounding boxes or some other volume enclosing the object (see [4] for a thorough treatment of bounding volumes).

The first step in constructing a bounding volume hierarchy is therefore to enclose the individual objects with bounding volumes. When this is done, the objects (with their respective bounding volumes) are grouped together into a tree structure. Most applications use a top-down approach when constructing the tree [4]; the whole scene is first enclosed by a minimal bounding volume and this is then recursively partitioned with ever-shrinking clusters of objects close to each other being enclosed by likewise shrinking bounding volumes. The process continues until the each leaf node in the bounding volume tree contains a single object. These leaf nodes contain the actual geometry and are displayable. In Figure 2.2 a sample scene and its bounding volume hierarchy are presented.

Creating a bounding volume hierarchy is non-trivial. This is illustrated by the fact that the authors of [33] used an external editor to manually create the hierarchy. The main problem is that the number of possible hierarchies grows exponentially with the number of objects in the scene. In spite of this, various approaches to automate the creation process

have been explored (see for example [32]). Bounding volume hierarchies can be used for dynamic objects as well, one strategy being to remove an object from the hierarchy whenever the object moves enough to no longer fit within its parent's bounding volume. A new position in the hierarchy is then computed recursively, starting from the root.

2.1.3 Regular grids

A regular grid is probably the least complicated way of organizing the objects of a scene. The geometry is simply discretized into *voxels*, axis-aligned three-dimensional cells of equal size. The problem with regular grids is that densely and sparsely populated areas of the scene are partitioned using the same subdivision. This means that some voxels will contain a lot of geometry while others are empty or almost empty.

As a result of this regular grids are only suitable for scenes with more or less uniform content distribution. A scene containing randomly created and positioned objects is an example of such a distribution. Unfortunately, such randomly created scenes are common only in academic research (see for example [7]), which makes the results for studies that make use of a random content distribution and a regular grid scheme questionable.

Another example of a scene that could benefit from a regular grid structure is a typical terrain, which is a large object that represents the ground in an outdoor environment. Figure 2.3 shows a two-dimensional scene partitioned by a regular grid. Note that the regular nature of the grid makes it suitable for dynamic scenes as well, as long as we don't physically split any polygons; updating an object's position only involves removing the object from its current voxel and placing it into a new one.

When creating the regular grid, the resolution must first be decided. In one study using a regular grid structure, the optimal number of cells in the largest dimension was found to be between 35 and 50 using a very complex model [24], but since there is no such thing as a typical scene, this number is only indicative for the order of magnitude of the resolution.

2.1.4 Octrees

When constructing an octree [19], the whole scene is first encapsulated by an axis-aligned bounding box, the sides of which are equal in size. The size of the box is determined by calculating the spatial extensions of the scene; the length of the side of the box is equal to the maximum of the scene's three spatial extensions, one for each three-dimensional axis. The box is then split through the center along all three axes, thereby creating eight new smaller boxes. When working in two dimensions, the corresponding structure is called a

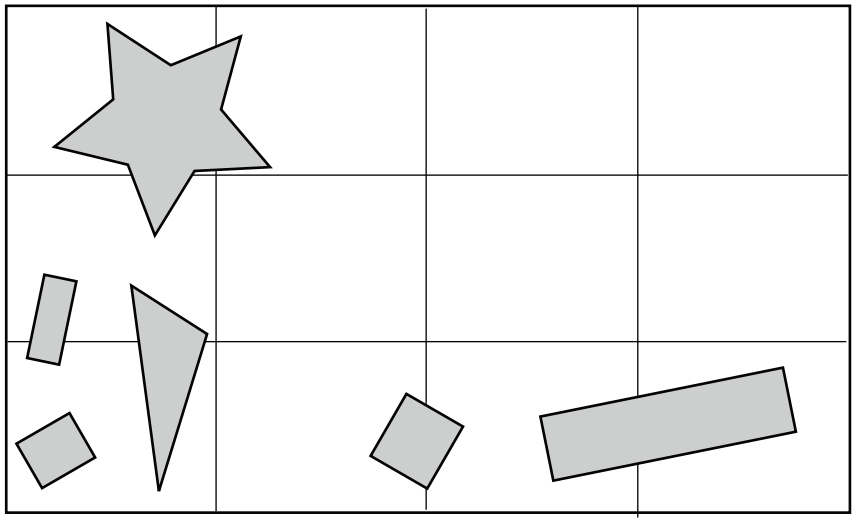

Figure 2.3: A two-dimensional scene partitioned by a regular grid. Notice the crowded region in the lower left corner, while at the same there are several empty voxels in the upper right corner.

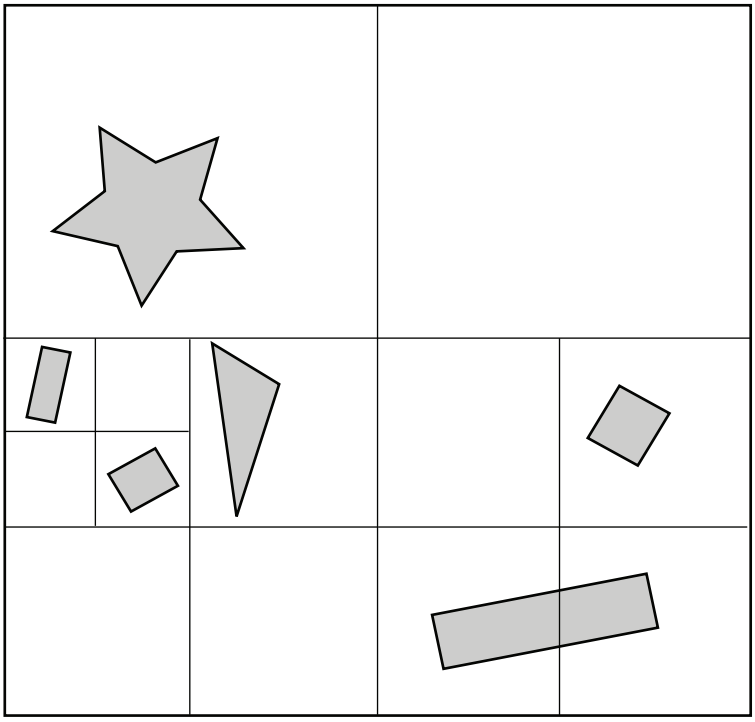

Figure 2.4: A two-dimensional scene partitioned by a quadtree.

quadtree where each cell is split into four subcells.

The subdivision process continues recursively until a certain criterion is met. The recursion can for example be terminated when the tree reaches a predefined depth or when the total number of nodes exceeds some threshold value. Another common choice is to end the recursive process when the number of objects per cell reaches a certain limit. As a result of this, the number of objects per cell can be nearly constant [2]. See Figure 2.4 for an example scene partitioned by a quadtree, the octree's two-dimensional counterpart.

Since an octree is inherently a cubic space partitioning scheme, it does not adapt very well to flat scenes; a typical example would be a city that spreads out horizontally but does not reach up very far vertically. It is also poorly suited for dynamic objects since a moving object must be removed and reinserted when it shifts position. This process changes the content distribution in the scene and we may be forced to, for example, merge old nodes that are suddenly almost empty or split a node that has become overcrowded. These are operations that are potentially expensive and should be avoided. Being a hierarchical structure the octree does, however, bring one significant advantage; if a node high up in the tree is found to be non-visible, all of its subnodes can be discarded as well. This is the fundamental principle that makes hierarchical spatial data structures well suited for visibility calculations.

2.1.5 Loose octree

Several more advanced variations of the basic octree structure have been developed. Loose octrees [36] provide easier handling of dynamic objects while maintaining the basic, hierarchical structure.

The voxels are called loose because the cells which are located on the same level in the tree are allowed to overlap, that is, nodes no longer define a unique area of space. Instead, the same space can belong to two neighbouring cells (see Figure 2.5). Where a normal octree would have cells of size $N * N * N$, a loose octree has cells of size $kN * kN * kN$, $k \geq 1$. The voxels' centers are still located in the same place, only the size of the voxels is expanded. The overlapping of the voxels makes culling less effective because the cell borders are no longer clearly defined, making it necessary to perform visibility checks for more cells than with a regular octree.

As can be seen in Figure 2.5, some special advantages arise for $k = 2$. Knowing the size of an object, the level of the loose octree it fits in is automatically known as well. Imagine an object enclosed by a bounding sphere with diameter 2; at the level of the octree where $N = 2$, the sphere will fit into a voxel regardless of its position within the voxel, because the voxel stretches out in all directions. The worst-case scenario is that

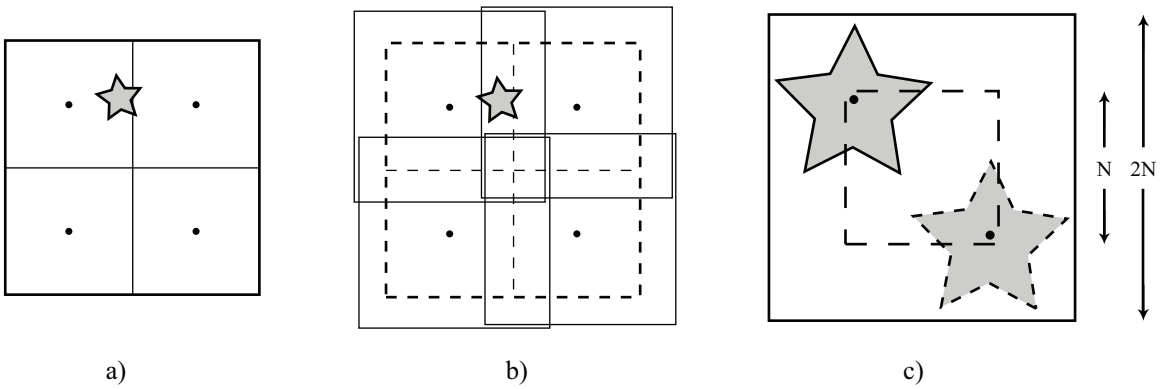

Figure 2.5: In a) a regular octree is shown. The black dots indicate the centers of the cells. Notice that the star-shaped object intersects one of the cell boundaries and would thus either have to be split or shared between the two nodes. In b) a loose octree with $k = 1.5$ is shown (the cell sides are therefore 50 % longer). The cells have been slightly displaced so that they can be discerned. The star now fits completely into the upper left cell. In c) a node in a loose octree with $k = 2$ is shown. The cell side length is now increased by a factor of two. The star-shaped object has a diameter of N, meaning that it will fit into the expanded cell wherever it is positioned within the cell (two sample positions are shown). The center of the object is shown with a dot. *(Illustration after Akenine-Möller and Haines [4])*

the center of the bounding sphere is located exactly on a grid border; in this case, there is still one radius' worth of voxel space left. This information makes it easy to update the position of a dynamic object because the level of the hierarchy where the object will fit in is easily calculated.

2.1.6 KD-trees

KD-trees [8], also called axis-aligned BSP (binary search partitioning) trees, are similar to octrees. Like an octree, the construction starts with finding a minimal axis-aligned bounding box (but unlike an octree, it does not have to be a cube). The main feature is that a kD-tree is binary, that is, at each subdivision stage the space is split into two parts. This means that number of children for a specific node in the tree is independent of the number of dimensions in the search space, whereas for example a node in an octree always has eight children due to its construction in three-dimensional space. For a node in a kD-tree the number of children is always two or zero, regardless of the number of dimensions.

The kD-tree also differs from the octree in the way it is constructed. Whereas an octree is partitioned regularily (all children of a given node are of the same size), the

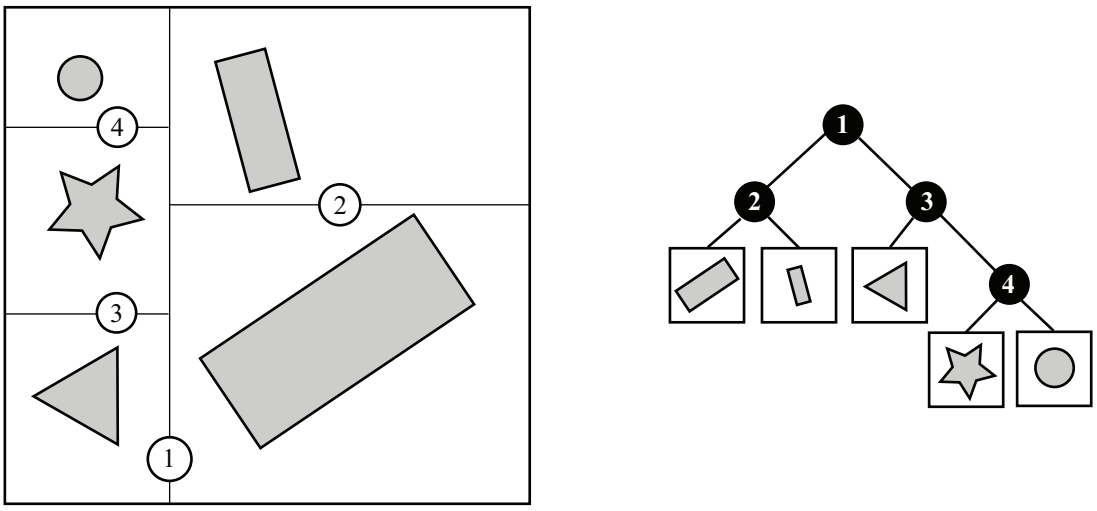

Figure 2.6: A two-dimensional scene partitioned by a kD-tree. The numbering indicates in which order the splitting planes were selected.

partitioning of a kD-tree can be varied by deciding the position of the partitioning plane according to some metric. The metric can be, for example, equal number of objects in both subspaces or least number of objects split. The recursive partitioning stops when a predecided criterion is met, usually when the number of objects in a cell is below some threshold or when the tree's depth is at a given level. See Figure 2.6 for an example of a kD-tree.

Since each subspace is split only by one axis-aligned splitting plane, a decision has to be made along which axis to split at each node in the tree. In [8], the author cycled through the X-, Y- and Z-axes (the first split, at the root, is made along the X-axis, the following along the Y-axis and finally along the Z-axis, then starting over with the X-axis). Another method is to check which side of the box is largest and split along that axis [4].

The kD-tree is a good choice for representing a static scene and is used in numerous modern algorithms and applications [2], [11]. It also easily supports rough front-to-back sorting [4], which is important when making visibility calculations.

2.1.7 P-HBVO's

Introduced by Meißner *et al.* in 1999 [30], the polygon-based hierarchical bounding volume optimization (p-HBVO) method is based on the kD-tree. A similar method was suggested earlier for use in ray-tracing applications [31]. The authors note that the p-HBVO method generates a well-balanced tree in the sense that any given node has two subtrees of roughly equal size. Being balanced, the time needed for traversing the tree is minimized.

The main difference between a regular kD-tree and a tree built with the p-HBVO method lies in the way the subdivision plane is chosen when it is time to partition a node. In the p-HBVO method, object mapping is done at the polygon level. The polygons are first ordered separately along all three coordinate axes using the polygon barycenter [1] as the sorting key. Based on the three ordered lists thus derived, potential subdivision planes are evaluated along all three axes. The optimal subdivision plane is chosen using a weight function that approximates the cost of rendering the polygons of both subspaces. By minimizing the weight function the splitting plane is obtained.

The cost of subdividing a cell, called H, into two children, H_{left} and H_{right}, along a given axis is given approximately by:

$$C_H(axis) = \frac{S(H_{left})}{S(H)} * |H_{left}| + \frac{S(H_{right})}{S(H)} * |H_{right}| \tag{2.1}$$

where

- $|H|$ is the number of polygons within cell H
- $S(H)$ is the total surface area of the bounding box of cell H, and
- $axis \in \{X, Y, Z\}$.

2.1.8 Adaptive binary trees

Adaptive binary trees (ABT's) have not received much attention from academic researchers, at least not in the context of computer graphics. The concept has however been widely discussed in online computer graphics forums [15], indicating that it is a well-known and utilized structure in the computer graphics community. Adaptive binary trees bring the flexibility of loose octrees to the kD-tree structure. Additionally, the cells on each level can dynamically adapt to better fit their content.

As with a kD-tree, the construction process starts with enclosing the scene with an axis-aligned bounding box. The process then continues recursively, splitting the cell into one left node and one right node. Choosing the subdivision plane requires some kind of heuristic; one option is to use the same one as was described in the previous section on p-HBVO's.

When the subdivision plane has been chosen, it will most likely split some primitives. This is where the tree becomes loose; both child cells, left and right, can expand into each

[1] For a triangle, the barycenter can be found by tracing three lines from each vertex to the center of the opposite side. The barycenter is where the three lines cross.

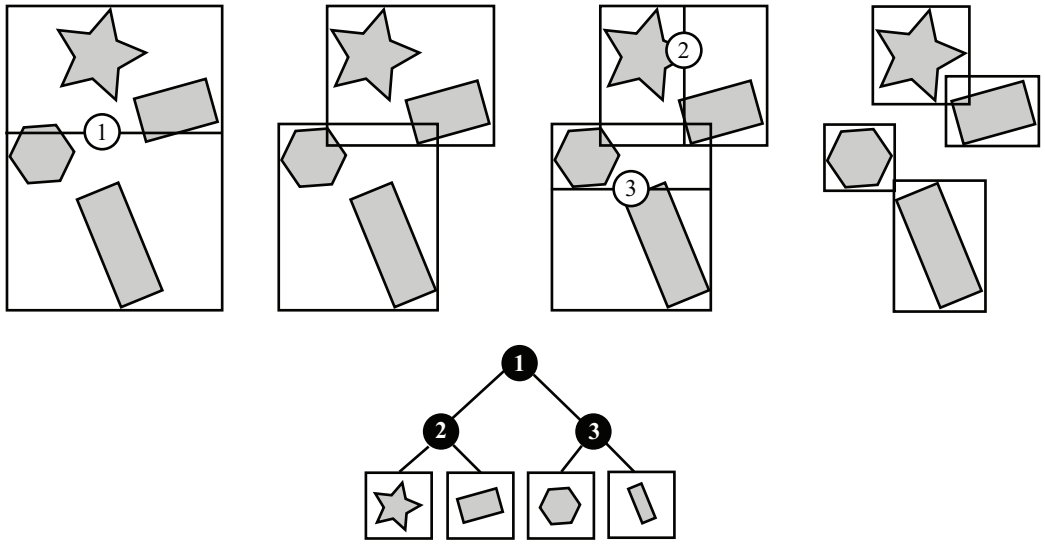

Figure 2.7: This scene shows the construction of an adaptive binary tree. In the leftmost picture the root node's subdivision plane has been chosen. Unfortunately, it splits two objects. In the next image, the two bounding volumes have grown into each other's spaces in order to fully accomodate the objects assigned to them. They have also been shrunk to better fit the geometry. In the third picture, two splitting planes have been chosen. Again, in the final image on the right, the cells have adapted to their content. The bottom picture shows the final tree.

other's space, just like a loose octree. Expanding the cell volume by only 5 % or 10 % will usually eliminate up to 90 % of the critical primitives [15]. While some primitives still have to be split the number of splits is not overwhelming.

When the left and right child nodes have been created, the tree will adapt further; each child node's volume is recalculated to better fit its contained geometry. See Figure 2.7 for an example. This gives us tighter fitting bounding volumes as compared to a normal kD-tree, something that is useful for accelerating visibility tests [6].

In an ABT, all geometry is stored in the leaf nodes. A node is determined to be a leaf node when it contains less than some threshold number of primitives. When all leaf nodes have been grown, the tree is recreated from the bottom up by combining the child bounding boxes into the parent node's bounding box. The need for this step stems from all the adapting done during the top-down recursion; the children nodes' bounding boxes are no longer the same as they were when they were originally created by their parent.

2.2 Summary

Before visibility calculation algorithms can be efficiently applied, the scene geometry must be organized in some way. A hierarchical organization brings the benefit of a tree structure; discarding a node will discard all of the node's children as well.

The scene geometry can be treated as a collection of objects (object-level mapping) or a collection of raw polygons (polygon-level mapping) when creating the spatial data structure. Performing object-level mapping can be preferrable for a scene containing dynamic objects while polygon-level mapping is often done for static scenes. Care must be taken to not make the displayable nodes contain too few primitives because the graphics hardware works best with large batches of polygons.

The spatial data structures presented in this chapter include the non-hierarchical regular grid and the hierarchical octree, kD-tree, p-HBVO-tree as well as the adaptive binary tree. The kD-tree has been proven to be a good alternative for static scenes. The p-HBVO-tree further adds on the kD-tree by introducing an intelligent way of choosing optimal subdivision planes. The adaptive binary tree provides an elegant solution for creating a tight structure with minimal polygon splitting, albeit at the price of a slight increase in complexity.

Chapter 3

Visibility in three-dimensional scenes

Assuming a walkthrough application, where the viewpoint moves through a virtual world consisting of a large amount of complex objects, the challenge of visibility determination is that at any point in time a large amount of the scene geometry will be invisible from the viewpoint, either because the objects are outside of the user's field of vision or because they are hidden by other objects located closer to the viewpoint. In this chapter the process of efficiently determining which objects are not visible and can be discarded, or *culled*, is described. Before going into the details of calculating visibility, some fundamental concepts will be defined.

Definition The *depth complexity* of a frame is equal to the number of primitives sent to the graphics hardware for processing divided by the total number of pixels in the frame buffer.

Figure 3.1 shows a particular frame that has high depth complexity. Although this situation is somewhat artificial, the idea is clear; it is desirable to avoid rendering objects

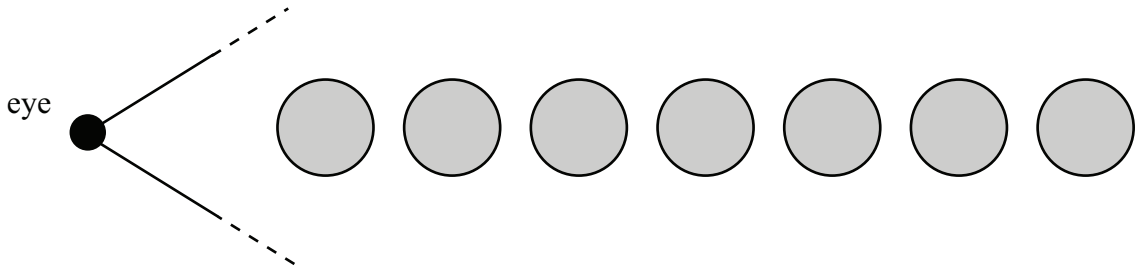

Figure 3.1: A sample scene with high depth complexity. If the spheres are rendered in back-to-front order the pixels in the center of the screen will be written to many times, even though the final image will only display the sphere located closest to the viewpoint. *(Illustration after Akenine-Möller and Haines [4].)*

that in any case will not be visible in the final image.

Definition An algorithm that performs visibility solving is *output sensitive* if the runtime is proportional to the amount of visible geometry and not to the total amount of geometry in the scene.

To achieve output sensitivity hierarchical processing of the scene is often necessary. By processing the scene hierarchically the part of the scene which is of interest - the one in the user's field of vision - can be found quickly. It should be noted that if an algorithm must access more of the total scene geometry than a linear function of the visible geometry, the principle of output sensitivity is violated [2].

3.1 The visibility pipeline

When approaching the problem of visibility determination, a pipelined approach is often taken [2]. Using a visibility pipeline allows us to implement different algorithms for different types of culling (how many of the stages that are actually implemented depends on the application and its requirements):

1. View frustum culling
2. Occlusion culling
3. Backface culling
4. Exact visibility calculation
5. Contribution culling

In the following sections the different stages of the visibility pipeline are described. Figure 3.2 illustrates the three first stages in the pipeline.

3.1.1 View frustum culling

The view frustum is the volume of the scene which is currently visible to the user. When a perspective projection is used, it takes the form of a cut pyramid where the volume is bounded by six planes. Four of the planes can be thought of as extending from the eye through the borders of the computer screen while the two remaining planes are orthogonal to the view direction and are called the *near* and *far* planes. The near plane acts as the "window" through which the scene is observed and is usually located very close to the

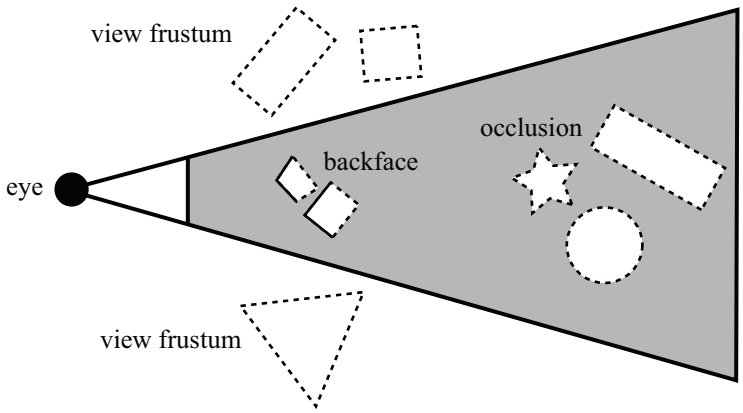

Figure 3.2: View frustum culling, backface culling and occlusion culling. *(Illustration after Cohen et al. [13].)*

viewpoint. The position of the far plane depends on the application; for a densely populated scene, a relatively short distance to the far plane is possible while for e.g. outdoor environments the far plane distance can be virtually infinite.

View frustum culling consists of testing objects against these six planes. This testing occurs in the application stage, on the CPU, and therefore objects culled here are never sent to the graphics hardware for processing. Testing every piece of geometry is expensive, especially for very detailed objects and therefore tests are usually done using bounding volumes. An even better way is to use a hierarchical spatial data structure, where culling away a node high up in the hierarchy will automatically cull all of the node's children and grandchildren as well.

3.1.2 Occlusion culling

Occlusion culling relies on spatial relationships between objects to find non-visible parts of the scene. This means that this type of culling is more complex than the other types. The basic idea is to discard objects that are blocked from view by other objects located closer to the viewer. Optimal occlusion culling will only draw the objects located closest to the viewpoint, therefore this type of culling is central to achieving low depth complexity.

While view frustum culling can be considered a largely solved problem occlusion culling is still the subject of much research. In Chapter 4 some methods to perform occlusion culling will be introduced.

3.1.3 Backface culling

When viewing a generic, convex object, an average of 50 % of its faces will be facing away from the viewer, thus being invisible. Backface culling is concerned with discarding the faces that are facing away from the viewer.

To achieve backface culling, two vectors are needed. One vector is traced from the viewpoint to an arbitrary point in the face's plane and is called the *view vector*. The other vector is the normal to the face's plane (remembering that faces are usually represented as triangles in computer graphics applications, the normal can be easily found from the three points that define the triangle [4]). If the angle between the two vectors is more than 90 degrees, the face points away from the viewer and can be discarded. In practice, it is enough to check the sign of the dot product between the two vectors [4]; if the dot product is negative, the face points away from the viewpoint.

Practically all modern graphics accelerators implement backface culling in hardware and it is often easily switched on by a simple command. This hardware backface culling will process polygons linearily. Techniques for performing hierarchical backface culling also exist [28]. It should be noted that polygons that are candidates for backface culling in hardware have already been sent to and preprocessed by the graphics processor; because of this, backface culling should be done only after view frustum and occlusion culling.

A special circumstance exists when the scene contains planar or near-planar objects, such as walls. These objects are not convex because they have no backside faces (the user is never meant to be able to see them from the back). Here, a situation can arise when all of the object's faces are pointing away from the viewer. In this case, backface culling can be applied already in the application stage to cull away the whole object [2].

3.1.4 Exact visibility calculation

This phase determines the visibility for each pixel in the final image. In modern graphics hardware this stage is practically always implemented using a depth buffer, also called the Z-buffer [12]. The Z-buffer is of the same size as the screen but instead of containing a color value for each pixel a Z-value is stored. This Z-value describes how far from the viewpoint the polygon fragment [1] that occupies the pixel is located. When a new fragment is to be drawn to the screen, its Z-value is first checked against the Z-buffer. If the Z-value indicates that it is closer to the viewpoint than the fragment drawn earlier, or if the Z-buffer is empty in this particular location, the fragment is drawn. Otherwise, the previously drawn fragment is closer to the viewer and the new fragment is discarded.

[1] The part of a polygon that will be drawn to one particular pixel is called a fragment.

It should be noted that the Z-buffer algorithm does not solve the problem of depth complexity because the fragments that act as the input to the algorithm are already fully processed and nearly ready for rendering. Gaining low depth complexity would require us to remove these polygons already before they are processed by the graphics hardware. This illustrates the demand for a visibility pipeline with more intelligent algorithms in the stages leading up to the exact visibility calculation.

Another way of achieving exact visibility would be to order all polygons in back-to-front order before rendering them. This is commonly called the painter's algorithm; the background is drawn first and then the objects located closer to the viewer. This is a quite naïve solution since every polygon must be fully rendered even if they are occluded in the final image. For overlapping polygons the sorting order can also sometimes become incorrect, leading to polygons being rendered erraneously.

3.1.5 Contribution culling

An object whose projection is determined to occupy only a few pixels on the screen can also be discarded. This type of culling is also called *aggressive culling*; the image will not be kept intact, instead visual artifacts will be produced. It is up to the application developer to determine if it is worth performing contribution culling or not. One way of lessening the impact of the introduced errors is to use distance fog, thereby blurring objects located far away so as to make it more difficult to see that the object is rendered somewhat incorrectly.

3.2 Exact, conservative and approximate visibility

As was described above, the exact visibility determination stage of the visibility pipeline is concerned with drawing the closest object at any given pixel on the screen. We can therefore guarantee that the final image is displayed correctly by only using the exact visibility determination stage, as long as all objects in the scene are processed. The problem is that this approach leads to a lot of unnecessary processing; there is no point in rendering the interior of a building if the building itself is not visible, for example. Processing all of the scene geometry and not just the visible part means that output sensitivity is lost. To achieve output sensitivity, a more intelligent approach must be taken.

To quote Aila [2]: "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.". Conservative visibility algorithms

attempt to identify the part of the scene geometry that is of interest. The algorithms are called "conservative" because they will most often identify a superset of the visible part of the scene. The conservativeness of a given algorithm can be estimated using the following metric [2]:

$$conservativity = \frac{R}{V} \tag{3.1}$$

where

- R is the amount of visible geometry reported by the algorithm and
- V is the actual amount of visible geometry, determined by way of e.g. an item buffer [2] or by rendering each polygon a different colour and reading back the frame buffer to see what polygons show up in the final image [24].

As we can see, a conservativity of 1.0 would mean that only the visible part of the scene was reported, no more, no less. This is what conservative algorithms strive to accomplish, with varying success. Since the beginning of the 1990's, the emphasis in visibility research has largely shifted to the investigation of conservative algorithms.

The term PVS, *potentially visible set*, is used for the conservative superset of visible polygons reported by conservative visibility algorithms. The PVS contains all visible geometry and some hidden geometry.

While a conservative visibility algorithm will report a superset of the visible geometry, algorithms also exist that will report a subset of the visible part of the scene. These algorithms employ approximate visibility calculations. The idea is to sacrifice some part of the visible geometry (and often, some part of the realism of the scene) in order to achieve better performance. For instance, an algorithm making use of approximate visibility solving might report a wall as visible but ignore the small clock hanging on the wall as an unnecessary detail. An example of such an algorithm can be found in [25].

Approximate visibility calculation can be of use in applications that must stay responsive at all times, even if it means degrading the visual quality. However, the focus of this work is on conservative visibility calculation. The following section describes what methods conservative visibility algorithms can use for quickly determining the visible part of the scene.

[2] An item buffer stores a unique number for each pixel in the frame buffer. This can be used to identify which object or polygon ends up on the screen.

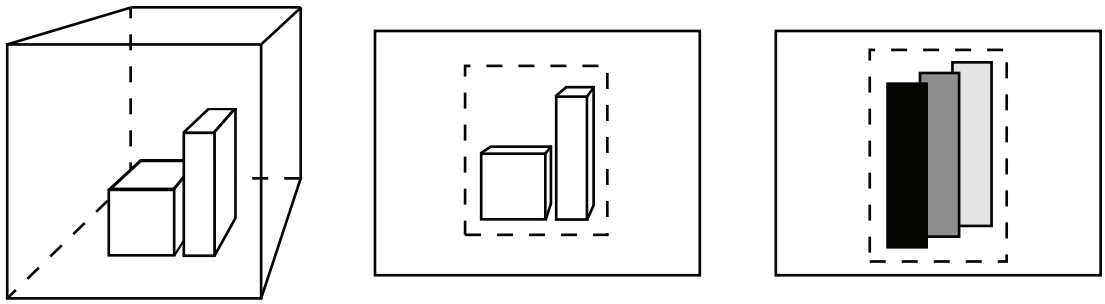

Figure 3.3: Three types of coherence; from left to right, object-space coherence, image-space coherence and temporal coherence. *(Illustration after Aila [2].)*

3.3 Coherence

Taking advantage of coherence is an important factor in any successful visibility algorithm. Although many forms of coherence can exist in a three-dimensional scene, three types are of special interest [2]; *object-space coherence*, *image-space coherence* and *temporal coherence*. Figure 3.3 illustrates these three categories.

Object-space coherence implies that for a group of objects located close to each other, some calculations might be made for the group as a whole with the results then being applied to all of the individual objects. It also means that if an object's bounding volume is e.g. found to be invisible it will not be necessary to process the contents of the bounding box any further. A hierarchical spatial data structure is a common way of taking advantage of object-space coherence.

Pixels located close to each other on the screen often belong to the same object and it is many cases possible to resolve the visibility for a region of the screen with a single computation. This is called image-space coherence. This type of coherence can also be associated with separate objects that can be located very far from each other; due to the perspective projection they might end up being drawn close to each other on the screen. Any algorithm that operates in two dimensions is said to exploit image-space rather than object-space coherence [2], for example the hierarchical Z-buffer algorithm that will be described in Chapter 4.

Temporal coherence is perhaps the one of these three types of coherence that is the most difficult to take advantage of. The basis for temporal coherence is that given a static or slowly moving viewpoint, something that can be said of the current frame will most likely also hold for the next. We will therefore use the result of the visibility calculations for the current frame as input to the calculations for the next frame. An example would be to assume that an object that is hidden in the current frame probably will stay hidden in the next few frames as well.

The problem of temporal coherence gains its complexity from the fact that it is generally not possible to know beforehand exactly how the viewpoint will move or how the visible set otherwise will change. In a sufficiently complex three-dimensional world, even a relatively slight motion might change the visible part of the scene dramatically; imagine, for example, turning a street corner in a detailed model of Manhattan. While many algorithms claim to be temporally coherent, they may in reality only benefit marginally from previously calculated results [2].

An interesting thought regarding temporal coherence is that, due to the cache architectures of modern computers, virtually all algorithms benefit from earlier calculations since the instructions needed to perform the calculations are cached in high-speed memory close to the CPU.

3.4 From-point visibility

The most natural way of solving the visibility for a given frame is perhaps to perform the calculations using the current viewpoint as a base. This is called from-point visibility calculation. One effect of from-point visibility calculation is that the visibility must be re-calculated every time the viewpoint moves; in reality, from-point visibility algorithms will perform their calculations once every frame.

There are also other ways of extracting the visibility information of a scene; one can for example imagine tracing a ray from the eye through each pixel on the screen. When the ray hits an object, the object is rendered to the pixel. This is called visibility along a line. Another approach would be to precalculate the visibility from every location inside a region, for instance from inside a room in an architectural model. As long as the viewpoint stays inside the region, the visibility does not have to be recalculated. This approach is called from-region visibility and can be a powerful method at least for indoor scenes.

The focus of this thesis is on algorithms dealing with visibility from a point because they are the ones best suited for covering arbitrary scenes. The bulk of current research also focuses on this family of algorithms.

3.5 Summary

The problem of visibility in a three-dimensional scene is complex enough for there to be no single algorithmical solution. Instead, applications usually take a pipelined approach using a combination of algorithms. It is desirable for visibility determination scheme to achieve output sensitivity, which means that the runtime should be proportional to the

amount of visible geometry, not to the total amount of geometry in the scene. Output sensitivity can be gained by for example hierarchically processing the scene to quickly find the interesting part, the visible one. Another desirable property of a visibility algorithm is low depth complexity. Ideally, only the object located closest to the viewer should be drawn for each pixel.

Good visibility algorithms strive to take advantage of coherency in the scene. In this way, a single visibility computation is used in more than one place in the scene, thereby amortizing the cost of the computation over several objects (for object-space coherence) or pixels (for image-space coherence). The cost of performing visibility computations can also be amortized over time using temporal coherence.

For generic three-dimensional scenes, from-point visibility calculation is the most logical choice, as opposed to e.g. from-region visibility solving.

Chapter 4

Occlusion culling methods

Occlusion culling is concerned with discarding objects that are blocked from view by other objects located closer to the viewpoint. Several strategies for performing occlusion culling have lately been researched. The purpose of this chapter is to introduce some of the more well-known solutions and analyze their strengths and weaknesses. One can identify two major categories of occlusion culling algorithms, namely those who perform large amounts of preprocessing (for example, a from-region algorithm that precalculates the visibility from e.g. every room in an architectural model) and those who perform occlusion culling online, without significant amounts of preprocessing. The focus of this chapter is on the latter type, because algorithms of this type solve four problems associated with precalculating the visibility:

1. The precalculated potentially visible set (PVS) is only valid for the original setup of the scene, therefore dynamic objects have to be excluded.

2. For algorithms calculating a from-region PVS, the PVS from a given viewpoint within a cell is often overly conservative. See Figure 4.1.

3. Computing the PVS from every possible viewpoint or from every region is a computationally heavy task, resulting in long application startup times.

4. It is difficult to implement an accurate PVS calculation for a general scene. While it is in most cases possible to calculate the PVS for e.g. indoor models where walls act as natural subdividers, for an outside environment it can be a very challenging task.

It should be noted that not all precalculating algorithms calculate the entire PVS beforehand; some variants only try to identify some polygons that might work well as occluders later on, for example. This might be an acceptable behaviour for certain scenes.

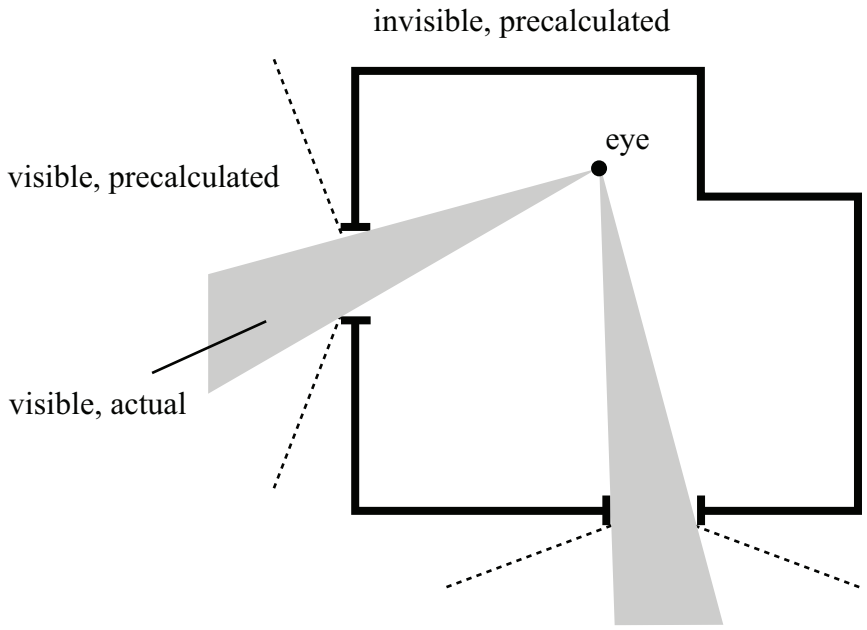

Figure 4.1: This scene shows why precalculating the visibility on a from-region basis can be overly conservative. The precalculated visibility must be valid from every point inside the region (in this case a room with two doorways), even though the actually visible region may be much more limited.

The price of the added flexibility of online occlusion culling algorithms is that more calculations have to made on a frame-to-frame basis. Towards the end of the chapter, a hardware extension specifically designed to ease the calculations needed for online occlusion culling will be introduced.

4.1 Fundamental concepts of occlusion

4.1.1 Occluders and occludees

An object which blocks another object from the user's view, given the current viewpoint, is called an *occluder*. All visible objects are potential occluders. The portion of the scene which is occluded by an occluder gives a measure of its *occlusion power*. An object which is blocked from view by one or more other objects is called an *occludee*. The *occluder set* consists of all occluders.

It may be noted that, for a conservative occlusion culling algorithm, it is acceptable to use a simpler representation of an object, e.g. its bounding box, when testing if the object is an occludee. This might lead to a case where the bounding box is found visible, but in reality the actual object is occluded (for instance, when only a corner of the bounding

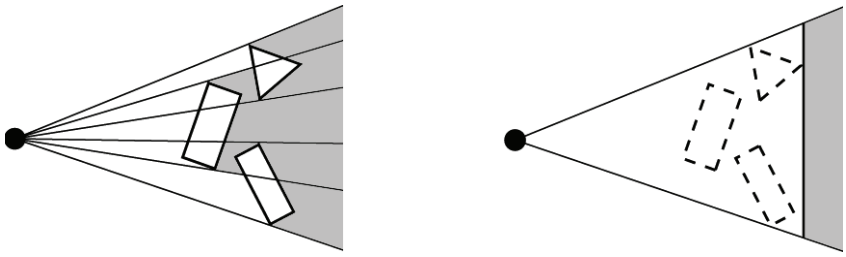

Figure 4.2: Virtual occluder. The combined occlusion power of the three objects on the left is represented on the right by a single entity, a virtual occluder.

box is visible). The object is then rendered unnecessarily, but the correctness of the final image does not suffer. The visibility overestimation is compensated by the fact that it is usually much easier to test the visibility of a simple shape like a bounding box instead of testing the detailed object it contains. For use as an occluder, however, the actual object must be used because the reverse of the above is not acceptable; Object A, found to be occluded by object B's bounding box, might in fact be visible if considering the actual object B. This would introduce errors in the final image.

4.1.2 Occluder fusion

It is easy to see that combining several objects provides more occlusion power than handling each object separately. A simple example would be a forest; a single tree might not occlude very much, but the sum of many trees will occlude a lot. Good occlusion algorithms automatically provide occluder fusion.

4.1.3 Virtual occluders

It can be observed that a cluster of objects which through occluder fusion occludes a large part of the screen can be replaced by a much simpler object providing the same occlusion function. See Figure 4.2 for an example. For most occlusion algorithms, dealing with big and simple polygons is much preferrable to investigating the occlusion power of several small objects. Law and Tan [29] use simplified versions of the original objects as virtual occluders while Bernardim *et al.* [9] use the nodes of the octree used to partition the scene. Virtual occluders can also mean volume-preserving simplifications of scene geometry; for example, a cylinder can be, for the purposes of occlusion testing, replaced with two crossed rectangular polygons.

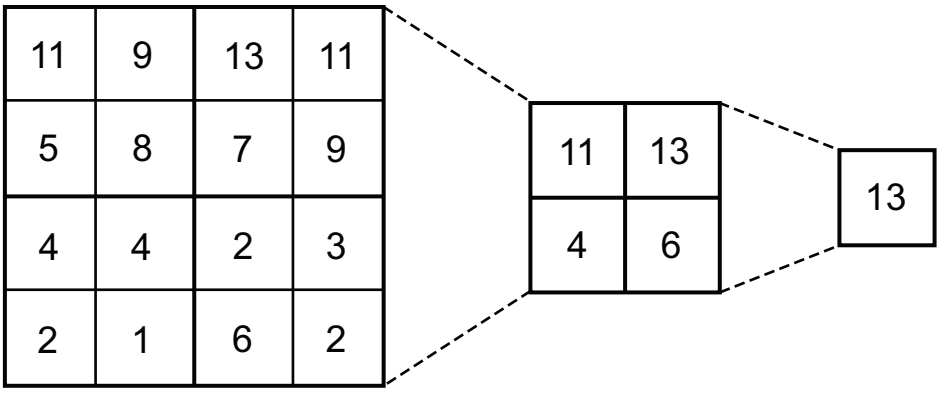

Figure 4.3: A hierarchical Z-buffer or *Z-pyramid*. On the left is a 4x4 region of the raw Z-buffer, the content of which represents the distance from the viewpoint to the polygon fragments rendered to that pixel. For each level of the pyramid, the highest Z-value of a 2x2 region is taken. Finally, only a single value remains, indicating the farthest Z-value of the original region.

4.2 Hierarchical Z-buffer visibility

Introduced by Greene in 1993 [20], this algorithm is perhaps the first occlusion algorithm to exploit all three forms of coherence (object-space, image-space and temporal coherence). It is a conceptually important algorithm that has also been implemented, albeit in an altered form, in recent graphics hardware.

Greene uses two hierarchical data structures, an octree in object-space and a Z-pyramid in image-space. The Z-pyramid's base is the Z-buffer, which represents the distances of the polygon fragments drawn on the screen as explained in Section 3.1.4. The Z-buffer may be initially empty but will be filled with Z-values along the way as objects are determined to be visible and subsequently drawn. At each following level in the Z-pyramid, the Z-value of four neighbouring fragments are compared and the one farthest away from the viewer is stored. The process continues in this fashion until there is only one value left which represents the farthest Z-value in the whole image.This process is illustrated in Figure 4.3.

In a preprocessing step the scene is organized into a basic octree (see Section 2.1.4). At render time, the octree is then traversed starting from the root. In order to determine the screen area in which to perform the visibility test for an octree node the faces that make up the node are projected onto the screen. The screen-space bounding rectangles of the projected faces are then used as the region for which visibility test will be performed.

While the visibility test region is thus known, the value with which to perform the test still needs to be calculated. This is determined from the nearest Z-value of the original face. This value is then compared to the Z-pyramid, starting at the top. If the value in the

Z-pyramid is closer than the face's Z-value, the face is occluded. Occluded octree nodes (that is, nodes whose faces have all been determined to be occluded) are immediately skipped.

Otherwise, if a node cannot be proved to be occluded for a region of the Z-pyramid, the algorithm continues recursively to the next level of the pyramid. The four quadrants on the next level that make up the previously tested region are now used for the visibility test. Here we perform the occlusion test for every quadrant intersected by the screen-space projection of the octree node. This makes the resolution of the test finer.

If the occlusion test fails for any quadrant, we continue the recursion into that quadrant. Ultimately it will either be proven that the whole polygon is occluded or a definitely visible pixel will be encountered when the recursion reaches the finest level of the Z-pyramid. This means that the octree cube is visible, so any geometry that is associated with it is rendered. The process then continues recursively with the children of the octree node, rendering them in a front-to-back order.

The Z-pyramid must of course be kept up-to-date. This is easily achieved by updating the finest level of the Z-pyramid every time the contents of the Z-buffer change and then propagating the changes to the coarser levels in the pyramid.

Temporal coherence is exploited using a simple temporal coherence list, which consists of the octree cubes found to be visible in the previous frame. The cubes in this list are drawn first without any occlusion testing, thereby providing a reasonably good initial content for the Z-pyramid. The algorithm then proceeds as normal, skipping the nodes already rendered but filling in the remaining gaps, adding visible nodes to the temporal coherence list. The final step is to perform the Z-pyramid test for all the nodes in the temporal coherence list in order to remove previously visible nodes that have become occluded in the current frame. This prevents the list from becoming too large.

A version of this algorithm using a cut pyramid with only a few levels has been implemented in hardware by among others ATI, who calls it *HyperZ technology*. The basis for this technology is a three-level Z-pyramid. For every 8x8 pixel region, the nearest and farthest z-value of the polygon occupying that region is taken and their visibility is checked against the Z-pyramid. The test is done just before the pixel pipeline and uses almost no bandwidth because the Z-pyramid can be stored on-chip [5]. However, the geometry is already in the GPU pipeline when this test is done (it is necessary to transform and roughly rasterize it before applying the test), even if it is later determined to be non-visible.

In essence, the hierarchical Z-buffer algorithm is a very good one. It provides occluder fusion, exploits all three types of coherence and is basically suitable for arbitrary scenes using any kind of primitives, although a better spatial data structure than an oc-

tree should probably be used. The problem is that current hardware does not support the necessary reading of the Z-buffer and updating of the Z-pyramid to make real-time frame rates achievable. If one day hardware that solves these problems is developed, the hierarchical Z-buffer algorithm might become the *de facto* standard for conservative visibility determination, just like the Z-buffer algorithm is the *de facto* standard for exact visibility determination [2].

4.3 Hierarchical occlusion map

Zhang's hierarchical occlusion map (HOM) [39] algorithm draws upon the hierarchical Z-buffer algorithm but is designed to be used with current graphics hardware. The scene is partitioned using a bounding volume hierarchy in a preprocessing stage, where the scene's most significant occluders are also gathered into an occluder database. This preselection of occluders implies that the algorithm is not very suitable for dynamic scenes, which makes it less interesting for the purposes of this thesis, but the main ideas of the algorithm are still interesting to explore. It is also theoretically possible to replace the occluder selection mechanism with something more dynamic.

The algorithm splits the visibility test into two parts, a coverage test and a depth test. For a given object, Zhang uses a screen-space bounding rectangle of the projection of the object's bounding volume to perform the coverage test. In this test, it is determined whether or not the object is covered by an occluder. It is however not yet decided whether or not the potential occludee lies *behind* the occluder; this is the task of the depth test.

The coverage test is done using a hierarchical occlusion map (HOM). For each frame, a subset of occluders from the occluder database is chosen using heuristics like the position of the viewer and the view frustum dimensions. The HOM is built by rendering these occluders into the frame buffer as pure white-on-black drawings. This gives us the silhouettes of the occluders and what regions of the frame buffer they occupy. The frame buffer is then read back and a pyramid is built by averaging squares of 2x2 values for every single value on the next, coarser level. The procedure is the same as for the hierarchical Z-buffer algorithm, except this time we get the average and not the maximum of the 2x2 square. This step can be done in hardware that supports mipmapping [1], a common feature in current consumer graphics chips. As a result of the downsampling some regions might

[1] A mipmap takes its name from the Latin phrase "multum in parvo", meaning "much in little". It is a lower-resolution version of a texture which is normally used when an object is located far from the viewer and it is not necessary to use the original, fully detailed texture. "Mipmapping" refers to the process of creating these smaller textures. Each smaller mipmap is a quarter of the size of the previous one.

become gray instead of black or white, meaning that a part of that region is occluded.

In order to test a particular object, a screen-space bounding rectangle of the projection of its bounding box is used. The level of the HOM where the rectangle fits most closely into one pixel is then selected (remember, as we get closer to the top of the HOM pyramid each pixel represents a larger region of the actual frame buffer). If the value in that pixel is not opaquely white, the object is determined to be visible. If the value is opaque, the object is covered; its projection coincides with the occluders'. Note that at this stage we can perform contribution culling (see Section 3.1.5) by setting the required opacity to something less than 100 %.

If an object is determined to be covered by the selected occluders it remains to determine if the object is behind or in front of its would-be occluders. This is where the depth test is applied. The depth test can be implemented in a number of different ways. One way is to maintain a depth buffer in software; when the occluders are rendered into the frame buffer to construct the HOM they are also rendered into a special depth buffer, which often has a lower resolution than the frame buffer. The depth buffer performs like the Z-buffer but it is initialized to the nearest value, not to the farthest value like the Z-buffer is. In fact, it is possible to implement the depth buffer using current Z-buffer hardware except that the depth-test function must be reversed (we want to pass objects that lie *behind* the current contents of the depth buffer). This assumes that the hardware supports easily reading back the result of a depth test.

When performing the depth test using a depth buffer, all entries in the buffer that are covered by the object's screen-space bounding rectangle are compared against the farthest Z-value of the object's bounding box. If a test indicates that the bounding box is located in front of the occluders, the test fails.

If both the coverage and the depth test pass the object is determined to be occluded and is culled away. If any one of them fails, the object is rendered. The HOM is however not updated even if an object is rendered. This lack of incrementality is a clear disadvantage of this algorithm. Another downside is that the algorithm relies completely on selecting good occluders in every frame and since this is a non-trivial problem it is hard to see how the algorithm can easily adapt to the general case. Temporal coherence is also difficult to handle and the process of reading back the frame buffer in order to perform the coverage test introduces significant latency. On the good side, the algorithm provides occluder fusion and does not require any special hardware.

4.3.1 Incremental occlusion map

In the dPVS visibility framework by Hybrid [2], a derivative of the HOM algorithm has been implemented that solves many of the original algorithm's problems. As was stated in the previous section, one of the main drawbacks of the HOM algorithm is the fact that it uses the frame buffer to create the occlusion map and has to read back the contents of the frame buffer in order to process the visibility. In the incremental occlusion map (IOM) algorithm, this step is avoided by extracting the silhouettes [2] of the occluders already in object space. Since this is a software operation it is easy to incrementally add occluders as needed. The drawback is a big increase in algorithm complexity; Aila reports that the current IOM implementation in dPVS uses over 10,000 lines of C++ code [2].

4.3.2 Occlusion map with software renderer

As discussed in an Internet game developer forum [16], implementing the HOM algorithm using a purely software renderer seems a viable solution to avoid reading back the Z-buffer. If the software renderer is invoked when the GPU is still busy rendering the previous frame, the occlusion map can be rendered almost for free [3]. This assumes a heavily optimized software renderer (a hand-optimized assembler implementation is suggested). As a further distinction from the original HOM algorithm, the coverage test is skipped completely in favour of a hierarchical depth test. This turns the HOM algorithm more or less into the hierarchical Z-buffer algorithm. The difference is that an occluder selection is still rendered (using the software renderer) as the first step.

The occluder selection consists of virtual occluders made up of low-polygon, volume-preserving simplifications of the actual scene geometry. Exactly which of these occluders are rendered is determined from their current position, their rotation and similar heuristics. Some parts of the scene geometry are also tagged beforehand as being good occluders, such as walls and ceilings. The occluders are kept in a separate spatial hierarchy independent of the visible geometry.

The advantage of this variation of the HOM algorithm is its simplicity and its reportedly good performance [16]. A clear disadvantage is that an occluder selection is

[2]The silhouette of an object is what is percieved as the object's border after it has been projected onto the screen.

[3]If the CPU is not busy doing other computations, that is. In an application that is only concerned with moving through a three-dimensional world this assumption might hold, but in for instance a game application the CPU must often also take care of complicated physics, artifical intelligence and game logic calculations.

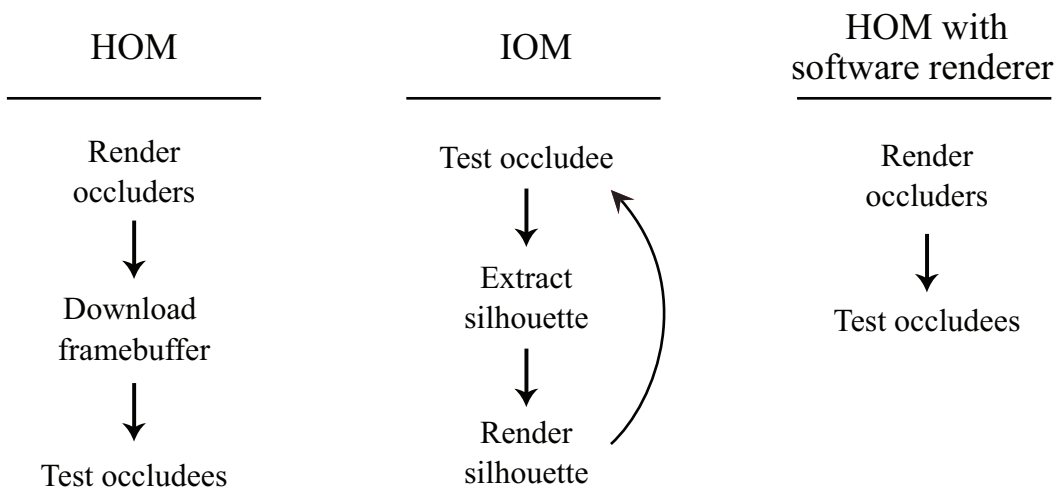

Figure 4.4: The pipeline of the original HOM algorithm compared to more recent developments. The original HOM algorithm suffers from the problem of downloading the frame buffer in the middle of the pipeline. The incremental IOM algorithm avoids this and further adds occluder incrementalism. Using a software renderer it is also possible to avoid downloading the framebuffer.

still needed and in some cases the virtual occlusion geometry has to be hand-made [16]. Occluder incrementation is also lost because an occluder selection is used.

Figure 4.4 shows the basic functionality of the HOM algorithm using a software renderer as well as the original algorithm and the incremental IOM algorithm.

4.4 Hardware occlusion queries

As we have seen in the previous algorithm descriptions, the central question an occlusion culling algorithm has to ask is often this: ”Would this object be visible if it were rendered?”. A hardware occlusion query means asking the hardware this question directly. This mechanism is readily available on current consumer graphics hardware. It was also available on at least one high-end graphics accelerator already in the early 1990’s, when Greene experimented using it in conjunction with his hierarchical Z-buffer algorithm [20].

Hewlett-Packard implemented a hardware occlusion query mechanism in their fx series of graphics hardware in 1999 [35]. Exposed as an OpenGL extension called GL_HP_occlusion_test [17], the function is also commonly implemented on current consumer graphics accelerators. Using this extension, a query can be made whether a set of primitives would have been written to the Z-buffer, should they actually have been drawn. The idea is that the visibility of a complex object can be investigated by issuing an occlusion query for a much simpler enclosing volume, such as a bounding box. If the bounding box

is determined to be occluded, it is not necessary to process the actual object any further.

Although a good idea in principle, the GL_HP_occlusion_test extension has two major shortcomings; it only returns *true* or *false* to indicate the queried geometry's visibility, whereas it is often interesting to know *how much* of the queried geometry is visible. An object which only contributes to very few pixels might be contribution culled, for instance. Knowing how much of an object is visible also presents some nice opportunities for special effects, for example the visible percentage of a light source might be used for determining its brightness.

Another drawback of the GL_HP_occlusion_test extension is that it is implemented as a simple stop-and-wait mechanism; the application must stall while waiting for the answer of a query. In other words, the queries are not serializable, something that goes against the trend of increased pipelining in modern graphics hardware. It also makes it harder to exploit parallellism between the CPU and the GPU, that is, to have the CPU perform other calculations while the GPU is busy rendering geometry.

NVIDIA introduced an extension called GL_NV_occlusion_query in 2002 [18] which solves the main problems associated with the HP extension. It is now possible to perform several queries in series and get their results at a later stage. In between, the CPU can be held busy doing other application-related computations. The queries also return an integer number indicating how many pixels passed the depth test, *i. e.* how many pixels were visible. When hardware occlusion queries are discussed in the context of modern graphics hardware and occlusion algorithms, the NVIDIA extension is usually implied.

4.5 Algorithmic usage of hardware occlusion queries

Hardware occlusion queries offer a temptingly simple solution to the visibility problem; enclose everything in bounding boxes, perform depth queries on them and render the ones determined to be visible. The problem is that the hardware still has to rasterize the bounding boxes, an operation which certainly isn't free (a bounding box will normally cover more pixels than the actual object because of its square nature). The feedback loop is also problematic; because of the heavy pipelining in modern graphics accelerators the queries cannot be very fast. Algorithms that make use of hardware occlusion queries thus attempt to minimize the queries performed by exploiting for example temporal coherence while still maintaining good culling discipline. When queries do have to be performed they are issued in series, amortizing the delay over several invocations.

It has been noted that using tighter bounding volumes than simple axis-aligned boxes can provide significant performance increases [6], one example being k-DOP's [4]. The

adaptive binary tree structure described in Section 2.1.8 also provides tighter bounding volumes than for example a normal kD-tree. Using bounding volumes that occupy a much large object space volume, as well as a larger on-screen region, than the actual object it encloses increases the chance of an object being signalled as visible even though it actually is not, thereby increasing the conservativeness of the visiblity calculation.

As is the case with most occlusion culling methods, a front-to-back rendering of the scene geometry is vital for good performance when using hardware occlusion queries. However, in today's applications there are other factors that need to be taken into account when deciding in which order to draw the objects, such as shader state changes. Sending objects in the same shader state after each other through the rendering pipeline is beneficial for performance. The two goals of front-to-back rendering and minimizing shader state changes are often at odds with each other. It seems that in practice, less importance is given to maintaining a front-to-back rendering order [1], something that will hurt performance especially for algorithms doing occlusion culling using the hardware query mechanism.

In Chapter 5, several algorithms that make use of the hardware occlusion query mechanism will be presented.

4.6 Summary

Online visibility determination is preferrable to precalculating the PVS, and in this chapter some methods of performing this in a from-point visibility context were explored, along with supporting fundamental concepts. The hierarchical Z-buffer algorithm is an example of an elegant algorithm that unfortunately needs special and so far nonexistent hardware to achieve interactive speeds. Hierarchical occlusion maps are better suited for use with current hardware but suffer from the lack of incrementalism and the fact that occluders have to be preselected. However, variations of the HOM algorithm show promise of being viable all-round software solutions.

Hardware occlusion queries have the potential of being a very good tool to perform occlusion culling, although they have to be used with care in order to avoid pipeline stalling.

Chapter 5

Hardware occlusion query algorithms

In this chapter a number of algorithms that perform occlusion culling using the previously introduced hardware occlusion query mechanism will be presented. In the next chapter, data acquired from simulations using some of the described algorithms below will be presented in order to compare their performance.

5.1 Fast and Simple Occlusion Culling

The "Fast and Simple Occlusion Culling"-algorithm was presented by Hillesland *et al.*, researchers from the Walkthrough group at University of North Carolina at Chapel Hill, in 2002 [24]. The idea is to present a very simple and generic solution to the occlusion problem, a solution that is also well suited for dynamic scenes. This algorithm is one of the first to make use of the newer occlusion query extension by NVIDIA.

The scene is partitioned using a regular grid data structure (see Section 2.1.3). A nested grid solution was also explored, but in practice the difference in performance was marginal. The voxels of the grid are rendered in rasterized planes roughly orthogonal to the view vector. One layer of voxels is rendered before moving to the next layer, located further away from the viewer.

The authors note that pipeline stalls are a problem when using hardware occlusion queries. Their solution is to simply perform N queries at a time and then retrieving N answers, where N is a number that has to be tuned manually to achieve optimal performance. This solution is perhaps overly simplistic.

Simulations were made using the power plant model [1], comparing the algorithm to

[1]The power plant model, readily available for download from the UNC-Chapel Hill Walkthrough Group's web pages [38], depicts a coal-fired power plant using more than 13 million triangles. Several authors use this model to verify their algorithms' performance because it is a quite good example of a

pure view frustum culling. For this simulation setup, the algorithm shows a noticeable performance increase as compared to not using occlusion culling at all. The potentially visible set determined by the algorithm is about 20 % of the total geometry contained in the view frustum. The average frame time decreased from 0.36 seconds when using only view frustum culling to to 0.09 seconds for this algorithm.

While providing a measurable performance increase the algorithm still needs development. The regular grid structure is not particularily well suited for scenes where the geometry density varies, for example. There are also a number of parameters that have to be hand-tuned for each individual use case, such as the number N and the resolution of the regular grid.

5.2 Conservative Prioritized-Layered Projection

The Prioritized-Layered Projection (PLP) algorithm [25] is a software algorithm that works with a fixed polygon budget. The algorithm attempts to render those polygons that are most important for achieving a correct image. Since it won't render all polygons, artifacts are introduced - small but still visible objects may become culled, for example. The algorithm to be described here is a conservative version of the PLP algorithm (known as cPLP) that can make use of hardware occlusion queries. The cPLP algorithm works as an extension to the PLP algorithm and hence the latter will be described first.

5.2.1 PLP

In the original article on PLP the scene is first subdivided using an octree structure, but actually the algorithm is not limited to any specific spatial partitioning scheme. However, it should be noted that inside the view frustum the algorithm does not function hierarchically; it only works with the leaf nodes that contain the actual geometry. The leaves are not traversed in a predefined order, as would be the case for a normal front-to-back traversal. Instead, the leaves are kept in a priority queue called the *front*, traversing leaves with high priority first. When a leaf is visited and determined to be visible it is added to the *visible set* and its immediate unvisited neighbours are added to the front. The traversal thus moves away from the viewpoint in layers of leaf nodes (note that in order to support this traversal algorithm, it is necessary for the nodes in the spatial hierarchy to know about all of their neighbours). The traversal stops when the algorithm's polygon budget is reached,

complex, densely occluded environment.

that is, when the number of polygons in the visible set is above some threshold. These polygons are then sent to the graphics hardware for rendering.

In order to set the priority of a given node, and thus the traversal order, several heuristics can be used. One possibility is to use the *solidity* of the node, meaning that nodes containing a lot of geometry are considered more solid than cells containing only some geometry. The solidity can then be accumulated along a traversal path; when a node is rendered, its solidity is transferred to the neighbours behind the node as seen from the viewpoint. The solidity transfer takes place using an algorithm that takes into account some properties of the individual neighbours, for instance by how much they overlap. The priority of a node can then reflect the accumulated solidity of the nodes lying between itself and the viewpoint. If a lot of solidity has been accumulated, it is likely that the node is occluded and it is subsequently given low priority.

Another simple heuristic would be to assign a priority of -1 to the node containing the viewpoint, -2 to its neighbours, -3 to the following neighbours and so on. This will make the expansion stretch out from the viewpoint in a regular fashion. This heuristic is both easy to implement and gives relatively good performance [26].

5.2.2 Conservative PLP

When the approximate PLP algorithm has reached its polygon budget and stops rendering a frame, two structures have been created: the depth buffer containing the Z-values of the approximate scene and the front queue containing the cells which the algorithm would have rendered next had it had more time. Based on this information some conservative image-space method is applied until the front queue is empty. In [26], an item buffer (see Section 3.2) is used while using the HP occlusion query mechanism (among others) is explored in the original paper [27]. It would be trivial to implement the conservativeness using the NVIDIA occlusion query extension, perhaps using a method similar to the one used by Hillesland *et al.* [24] for avoiding pipeline stalls.

Simulations using both the PLP and the cPLP algorithms have been performed [26] using the power plant model. For the PLP algorithm the average time needed for rendering a frame was 0.09 seconds while using the conservative algorithm increased the frame time to 0.67 seconds. The correctness of the approximate PLP algorithm was also measured; on average, 96.7 % of the visible polygons were rendered. The authors report that the incorrectly removed polygons were often located far away from the viewpoint, therefore having only a small impact on the user experience. This correctness was achieved using the simple heuristic described above and using a more sophisticated one would probably

increase the accuracy.

5.3 Coherent Hierarchical Culling

The Coherent Hierarchical Culling (CHC) algorithm [11] was presented in 2004 when the occlusion query extension had already been around for a while and researchers had had a chance to think about intelligent ways of exploiting it. As such, the CHC algorithm takes advantage of both spatial and temporal coherence and provides a somewhat more sophisticated method for avoiding pipeline stalls.

The subdivision scheme used is a kD-tree. At render time, the tree is traversed hierarchically front-to-back and terminates either at leaf nodes or at internal nodes found to be invisible. Such nodes are called *termination nodes*. Interior nodes that are classified as being visible are called *open nodes*. The sets of termination nodes and open nodes at frame i are called T_i and O_i, respectively.

In frame i, the algorithm skips occlusion queries for the nodes of O_{i-1}. This is logical because an open node visible in the previous frame is probably visible in the current one as well. For the nodes belonging to T_{i-1}, the algorithm performs an occlusion query. If the node is a leaf, the geometry associated with the node is also immediately rendered. This is because we assume that a previously visible leaf node is still visible. See [11] for a complete pseudocode implementation.

The visibility status of any given node is reset to invisible in each frame. When a node is classified as visible this information is pulled up in the hierarchy, from child to parent. This way we do not have to keep track of the O_i and T_i sets explicitly - it is enough to store a visibility bit and a counter in order to know in which frame the node was last visited. Figure 5.1 shows how the visibility changes from one frame to the next for a sample hierarchy.

The algorithm reduces the number of occlusion queries in two ways. Spatial coherence is exploited by only handling a subtree of the scene hierarchy in every frame (O_i and T_i). As a result of the visibility pull-up mentioned earlier every invisible termination node has a visible parent but can have invisible children. This means that the occluded parts of the scene are represented with a minimal amount of nodes. These are the nodes for which occlusion queries are issued and thus it is easy to see that the number of queries that have to be performed has been minimized. Temporal coherence is utilized by avoiding queries on open nodes, reducing the number of issued queries by a factor of $1/k$, where k is the branching factor of the scene hierarchy [11].

In order to combat the latency associated with occlusion queries the CHC algorithm

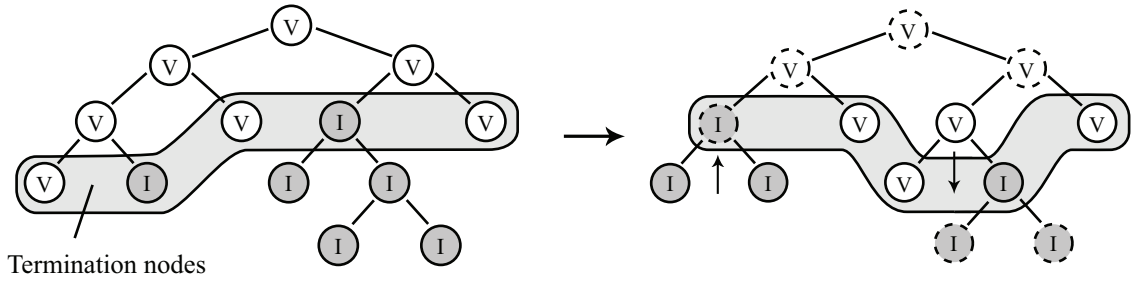

Figure 5.1: In this figure, two consecutive frames of a sample hierarchy are presented. In the bottom frame, the nodes for which occlusion queries were issued are marked with a solid outline. As we can see on the left, the invisible status of two sibling nodes has been pulled up to their parent. On the right, visibility has been pulled down as a result of a child node becoming visible. The figure also shows the set of termination nodes for each of the frames. *Illustration after Bittner et al. [11].*

does not simply perform an occlusion query and then wait for the hardware to provide the result. Instead, the queries are interleaved with the rendering of (previously) visible geometry. If this approach is successful, the results of new queries become available in between and the pipeline stalls are completely eliminated. The interleaved behaviour is achieved by using a queue for the occlusion queries; when an occlusion query is issued, it is added to the back of the queue. The algorithm continuously checks if the result of the occlusion query at the front of the queue is available and if not proceeds with rendering previously visible geometry.

If the visibility is changing, a situation can arise when an occlusion query is issued for a previously invisible node A that is actually occluded by another previously invisible node B that has become visible in the current frame. Because A and B were both invisible, occlusion queries were issued for both of them. However, if B is still in the query queue when A is tested, A is falsely determined to be visible because the occluding geometry of B has not been rendered yet. In the simulations performed in [11], it is shown that the increase in rendered nodes due to this problem is in practice very small.

Some possible optimizations are discussed, although the basic algorithm already provides significant performance gains compared to using only view frustum culling. The CHC algorithm also outperforms a naïve stop-and-wait algorithm that makes use of hardware queries. Since no preprocessing is performed and no tuning is needed for special cases, the CHC algorithm is well suited for the general case. Further, the implementation of the algorithm is not overly complex. There are some drawbacks, however; in any given frame, hardware queries have to be made for all previously visible leaf nodes in order to correctly set up the visibility information throughout the hierarchy via the visibility pull-

up functionality. This can be a source of unnecessary queries. Another issue is that the transition from a node being view frustum culled to being occlusion culled is not smooth, since a hardware query is issued for any node that was view-frustum culled in the previous frame. This behaviour will certainly lead to the issuing of unnecessary queries [22]. The CHC algorithm also adapts poorly to scenes with low depth complexity where the overhead of performing occlusion queries can actually degrade the performance enough to make it slower than using only pure view frustum culling.

5.4 Near Optimal Hierarchical Culling

Introduced in 2006, the Near Optimal Hierarchical Culling (NOHC) algorithm [22] starts out in a normal fashion by first hierarchically subdividing the scene geometry (the p-HBVO subdivision scheme introduced in Chapter 2 is used). The algorithm shares similarities with the CHC algorithm but takes a more analytical approach to the problems associated with using hardware occlusion queries.

Unlike the CHC algorithm, queries can also be issued for nodes outside the set of termination nodes. In fact, several nodes in the same subtree can be considered if at first a query fails. A cost/benefit function is applied to every node inside the view frustum and a query is issued if it is decided that the cost associated with the query is less than the expected benefit or if the node belongs to some special category, such as being previously occluded. The cost of making an occlusion query for a given node is also compared to the cost of issuing queries for its children separately; take, for example, a situation where one child node is located outside of the view frustum and one inside. In this case, it is cheaper to test only the single child node which is located inside the view frustum.

The cost/benefit function involves estimating how much time it would take to directly render a node H_i as compared to issuing an occlusion query for it and rendering it if found visible. Similarily to the CHC algorithm, queries are always issued for previously occluded nodes. For a previously visible node, a query is made only after the node has been rendered without being subjected to occlusion queries for n frames, so that the cost $C(H_i)$ is less than the benefit $nB(H_i)$.

The cost $C(H_i)$ is determined for each level of the hierarchy separately but the benefit $B(H_i)$ is accumulated over all levels. Therefore, the benefit has to be distributed among all levels by dividing it with the depth of the hierarchy. Given the total number of nodes in the entire hierarchy N_h, we get:

$$\begin{aligned} C(H_i) &= t_o(H_i) \\ B(H_i) &= p_o(H_i)(t_r(H_i) - t_o(H_i))/log_2(N_h + 1) \end{aligned} \tag{5.1}$$

where

- $t_o(H_i)$ is the estimated time needed to perform an occlusion query
- $p_o(H_i)$ is the occlusion probability, and
- $t_r(H_i)$ is the estimated time needed to render the node

The parameters t_o and t_r are measured when starting the application since they are hardware dependent. The details of the measurement process can be found in [22]. The probability p_o is a function of the portion of the screen that has already been drawn to by previous nodes, located in front of the current node in the front-to-back traversal of the hierarchy, and the size of the current node. In essence, occlusion is considered highly probable if a lot of geometry has already been rendered and if the current node is small. Temporal coherence is brought into the equation by considering the occlusion status of the node in the previous frame. If a node was occluded in the previous frame, it is assumed to be occluded again. If a node was visible, the probability of it being occluded in two consecutive frames is considered, thereby introducing the concept of visible objects tending to remain visible. Again, the details of the occlusion probability estimation process are available in [22].

When an occlusion query has been issued for a node H_i the NOHC algorithm also considers whether to add the set of the children of H_i, called $S(H_i)$, to the traversal queue immediately or only after H_i has been found to be visible. Adding them immediately preserves the front-to-back traversal but the time spent processing the $S(H_i)$ nodes is wasted if H_i is later found to be invisible. Further, an immediate insertion delays our percieved performance gain by one frame, because removing a node due to occlusion will only remove its children in the following frame. Delaying the insertion can provide an immediate performance gain, although the front-to-back traversal order is broken and nodes that would actually have been occluded by $S(H_i)$ can mistakenly be determined to be visible. The NOHC algorithm addresses this issue by immediately adding $S(H_i)$ to the traversal queue if H_i was previously visible or view frustum culled and delaying the insertion if H_i was previously occluded. For comparison, the CHC algorithm only inserts $S(H_i)$ directly if H_i was previously visible.

While the NOHC algorithm certainly is a step further in complexity compared to the previously covered algorithms, the implementation of it should not be an impossible task. The authors compare its performance primarily to the CHC algorithm and use the same models and walkthrough paths as used in [11]. The NOHC algorithm displays a clear performance increase over CHC and unlike the latter never displays poorer performance than pure view frustum culling even for scenes with low depth complexity. The NOHC algorithm thus shows that it is capable of adapting to a variety of different scenes. Avenues for improvement still exist, however, mainly by researching better analytical models for the cost/benefit function that governs the issuing of occlusion queries. One slight drawback is that the algorithm relies on performing various measurements of rendering speeds in a preprocessing stage during application startup, something that can be time-consuming. This is especially true if the scene contains objects using many different materials and shaders because the rendering times must be measured independently for each type of shader, or at least for every group of shaders of similar complexity. Relying on hardware measurements also means that the actual performance gain will vary among hardware platforms making it impossible predict performance exactly.

5.5 Summary

The development of the hardware occlusion query mechanisms has given birth to a new subarea of occlusion culling research. Algorithms that take advantage of occlusion queries are attractive because of their inherent simplicity; a query is made and if it fails, the geometry is not rendered. The main issue to consider is the latency between issuing an occlusion query and recieving the response; if this latency did not exist, the hardware occlusion culling mechanism would be an optimal solution to the visibility problem.

Several approaches to the problem with varying degrees of complexity were presented in this chapter. Currently, the best solution seems to be to tie the decision between issuing a query for a node or simply rendering it to a cost/benefit function that can adapt to different types of scenes with varying depth complexity, spatial coherency and so on.

Chapter 6

Implementation and results

In order to test what performance gains can be expected from employing occlusion culling as well as to perform direct comparisons between occlusion culling algorithms, an occlusion culling testbench application was developed. The application was developed in an object-oriented fashion using C++ and consists of around 15,000 lines of code spread out over 42 classes. Figure 6.1 shows an UML class diagram of the most important program components.

In order to generate the three-dimensional scenes with which the different culling algorithms could be tested the program supports loading PLY files[1]. The test data used consisted of three scenes providing very different types of occlusion:

- A scene consisting of randomly placed and orientated objects of two types, one with a medium amount of primitives and the other with a very large amount of primitives. The total number of triangles was 2,119,200.
- A subset of the power plant model, totalling 4,764,188 triangles.
- A scene consisting of four very detailed dragon statues courtesy of the Stanford 3D Scanning Repository [37], placed in a circle for a total number of triangles of 3,485,656.

The random scene offers good overall occlusion and can be expected to be the one for which occlusion culling can offer the largest performance increases. The power plant scene is a realistic model of a real-world architectural entity and offers very varying degrees of occlusion, depending on which part of the scene is observed. Both the random

[1]PLY is a file format used to describe three-dimensional objects and has been developed at Stanford University. It has been widely adopted for use in computer graphics research.

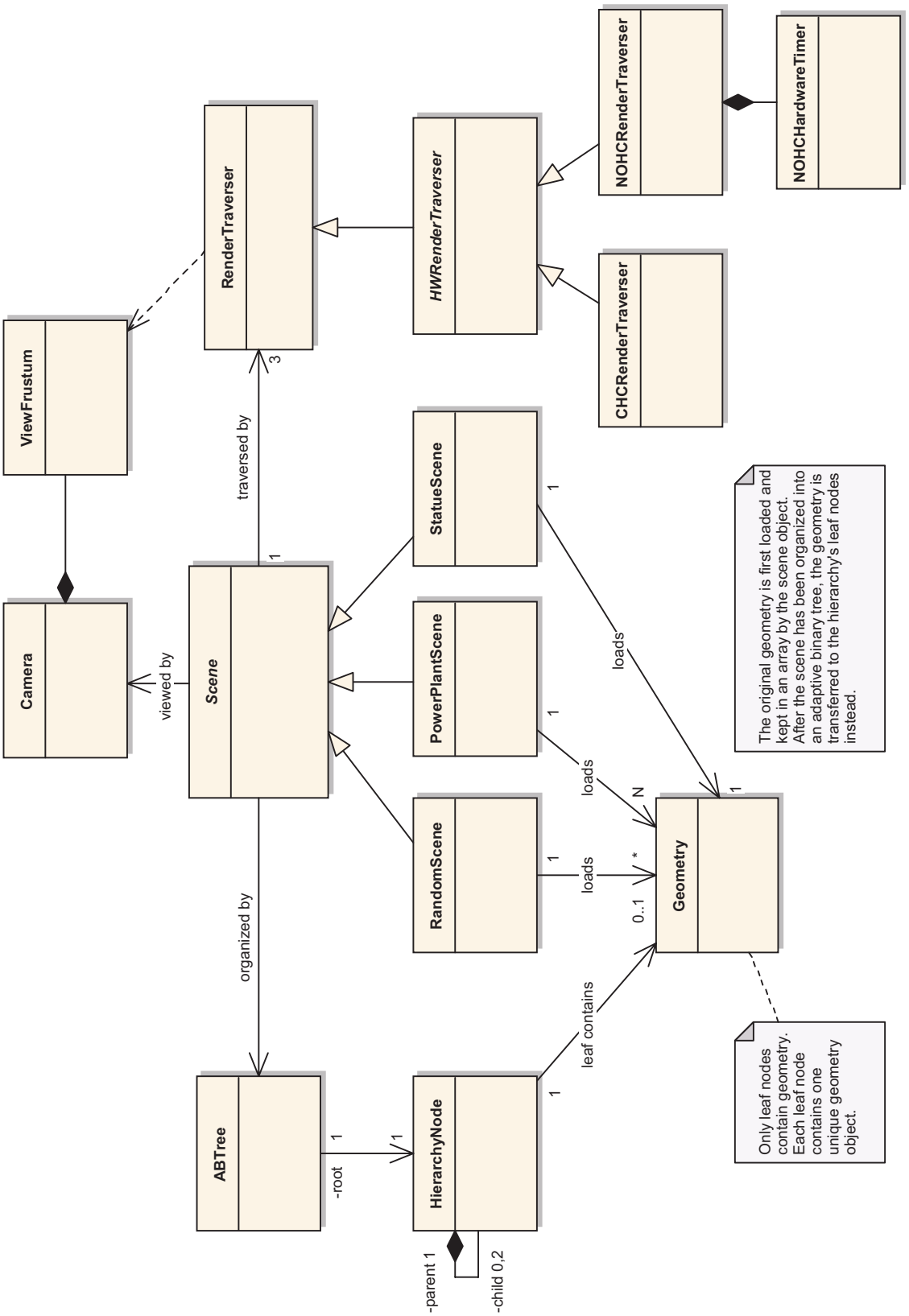

Figure 6.1: Occlusion culling testbench class diagram.

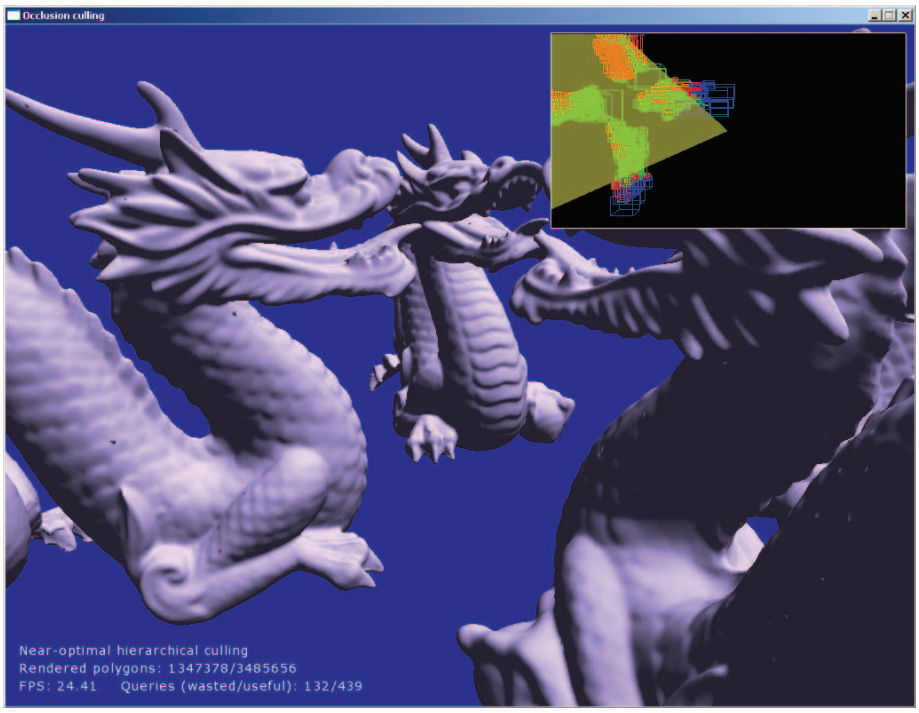

Figure 6.2: An example view of the occlusion culling testbench program. Visible hierarchy nodes are rendered in green, occluded nodes in red and nodes outside the view frustum in blue.

and power plant scenes generally have high depth complexity. The statue scene, on the other hand, is a scene with relatively low depth complexity Here the model is highly detailed but from any given viewpoint only a small part of it is occluded. The statue model offers especially interesting challenges for an occlusion algorithm that strives to be applicable to any type of scene, because the approaches that work well for a case with high depth complexity does not necessarily cover low depth complexity as well. The three test scenes were all organized using the same spatial subdivision scheme, namely an adaptive binary tree (see Section 2.1.8).

In order to gather the occlusion culling benchmarks systematically the program supports recording the motion of the viewpoint around the scene and then repeating the same path several times, only altering the way the visibility culling is performed. This keeps all factors constant except the culling approach. In order to keep the focus on the culling of primitives, the objects were rendered with low visual detail (e. g. no texturing was applied). The application maintains a visual log of how the visibility culling proceeds by simultaneously displaying an overhead view of the scene with the viewpoint located in the middle and visibility-culled hierarchy nodes drawn in different colors. See Figure 6.2 for a sample view of the program.

Tests were performed for three different visibility calculation methods:

- View frustum culling only.
- Coherent hierarchical culling (see Section 5.3).
- Near-optimal hierarchical culling (see Section 5.4).

The NOHC algorithm was implemented slightly differently from the original description [22]. The main difference is in the analytical model used to decide if it is desirable to recurse into a given hierarchy node's children or if it is better to perform an occlusion query at the current level. The changes are summarized as follows:

- When calculating the cost of recursing into a node's children, the cost of performing new occlusion probability calculations at the childrens' level in the hierarchy is taken into account in addition to the rendering costs. The cost of recursion amounts to a number of square root calculations.
- Temporal coherency is only exploited for nodes that have been confirmed to be visible by an occlusion test.

The last point means that when a node is determined to be visible for reasons other than a failed occlusion test, for example if the algorithm decides that it is cheaper to render a node than it is to perform an occlusion query for it, the node loses its confirmed visibility status and temporal coherency calculations can no longer be applied. This is logical because if the algorithm arbitrarily decides that a node is visible without performing an occlusion test it is essentially guessing. Under these circumstances it is not desirable to assume that a node thus determined to be visible will stay visible, because making a bad guess (flagging a node which is in reality occluded as visible) can incur penalties over a large number of consecutive frames, as the program in vain attempts to take advantage of temporal coherency. The changes were made because it was discovered that the near-optimal hierarchical culling algorithm, at least in the implementation used for the purposes of this thesis, in many cases radically overestimated the visible part of the scene, thereby rendering many polygons that were actually occluded.

6.1 Results

Measurements were made using two different hardware setups. The main measurement metric was frames-per-second, which directly correlates with how the user perceives the system. The average number of triangles rendered was also calculated. A third metric which is interesting for algorithms that implement hardware-assisted occlusion culling is

	ATI Radeon 9700			Nvidia GeForce 6600 GT		
	FPS	**Triangles**	**QW**	**FPS**	**Triangles**	**QW**
	Random scene					
VFC	37.6	1,049,091		39.1	1,049,091	
CHC	66.3	338,441	173.2	56.2	338,061	172.8
NOHC	76.0	353,415	56.2	58.6	344.926	87.0
	Power plant scene					
VFC	24.1	1,232,929		22.2	1,232,954	
CHC	27.4	600,598	305.6	32.8	492,489	249.2
NOHC	35.5	661,138	65.1	37.5	519,380	93.0
	Dragon statue scene					
VFC	14.1	2,878,831		17.5	2,878,831	
CHC	22.3	1,743,632	914.8	20.0	1,743,690	915.2
NOHC	21.5	1,793,318	98.1	24.4	1,774,368	222.0

Table 6.1: Comparison of average frame rate in frames per second (FPS), average number of triangles rendered and average number of occlusion queries made for visible nodes (QW) for different visibility culling methods and graphics cards.

the number of wasted occlusion queries, that is, the number of occlusion queries that were performed for nodes ultimately found to be visible. This metric, along with the number of triangles rendered, illustrates with how well the algorithm succeeds in determining the visible set. The results are given for view frustum culling, coherent hierarchical culling and near-optimal hierarchical culling with the modification mentioned above.

An illustration of the average frame rate over a period of time is given in Figure 6.3.

6.2 Discussion

As can be seen from the results in Table 6.1, the occlusion culling methods that were implemented always provided a noticeable gain as compared to only using view frustum culling. The most tangible benefits were experienced in scenes with high depth complexity, such as the random scene and the power plant scene. As for the relationship between the CHC and NOHC methods, the NOHC algorithm performed better in all cases except one; the dragon statue scene using the ATI Radeon 9700 hardware. It seems that in this case, the NOHC algorithm overestimates the cost of performing occlusion queries. The difference is however marginal.

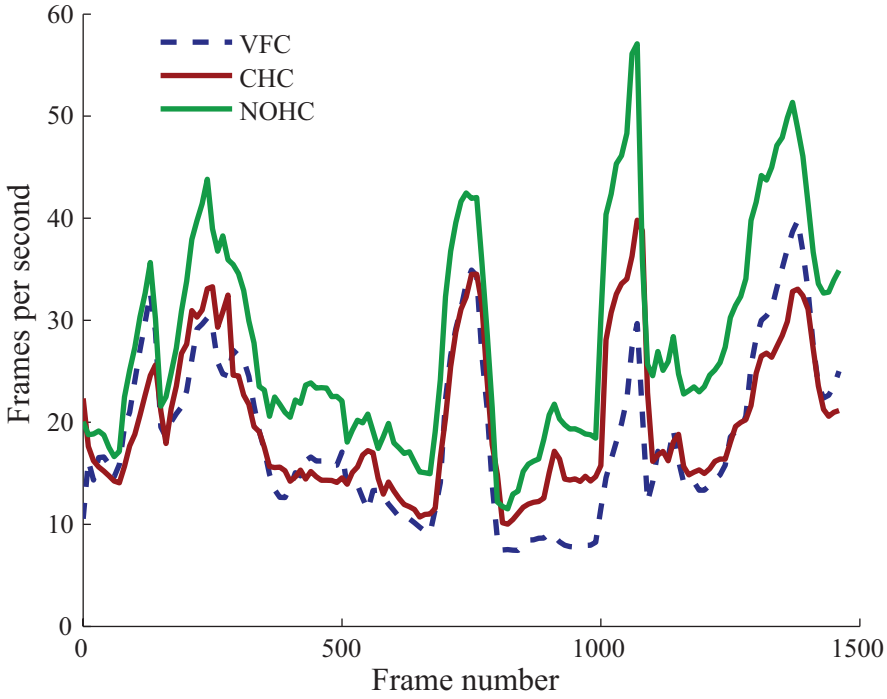

Figure 6.3: The average frame rate as a function of time for the three different culling methods while rendering the power plant model, using an ATI Radeon 9700.

It is also interesting to view the test results in light of algorithm implementation complexity. While view frustum and coherent hierarchical culling both were quite easy to implement, using only introductory material [4] and a short conference paper [11] as references, the near-optimal hierarchical culling algorithm demanded significantly more work in order to achieve acceptable performance levels. The difficulty in implementing the NOHC algorithm lies with the fact that the algorithm consists of several subalgorithms that have to work in unison in order to achieve good culling performance, such as the hardware timing subsystem and the occlusion probability calculations. Implementing the NOHC algorithm might not have been viable at all had the original authors [22] not kindly made their source code available for use as a reference. Still, the implementation used here is still not perfect as illustrated by the worse-than-CHC performance for the statue scene using the ATI Radeon 9700.

Considering both the level of implementation difficulty and the visibility culling performance, the CHC algorithm might very well be a better option in many cases than the NOHC algorithm. If however the best possible culling performance is required and sufficient time for development is available, the NOHC algorithm can be seen as preferrable.

The performance of both occlusion culling methods is dependent on the scene hierarchy. A finer subdivision generally provides better occlusion culling results. The spatial

hierarchy used in the occlusion culling test bench performed well in most cases, however, in the power plant scene, small objects such as pipes that consist of long and thin polygons caused significant polygon splitting; upwards of 10 % of the original scene's polygons ended up being split. It was also discovered that the adaptive binary tree implementation frequently and inadvertedly produced a large number of nodes containing very few polygons. Although this situation could probably have been alleviated by using better heuristics when choosing the subdivision plane when splitting a node, time was short and the original subdivision scheme was kept. Instead, only a subset of the original power plant model was used. This subset still provides good challenges for occlusion culling algorithms. Even though the employed scene hierarchy may be suboptimal, both occlusion culling methods would suffer similar performance penalties and therefore the scene hierarchy should not be a factor when measuring the difference in occlusion culling performance between the two.

Finally, the method for traversing the scene hierarchy plays an important role for occlusion culling. In the implementation of the occlusion culling test program used here, a rough front-to-back traversal is used, meaning that when the traversal proceeds from a given node, the child which is closer to the viewpoint is entered first. This is known as a depth-first traversal. As noted in [11], a distance queue is a better choice; when the traversal proceeds from a node, its children are entered into a queue at a queue position that depends on the distance between the node and the viewpoint. This makes the traversal breadth-first, which means that nodes whose distance to the viewpoint is similar are traversed together, regardless of the hierarchy depth of the nodes. The traversal is said to proceed in *depth layers*, expanding out from the viewpoint. This approach is beneficial from an occlusion culling point of view mainly because the processing order within a depth layer is arbitrary, leading to nodes whose would-be occlusion is independent of each other being processed after each other. For the simpler depth-first traversal, if a node is occlusion tested and found to be occluded, the nodes traversed directly after it are most likely occluded as well; the following nodes' occlusion status therefore depends on the preceding node. A breadth-first traversal is particularily efficient in solving the problem of missed occlusion by nodes still in the query queue as explained in Section 5.3.

Taking the two points introduced above into consideration, it is possible that the performance of both occlusion culling algorithms implemented in the testbench application could be significantly increased, at least in terms of relative performance in comparison with using only view frustum culling. Since the two implemented occlusion culling algorithms are conceptually fairly similar, it is likely that they would both enjoy similar performance increases if the above mentioned improvements were implemented.

Chapter 7

Conclusion

This thesis gives an introduction to the issues involved with visibility calculations in three-dimensional scenes. In particular, the field of occlusion culling is presented in detail because this area is both the most interesting and most difficult part of correctly determining the visible portion of a three-dimensional scene in computer graphics. Along the way, the bits and pieces that make up a successful visibility solving application have been introduced, such as the spatial structures that are used for hierarchically organizing the scene.

In order to gain knowledge about the actual performance increases that can be expected to be achieved from implementing occlusion culling, simulations using several different visibility calculation algorithms are performed. The occlusion culling algorithms that were implemented both take advantage of a hardware mechanism [18] that currently seems to offer the best solutions for online occlusion culling. By providing benchmarks for only these hardware-extension-dependent algorithms implies that many occlusion culling algorithms - those that do not make use of the hardware extension - are not considered. While this of course has an impact on the value of the simulation results, taking advantage of the hardware occlusion culling mechanism appears to be the most beneficial way of tackling the visibility problem and was deemed the most logical area on which to place the investigative focus.

The results of the simulation runs, which were performed using three-dimensional scenes that are similar to models used in actual industry applications, show that implementing online occlusion culling can offer quite significant performance increases over view frustum culling alone. The best performance was achieved using a somewhat complex algorithm [22], the implementation of which proved to be rather difficult. A simpler solution [11] also provided good results and may have the better performance-complexity ratio of the two.

Performance benchmarks were provided using two different hardware setups. In hind-

sight, this number was perhaps too small given the plethora of consumer graphics hardware currently available on the market. Another limitation is that the scenes used were rendered naïvely, without taking into account the additional complexity that actual applications usually associates with the scene geometry, such as textures and shaders. Taking these things into account might for example mean that the traversal order of the spatial data structures must be changed in order to minimize shader state changes, something that will almost certainly have a negative impact on visibility calculation performance. Therefore, the actual performance increases as measured in this thesis can only be seen as indicative of the gains achievable in an actual application.

A logical next step would be to rewrite the benchmarking software to more accurately reflect a real-world application. It would also be interesting to implement some solutions that do not depend on the hardware for performing occlusion culling in order to verify that using the available hardware mechanisms is more beneficial. As the field of occlusion culling research is a moving target, new algorithms could be implemented as they are presented to the visibility research community in order to continue the work of objective algorithm performance comparisons as begun here.

References

[1] Aila, T., Miettinen, V., Nordlund, P. 2003. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics, **22**, 3, pp. 792-800*. Available at http://www.tml.tkk.fi/~timo/publications/aila2003siggraph_paper.pdf .

[2] Aila, T., Miettinen, V. 2005. dPVS Reference Manual, version 6.10. Hybrid Graphics Ltd. Available at http://www.hybrid.fi/dpvs .

[3] Assarsson, U., Möller, T. 2000. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools, **5**, 1, pp. 9-22*. Available at http://www.ce.chalmers.se/staff/tomasm/pubs/vfcullbox.pdf.gz .

[4] Akenine-Möller, T., Haines, E. 2002. *Real-Time Rendering (Second Edition)*. A K Peters Ltd.

[5] Barron, D. 2002. Dissecting the ATI Radeon 9700 Pro. Available at http://www.firingsquad.com/hardware/radeon_9700/ .

[6] Bartz, D., Klosowski, J., Staneker, D. 2001. *k*-DOPs as Tighter Bounding Volumes for Better Occlusion Performance. *ACM SIGGRAPH Visual Proceedings 2001*. Available at http://www.gris.uni-tuebingen.de/~bartz/proj/large/paper/sketch2001.pdf .

[7] Batagelo, H. C., Wu, S-T. 2002. Dynamic Scene Occlusion Culling using a Regular Grid. *XV Brazilian Symposium on Computer Graphics and Image Processing, pp. 43-50*. Available at http://www.dca.fee.unicamp.br/~ting/Publications/P2001-2005/batagelo-wu-2002-3dvis.pdf .

[8] Bentley, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM, **18**, 9, pp. 509-517*. Available at http://portal.acm.org/citation.cfm?id=361007 .

[9] Bernardim, F., El-Sana, J., Klosowski, J. T. 2000. Directional Discretized Occluders for Accelerated Occlusion Culling. *Computer Graphics Forum, **19**, 3, pp. 507-516*. Available at http://www.cs.bgu.ac.il/~el-sana/Paper/ddo.pdf .

[10] Bittner, J., Wonka, P. 2003. Visibility in Computer Graphics. *Environment and Planning B: Planning and Design 30, 5, pp. 729-756.* Available at http://www.cg.tuwien.ac.at/research/publications/2003/Bittner-2003-VCG/TR-186-2-03-03Paper.pdf .

[11] Bittner, J., Wimmer, M., Piringer, H., Purgathofer, W. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. em Computer Graphics Forum (Eurographics 2004), **23**, 3, pp. 615-624. Available at http://www.cg.tuwien.ac.at/research/vr/chcull/ .

[12] Catmull, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. Thesis, Dept. of Computer Science, University of Utah, Salt Lake City, Utah.

[13] Cohen-Or, D., Chrysanthou, Y., Silva, C. T., Durand. F. 2003. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics, **9**, 3, pp 412-431*. Available at http://people.csail.mit.edu/fredo/PUBLI/surveyTVCG.pdf .

[14] Durand, F. 1999. *3D Visibility: Analytical Study and Applications*. Ph.D. Thesis, Université Joseph Fourier, Grenoble. Available at http://people.csail.mit.edu/fredo/THESE/vo.pdf .

[15] LiquidAir. "Combining Octrees and BSP trees.. is it possible?" 9 November 2002. GameDev.Net discussion forum. Accessed on 25 august 2006. Available at http://www.gamedev.net/community/forums/topic.asp?topic_id=123169 .

[16] Dirge. "Occlusion culling." Online posting, 27 April 2003. GameDev.Net discussion forum. Accessed on 24 August 2006. Available at http://www.gamedev.net/community/forums/topic.asp?topic_id=153497 .

[17] *GL_HP_occlusion_test*. OpenGL extension specification. Available at http://www.opengl.org/registry/specs/HP/occlusion_test.txt .

[18] *GL_NV_occlusion_query*. OpenGL extension specification. Available at http://www.opengl.org/registry/specs/NV/occlusion_query.txt .

[19] Glassner, A. S. 1984. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics & Applications, 1984, **4**, 10, pp. 15-22*.

[20] Greene, N., Kass, M., Miller, G. 1993. Hierarchical Z-buffer Visibility. *Computer Graphics (Proceedings of SIGGRAPH '93), pp. 231-238*.

[21] Gummerus, S. 2003. *Conservative From-point Visibility*. M. Sc. Thesis, University of Tampere. Available at `http://www.cs.uta.fi/research/theses/masters/Gummerus_Sampsa.pdf`.

[22] Guthe, M., Balázs, Á., Klein, R. 2006. Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. *Proceedings of the Eurographics Symposium*. Available at `http://www.cg.cs.uni-bonn.de/docs/publications/2006/guthe-2006-neal-optimal.pdf`.

[23] Hart, J. 2003. Ray-tracing in Graphics Hardware. *SPEC/GPC press conference, SIGGRAPH 2003*. Available at `http://www.spec.org/gpc/publish/hart.htm`.

[24] Hillesland, K., Salomon, B., Lastra, A., Manocha, D. 2002. *Fast and Simple Occlusion Culling using Hardware-Based Depth Queries*. Technical Report 02-039, Department of Computer Science, University of North Carolina at Chapel Hill. Available at `ftp://ftp.cs.unc.edu/pub/publications/techreports/02-039.pdf`.

[25] Klosowski, J. T., Silva, C. S. 2000. The Prioritized-Layered Projection Algorithm for Visible Set Estimation. *IEEE Transactions on Visualization and Computer Graphics, **6**, 2, pp. 108-123*. Available at `http://www.sci.utah.edu/~csilva/papers/tvcg2000.pdf`.

[26] Klosowski, J. T., Silva, C. S., Corrêa, W. 2002. Fast and Simple Occlusion Culling. *Game Programming Gems 3, Charles River Media*. Available at `http://www.sci.utah.edu/~csilva/papers/gpg3.pdf`.

[27] Klosowski, J. T., Silva, C. S. 2001. Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm. *IEEE Transactions on Visualization and Computer Graphics, **7**, 4, pp. 365-379*. Available at `http://www.sci.utah.edu/~csilva/papers/tvcg2001cr.pdf`.

[28] Kumar, S., Manocha, D., Garrett, W., Lin, M. 1999. Hierarchical Back-Face Culling. *Computers and Graphics, **23**, 5, pp. 681-692*. Available at `ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/VISIBILITY/backface.ps.gz`.

[29] Law, F-A., Tan, T-S. Preprocessing Occlusion for Real-Time Selective Refinement. *Proceedings of the 1999 Symposium on Interactive 3D Graphics, pp. 47-53*.

[30] Meißner, M., Bartz, D., Hüttner, T., Müller, G., Einighammer, J. 1999. *Generation of Subdivision Hierarchies for Efficient Occlusion Culling of Large Polygonal Models*. Technical Report WSI-99-13, WSI/GRIS, University of Tübingen. Available at `http://www.gris.uni-tuebingen.de/~bartz/Publications/paper/wsi99-13.pdf`.

[31] Müller, G., Fellner, D. W. 1999. *Hybrid Scene Structuring with Application to Ray Tracing*. Technical Report TUBS-CG-1999-05, Technical University Braunschweig. Available at `http://pengo.cg.cs.tu-bs.de/bs_root/lv/PubArc/tr/TR-tubs-cg-1999-05.pdf`.

[32] Omohundro, S. M. 1989. *Five Balltree Construction Algorithms*. Technical Report no. 89-063, International Computer Science Institute. Available at `http://www.icsi.berkeley.edu/ftp/pub/techreports/1989/tr-89-063.pdf`.

[33] Rubin, S., Whitted, T. 1980. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *Proceedings of ACM SIGGRAPH 1980, pp 110-116*.

[34] Samet, H. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

[35] Severson, K. 1999. *VISUALIZE Workstation Graphics for Windows NT*, HP product literature.

[36] Ulrich, T. 2000. Loose Octrees. *Game Programming Gems*, Charles River Media, pp. 444-453. Available at `http://tulrich.com/geekstuff/partitioning.html`.

[37] The Stanford 3D Scanning Repository. Available at `http://graphics.stanford.edu/data/3Dscanrep`.

[38] The Walkthrough Group at University of North Carolina - Chapel Hill. Power plant model. Available at `http://www.cs.unc.edu/~geom/Powerplant`.

[39] Zhang, H. 1998. *Effective Occlusion Culling for Interactive Display of Arbitrary Models*. Ph.D. Thesis, Department of Computer Science, University of North Carolina - Chapel Hill.

Appendix A

Sammanfattning på svenska

Av alla informationsteknologins delområden genomgår datorgrafiken, i synnerhet den tredimensionella grafiken, en speciellt snabb utveckling. Applikationerna inom detta område är sysselsatta med att rita ut eller *rendera* en tvådimensionell bild av en virtuell, tredimensionell värld på användarens skärm. Den tredimensionella världen med alla dess objekt kallas generellt för *scenen*. Termen *scenkomplexitet* används för att beskriva hur detaljerad scenen är och uttrycks ofta i antal trianglar. Scenkomplexiteten är i princip obegränsad; en scen kan alltid innehålla fler objekt och objekten kan göras mer detaljerade. Till exempel kan man åstadkomma tredimensionella modeller, innehållande miljontals trianglar, av verkliga statyer genom laseravbildning.

Även om grafikhårdvaran naturligtvis också har genomgått väsentliga kapacitetsökningar så gör den lika stora eller till och med större ökningen i scenkomplexitet att det finns ett verkligt behov av algoritmiska lösningar som kan snabba upp renderingen av en scen. Målet är alltid att erbjuda användaren en interaktiv upplevelse; för hög scenkomplexitet i kombination med långsam hårdvara och dåliga renderingsmetoder kan göra hela datorsystemet långsamt och svåranvänt.

Visibilitetsalgoritmer är centrala för renderingen av tredimensionell grafik. I många fall, speciellt i applikationer såsom spel där användaren kan röra sig fritt i en tredimensionell miljö, finns det många objekt som vid någon viss tidpunkt inte är synliga och således inte behöver renderas. Ett exempel på ett dylikt scenario vore en modell av en stad där användaren rör sig längs gatorna; de icke närliggande kvarteren liksom även byggnadernas interiörer är skymda från sikt av husens väggar. Ett annat exempel är en vanlig kopieringsmaskin där det komplexa maskineriet inuti är skymt av paneler. Att rendera dessa garanterat osynliga objekt kan bara ses som ett slöseri med processorkraft.

Modern grafikhårdvara har oftast en så kallad pipelinestruktur, där den råa datan som sänts till grafikkortet för rendering måste gå igenom många olika steg innan den är klar att

visas på användarens skärm. Om de delar av scenen som är osynliga kan gallras bort redan före grafikhårdvaran tar över databehandlingen kan man förvänta sig att uppnå märkbara prestandaökningar.

Visibilitet i tredimensionella miljöer är ett populärt forskningsområde som dock lider av en brist på objektiva jämförelser mellan de många olika tillvägagångssätten [13]. Syftet med denna avhandling är att erbjuda en åtminstone partiell lösning till detta problem genom att beskriva och jämföra en mängd olika algoritmer. En grupp speciellt intressanta algoritmer beskrivs mer detaljerat och implementeras även för att möjliggöra direkt prestandamässig komparering. Grundidén är att göra det lättare för utvecklare av mjukvara för tredimensionell grafik s att se vilka metoder som det kan löna sig att utforska vidare och eventuellt implementera.

Avhandlingens tyngdpunkt ligger hos en familj algoritmer som alla använder sig av en hårdvaruextension utvecklad enkom för att kunna utnyttjas för visibilitetskalkyler. En hårdvarulösning är normalt snabbare än en motsvarande mjukvarulösning eftersom den statiska hårdvaran är lättare att optimera. Detta gör det logiskt att anta att dessa algoritmer har ett visst inbyggt försprång över andra metoder. Hårdvaruextensionen som algoritmerna kräver finns i dagens läge tillgänglig på i snart sett all kommersiellt tillgänglig grafikhårdvara.

De tillvägagångssätt som jämförs här strävar alla efter att vara allmänt tillämpningsbara; det finns även många algoritmer som endast fungerar såtillvida att speciella förhållanden råder, exempelvis att den tredimensionella scenen är en arkitektonisk modell bestående av rum, dörrar och korridorer. Istället för dessa specialmetoder, som ofta kräver betydande förhandsprocessering av scenen, har tyngdpunkten lagts på algoritmer som kan hantera visibilitetsproblemet i realtid, dvs. utan nämnvärda försteg.

För att kunna beskriva de algoritmer som har valts ut för jämförelse är det nödvändigt att introducera vissa fundamentala koncept som all tredimensionell grafik bygger på.

Tredimensionella scener

En tredimensionell scen består av ett antal *objekt*, till exempel hus, träd eller bilar. Dessa objekt kan ha blivit skapade med hjälp av något slags modelleringsverktyg, alternativt kan de vara slutprodukten av någon form av datainsamlingsprocess, såsom magnetröntgen, en laserytavläsning eller satellitbildupptagning.

Objekten består i sin tur av *ytor* eller *primitiver*, i normala fall trianglar. Dessa primitiver utgör objektets yta. Genom att använda fler primitiver kan man ge ett objekt fler detaljer. Om man antar att en primitiv är det samma som en triangel kan dess area fullständigt

beskrivas med tre *punkter*. Dessa punkters positioner samt vilka punkter som länkas ihop till vilka ytor är den minsta mängd information som krävs för att göra renderingen av ett objekt möjlig. I verkligheten finns det oftast mycket mer information tillgänglig om ett givet objekt, såsom till exempel dess färg.

Föruom ett objekts punkter, som bestämmer objektets utsträckning i *objektrummet*, är en transformationsmatris som bestämmer objektets position, rotation och skala i *världsrummet* även ofta associerad med objektet. Genom att använda transformationsmatriser för positioneringen av de olika objekten i en scen kan man rita ut samma objekt många gånger om, på olika platser, bara genom att uppdatera transformationsmatrisen.

Alla de objekt som scenen består av kallas kollektivt för *scengeometrin*. Denna kan ytterligare delas upp i *statisk* och *dynamisk* geometri. Den statiska geometrin är den del av scenen som inte förändras med tiden, såsom vägar och hus. Till den dynamiska geometrin hör objekt som kan förflytta sig eller ändra form, till exempel en bil som kör omkring på en stads gator, i vilket fall gatorna skulle höra till den statiska geometrin. Fokuset i denna avhandling är på den statiska delen av geometrin.

När en scen med icke-trivial komplexitet finns tillhands uppstår ett behov av att organisera geometrin. För detta ändamål används spatiella datastrukturer.

Spatiella datastrukturer

En spatiell datastruktur organiserar scengeometrin i antingen två eller tre dimensioner, vanligen hierarkiskt. Dessa strukturer är viktiga därför att de gör det möjligt att hantera scengeometrin på en mer abstrakt nivå än de individuella objekten och deras ytor. Det är viktigt att inse att en spatiell datastruktur inte utgör en tredimensionell scen i sig utan bara är ett sätt att behandla den underliggande geometrin.

Alla spatiella datastrukturer delar upp scenen i mindre delar som kallas *celler*. I en hierarkisk struktur organiseras cellerna som noder i ett träd där rotnoden innehåller hela scenen och noder djupare ner i trädet omsluter successivt krympande delar av scenen. Hierarkiska strukturer är speciellt lämpade för bruk i visibilitetskalkyler eftersom en bortgallring av en osynlig nod automatiskt gallrar bort dess undernoder också. Detta gör att man oftast kan gallra bort stora delar av scengeometrin med endast några enstaka kalkyler.

De flesta spatiella datastrukturer är rätt tidskrävande att skapa. Detta görs därför oftast före den verkliga renderingen sätter igång, när applikationen startas upp. Det är dock oftast möjligt att uppdatera strukturerna efterhand om scenen innehåller dynamisk geometri. Vissa strukturtyper är bättre lämpade för dynamiska uppdateringar än andra; om en ändring i den underliggande scengeometrin till exempel betyder att hela datastrukturen måste göras om från början betyder det att strukturen inte är särskilt väl lämpad för rörliga

objekt.

En fördel som en spatiell datastruktur för med sig är att den möjliggör en genomgång av scenen från närliggande objekt till avlägsna objekt [4]. Detta betyder att de objekt som ligger närmast synpunkten renderas först, sedan objekten bakom dem och så vidare. Detta tillvägagångssätt är av stor vikt för framgångsrika visibilitetskalkyler, vilket kommer att visas senare. En spatiell datastruktur erbjuder den för detta ändamål nödvändiga informationen om vilka objekt som ligger nära varandra och det är oftast möjligt att lätt hitta ett sätt att genomföra en scengenomgång av denna typ [2].

Visibilitet i tredimensionella scener

Från en given synpunkt inne i en tredimensionell scen kan man dela upp scengeometrin i två delar, den synliga och den osynliga delen. Inom den osynliga delmängden objekt kan man urskilja ytterligare två kategorier; objekt som är osynliga eftersom de ligger utanför användarens synfält samt objekt som är osynliga eftersom de blockeras av något annat objekt som ligger närmare synpunkten. Om man behandlar de enskilda primitiverna som objekten består av hittar man ytterligare en kategori osynliga element; primitiver som är osynliga eftersom de är belägna på den sida av objektet som vetter bort från användaren. Ett exempel på denna sista kategori vore månens baksida.

Av de ovan nämnda kategorierna är den mest intressenta den som innefattar objekt dolda av andra objekt. Objekten i denna kategori sägs vara övertäckta eller *ockluderade*. Den största delen av dagens forskning inom området, liksom denna avhandling, är inriktad på att hitta effektiva metoder för bortgallringen av ockluderade objekt.

Det finns många olika tillvägagånssätt för att gallra bort ockluderade objekt. En intressant grupp metoder använder sig av en nyutvecklad hårdvaruextension för att identifiera vilka objekt som kan plockas bort på detta sätt. Två algoritmer ur denna grupp som verkar ha speciell potential är “Coherent Hierarchical Culling” [11] och “Near Optimal Hierarchical Culling” [22].

Hårdvaruextensionen som dessa algoritmer använder sig av fungerar genom att returnera storleken av arean på skärmen som *skulle ha* modifierats om ett givet objekt skulle ha renderats. För ett ockluderat objekt, som inte modifierar skärmens innehåll eftersom det inte är synligt, returneras således värdet noll. För att utföra detta test kan cellerna i den spatiella datastrukturen användas; om en nod testas och visar sig vara ockluderad kan man gallra bort både den och alla dess undernoder. Genom att använda den spatiella datastrukturens noder kan man i de första skedena av visibilitetstestningen undvika att rita ut den ofta komplicerade underliggande scengeometrin. Cellerna i datastrukturen kan

oftast representras som enkla lådor och för dessa går det i regel snabbt att utföra olika visibilitetskalkyler.

Problemet med extensionen för gallring av ockluderade objekt är att en viss latens mellan att funktionen anropas och svaret blir tillgängligt. På grund av denna latens går det inte att helt enkelt utföra ocklusionstestet på alla noder i den spatiella datastrukturen. Funktionsanrop bör istället göras endast för de noder som med stor sannolikhet är ockluderade medan noder som antagligen är synliga bör ritas ut direkt utan att hårdvarutestet utförs. Hur detta problem löses är den största skillnaden mellan CHC- och NOHC-algoritmerna. NOHC-algoritmen, som har avsevärt högre komplexitet, gör en något mer sofistikerad analytisk undersökning av hur sannolikt det är att en given nod är ockluderad.

För att kunna mäta hur stor prestandaökning man kan förvänta sig av att implementera gallring av ockluderade objekt samt för att objektivt jämföra CHC- och NOHC-algoritmernas effektivitet har en testapplikation implementerats. Resultaten som erhållits genom att använda de två algoritmerna i renderingen av olika realistiska scener, exempelvis en detaljerad modell av ett kolkraftverk, visar att båda algoritmerna kan snabba upp renderingsprocessen avsevärt; jämfört med att inte över huvud taget använda gallring av ockluderade objekt går renderingen uppemot 80-100 % snabbare. Vidare visar resultaten att NOHC-algoritmen erbjudare 20-30 % högre prestanda än CHC-algoritmen för scener med hög komplexitet, dvs. för scener i vilka en stor del av geometrin är ockluderad från en given synpunkt. För scener med låg komplexitet var skillnaden mellan NOHC- och CHC-algoritmen mindre, något som troligtvis hör ihop med den något komplexa implementeringen av NOHC-algoritmen; det är lätt att göra fel när NOHC-algoritmen implementeras och dessa fel märks genast i algoritmens effektivitet. Om man tar denna komplexitet i beaktande är det möjligt att CHC-algoritmen i många fall kan vara ett bättre val, speciellt om tiden som finns disponivel för implementering är knapp. Är däremot högsta möjliga prestanda ett kriterium, samtidigt som tillräckligt med tid för implementeringen finns, är NOHC-algoritmen sannolikt ett bättre val.